Programmable Synthesis of Element Textures and Application to Cartography

Hugo Loi

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques-Informatique**

Arrêté ministériel : todo

Présentée par

Hugo LOI

Thèse dirigée par **Joëlle THOLLOT, Thomas HURTUT et Romain VERGNE**

préparée au sein **du Laboratoire Jean Kuntzmann (LJK)**
et de **l'École doctorale EDMSTII**

Programmable Synthesis of Element Textures and Application to Cartography

Thèse soutenue publiquement le **16 octobre 2015**,
devant le jury composé de :

Craig S. KAPLAN
Associate Professor, University of Waterloo, Rapporteur
Sylvain LEFEBVRE
Research Associate, Inria Nancy, Rapporteur
Cyrille DAMEZ
Software Engineer, Allegorithmic SAS, Examinateur
Sidonie CHRISTOPHE
Research Associate, French National Mapping Agency (IGN-France), Examinateur
Joëlle THOLLOT
Professor, Grenoble Université, Directeur de thèse
Thomas HURTUT
Assistant Professor, Polytechnique Montréal, Directeur de thèse
Romain VERGNE
Assistant Professor, Grenoble Université, Directeur de thèse
Marie-Paule CANI
Professor,Grenoble Université, Présidente

ABSTRACT

This thesis introduces a programmable method to design 2D arrangements for *element textures*: Textures made of small geometric elements. These textures are ubiquitous in numerous applications of computer-aided illustration. Our approach targets technical artists who will design an arrangement by writing a script. Such scripts are based on three types of operators dedicated to the fast creation of stationary arrangements:*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. 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. In particular, we introduce an additional contribution to this main framework: We present a method that demonstrates interactive performances at synthesizing (infinite) regular and semi-regular arrangements using the *Tiled Planar Map*, a novel data structure that contains few yet sufficient information for representing these arrangements. Finally, we show how to extend our system for designing spatially-varying textures using control fields. In particular, we present a practical application to cartography in which we collaborated with the French National Mapping Agency (IGN-France) to automatically synthesize hatchings in rocky mountain areas of topographic maps.

ACKNOWLEDGEMENTS

Hi there. I have no idea what this section will look like when it is finished, since I am working on it the exact opposite way than what you're supposed to do for scientific documents: write a draft, submit it to a few colleagues, add a bullet list of new ideas to insert, write again, revise, resubmit, back and forth until the thing is considered perfect. Actually I got twenty minutes before my laptop battery runs out, before which I have to summarize four years of a fun and fulfilling life. So my updated strategy is writing once, hitting "save" and upload. The result is not going to be perfect. In the case you opened this document to figure out if you are cited or not in my acknowledgements, you will just have to endure it. This is known as the "captive audience principle". If you want to look up for your name using your digital reader then proceed at your own risk: it may be misspelled or replaced by an embarrassing metaphor. Now let's start.

The first to go is Quentin Doussot, a guy I knew from my previous life. At the time I was shy and stubborn so having a welcoming, friendly hand to shake every morning saved me in the first months. This man walked in socks around the lab, which was very comforting.

Then comes Laurent Belcour which taught me the power of words: a simple question asked about some Fourier stuff and he went right into an afternoon-long religion war with Eric, who comes later. You got to know which topic is sensitive for whom: Fourier is definitely sensitive for Laurent.

I never knew where Fabrice Neyret came from. Maybe he was a mountain hermit who someday got trapped inside Inria and never found the exit door. Since then he lurks in the lab corridors, watering dry students with generous loads of his knowledge. Make sure to sleep a good night before you go and ask for advice, otherwise you won't remember everything.

Once upon a time, Jeremy Jaussaud said something brief which made several people happier on the long run. Cool superpower he got there. Apparently he likes doing game jams as well, which is nice - I never got to check that out though.

A lot of people made Inria a great place in the daylight, a few of them were still there to keep it great overnight. In particular, Eric Heitz, Benoit Zupancic and Alban Fichet deserve special thanks for making this half of my life shine like sunlight. Cokes, milk, "that other SOB", Micko Black and zeu Lounge will remember you for eternity.

Also I address special thanks to Zeno Loi that helped me make whiteboards alive over one magic night, which was sort of a real-life Ib.

To keep it to the serious side, this PhD would never have been finished and would never have benefited from such inspiration on day only. I wish to thank FranÃ§ois Sillon and Nicolas Holzschuch for understanding this side of Science and letting Inria live at night, just like all other high-class international computer science labs.

A guy named Leo Allemand-Giorgis saved my back several times, along with another guy named Pierre-Luc Manteaux. Without them a short animated movie named "Dessins de Mome" would have been let unfinished. Instead it was a mediocre piece of art born out of an amazing adventure. I am glad we did it.

On the topic of watching animated movies rather than drawing them, a good one is Paprika. Check it out or ask Armelle Bauer if you don't have the DVD. She has pretty good other ones too.

I feel like Pierre Benard should be somewhere in here. In the end I just know him as a cool guy just like all the other Animarians (hey there), but he also happened to write my favorite computer graphics paper. That's pretty cool to me.

On the same line, I would like to thank Szymon Rusinkiewicz for his software RTSC. Indeed it was a lot of fun to procrastinate playing with RTSC. That participated heavily to awaken curiosity in me, which I still use to enjoy stuff I did not expect to work on.

I dreamt working for Disney since I was a kid. Wojciech Jarosz allowed me to do that, which is incredibly lucky I must say. I remember how shaky I was on the first skype interview. Just after that I went and work with Adam Finkelstein in Princeton, which sounded like a double achievement to me. In the end it turned out that these two places were amazing, true paradises for research. My conclusion was that I wanted to start my company rather than continuing research. Looks like finding what you need requires iterations, apparently.

What truly made me happy and allowed me to perform accomplishing work were the people I met there; in particular Antoine Milliez, who is nothing less than a never-depleted source of enthusiasm. Someone find a way to distribute Antoine energy in cans; we'll be done with war and depression.

Benoit Arbelot, I am talking to you. We are big boys now, we shall finish this Justine DLC soon.

Cyril Soler, thanks for this memorable latex gloves party.

Guillaume Loubet, keep this ability to wrap up trolls and emotions in a few piano chords.

Working with the French National Mapping Agency was a great experience, in particular with Sidonie Christophe and Mathieu Bredif. These weeks of diving in the cartography world have been great also thanks to Jean-Sebastien Vinals, who is to be thanked for many other reasons. But since this is a set of citations rather than an exhaustive wrap-up, people will not know the extent of how Jean-Sebastien prepared me to even start this very PhD.

Around the thesis world, are also to thank Pop, Moune and Wawan. These enigmatic characters are true legends to me. As well is to thank Agathe, who suddenly appeared and made my days ten times better. I guess this document benefited from her presence as much as my work.

You were not cited so far? You will probably not be cited in the last two paragraphs even though you probably have very good reasons to be in here. The twenty minutes are about to run out. I still love you. It's ok.

Finally, Romain Vergne, Thomas Hurtut and Joelle Thollot were great adventure mates, inflexible teachers and enlightening supervisors. Believe it or not: Not all researchers are good mentors. Based on a relatively large study among the PhD students I met during the past four

years, it turns out that I was quite lucky to go with these three. Hey, maybe every single PhD student thinks the same. That would be awesome!

Finally finally, the guy who encouraged me into doing a PhD is none other than my father. “Try it, you’ll see this is amazing!”. I was not as enthusiastic. I tried. I’m happy I did. Cheers!

TABLE OF CONTENTS

CHAPTER

1

INTRODUCTION

This PhD thesis introduces a programmable model that helps to design *element textures*: Patterns made of small geometric elements. These textures are ubiquitous in a number of artistic contents such as 3D scenes, comic books and technical illustrations: Figure 1.1 shows examples of 3D models exhibiting element textures and Figure 1.2 shows examples of element textures used in illustrations.

FIGURE 1.1 – *Element textures on 3D models.* *Numerous natural and manufactured entities exhibit patterns made of small geometric elements, also called element textures. When making artistic representations of these objects, such textures have to be produced by a designer. The arrangements of elements in each texture were designed and generated using the tool we present in this manuscript. (image courtesy from Guillaume Loubet)*

1.1 ELEMENT TEXTURES

From the artist perspective, the word *texture* refers to a repetitive pattern aimed at covering a surface. Various kinds of assets fall in this category, including wallpapers, materials on 3D surfaces and illustration patterns such as in topographic maps or Computer-Assisted Design

tools. Textures carry a lot of the feeling artists want to express about the look of each surface. Therefore they also make a large part of the experience for people watching animated movies, artworks or technical illustrations.

Artists often produce textures manually, which implies a major pain point: workload. Art in general is well-known for asking quantities of workload, but textures hold a particular reponsibility for that: it is much longuer to draw a pattern all over a surface than just sketching a contour. This task is also repetitive: once the artists decided how they want to draw the texture, then they have to apply their idea consistently all over the surface. If they change their mind then they have to start over again.

Repetitive tasks are good candidates for computer assistance. During the two past decades, texture design has received much attention from research and software industry. Today, many solutions attempt to help artists designing large textures without drawing everything by themselves. Among these proposals, techniques for combining pixel noises, color maps and other operators as nodes of a graph have been widely accepted. Allegorithmic's software Substance proved the need for such computer-assisted tools and is now used in 80 percent of major game development studios [1].

However, many textures are not combinations of pixel noises. These textures still require lots of workload to design and to produce. We observe that a significant amount of these time-consuming textures are composed of separate geometric elements: textures representing separate objects such as leaves, bricks, cables, textures containing recognizable geometric features such as cracks, medium-driven textures such as pen-and-ink hatchings, and finally textures representing precise semantics such as terrain types in cartography or materials in Computer-Assisted Design. In this manuscript we will refer to these as *element textures*. Element textures are mandatory to 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 needed 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.

We believe that there are only a few steps towards a complete computer-aided tool for designing element textures. Let's review all the steps involved in the production of an element texture: creating geometric elements, arranging them together, choosing style attributes for each element and finally rendering the elements regarding the chosen style. Actually, almost all the steps of their production are handled by existing industrial or academic software. Elements themselves can be drawn manually with softwares such as Inkscape or Adobe Illustrator. They can also be generated with computer-assisted techniques [BA06, HL12, CK14]. Stylization and rendering geometrical data can be performed in Illustrator as well, or with more advanced research work [Her02, EWHS08, GTDS10, DiV13]. In this work we address the remaining problem of spatially **arranging** existing elements so as to fill a given region. The design of such arrangements is a key component of element textures design in that it describes the geometric content of the texture. By finding a relevant computer-aided way to arrange elements, we hope that a first version of a complete workflow will be feasible. The ideal goal is to lighten the repetitive tasks of professional artists in various domains, which would widen the room allowed to inventiveness: everyone is more likely to iterate over several trials when they can more easily

1. `www.allegorithmic.com`

implement each idea. Another desired consequence is to allow more casual artists to step into texture design and let everyone enjoy their creative inputs.

1.2 A Bill of Specifications for Computer-Aided Arrangement Design Tools

A computer-aided design tool for the production of 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 1.2 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 of as being based on multi-scale arrangements.

— *Usability*. Experienced users should be able to quickly and easily design or edit arrangements. A canonical and intuitive set of operations should also be provided to ensure a steep learning curve of the design tool.

— *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 the above properties.

Textures are specific since they are repetitive, enforcing their perception as a whole [TG80]. In practice this characteristic can be formalized as the result of a *stationary* process, meaning that the spatial statistics of an arrangement should not depend on its spatial location. In this manuscript, we will first consider textures under this property of stationarity (Chapters 3,4). Afterwards we will address *spatially-varying textures* which are locally repetitive while following low-frequency variations over space.

1.3 Our Approach

In this work 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 [Coo84], modeling [MWH*06], stylized rendering of 3D scenes [GTDS10, EWHS08] and motion effects [SSBG10]. As in these works, we target artists having programming abilities such as technical directors. In Chapter 3 we present the first programmable design tool dedicated to the creation of stationary arrangements while satisfying the four properties defined above.

To build this 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 texture design such as in Allegorithmic's Substance. 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 *planar maps* [BG89], 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 planar map is initialized with stationary partitions such as a grid; (2) instead of filters, local geometric transformations are next applied to refine the partitions; (3) merging operators finally allow multiple arrangements to be combined 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, this method can easily be extended by adding new operators as long as they satisfy the conditions that preserve stationarity.

We envision this design model as being extended towards a more interactive tool. On top of our core contribution, in Chapter 4 we present a method that synthesizes infinite regular and semi-regular arrangements at interactive rates. We introduce *Tiled Planar Maps*, a new data structure that contains few yet sufficient information for representing these arrangements.

Finally in Chapter 5 we show how to extend our system for creating spatially-varying textures by controlling the designed arrangements with user-provided control fields. From this extension, we present a practical application to cartography made in collaboration with the French National Mapping Agency (IGN-France): In this application, we designed and synthesized automatically hatchings that represent rocky mountain areas in topographic maps.

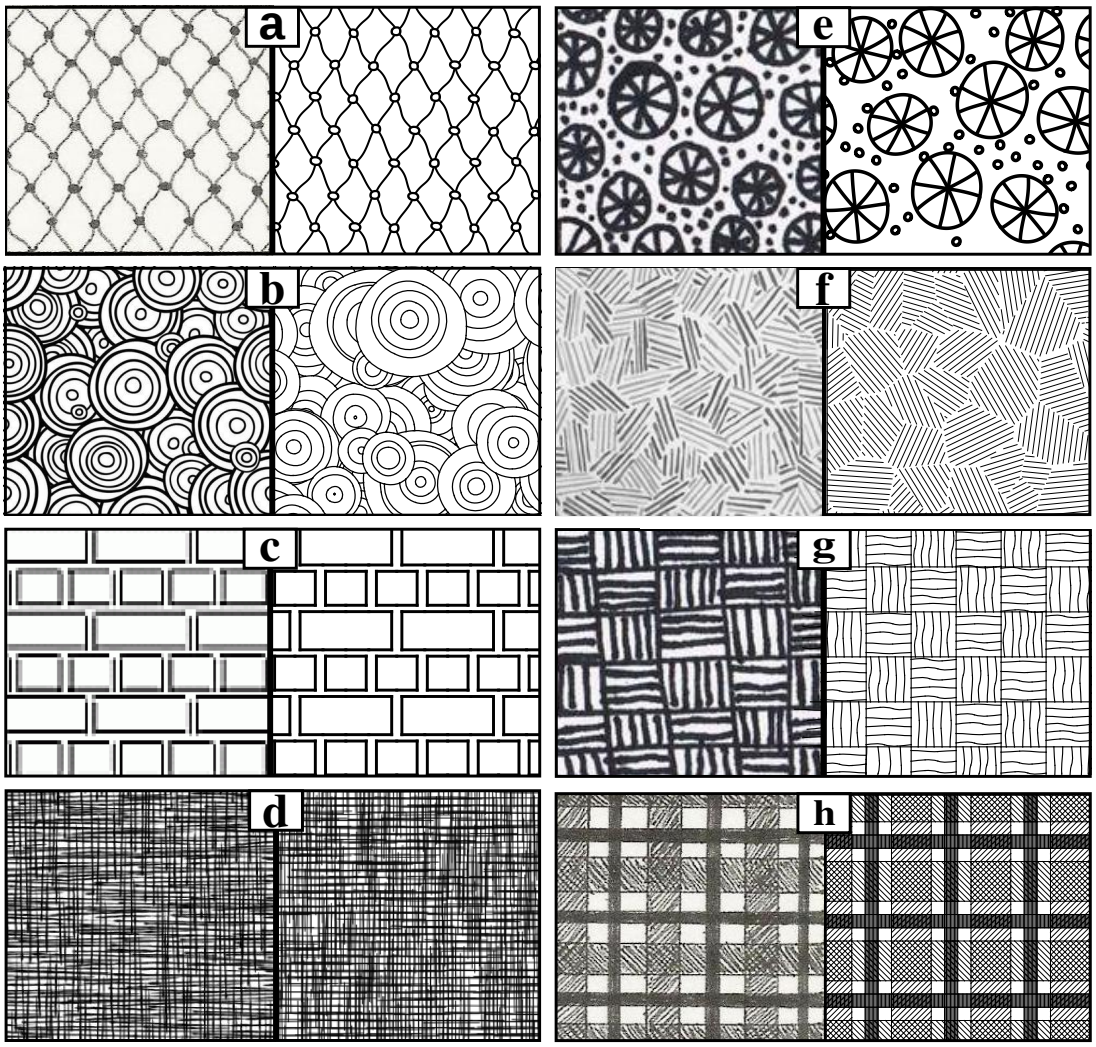


FIGURE 1.2 – ***Element textures commonly used.*** *Distributions with (a) contact, (b) overlap and (c) no adjacency between elements. (d) Overlap of two textures creating cross hatching. (e) Non overlapping combination of two textures. (f,g,h) Complex element textures with clusters of elements. For each example, we show a hand-drawn image (left), and our synthesized reproduction of its geometric arrangement (right). — Image credit: these textures can be found in professional art (d,g,h) [Gup97], casual art (a,e,f) [*`profusionart.blogspot.com`*, *`hayesartclasses.blogspot.com`*], technical productions such as Computer-Assisted Design illustration tools (c) [*`www.compugraphx.com`*], and textile industry (b) [*`www.123stitch.com`*].*

CHAPTER

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 [EMP*02], pixel-based textures lose 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 lose 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 exemplar 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 exemplar 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 1.2. We tested four recent methods [IMIM08, HLT*09, MWT11,

LGH13] and we observed limitations controlling contact or overlap (Figure 2.1(a)), regularities such as alignments (Figures 2.1(b) and 2.1(c)) and multi-level arrangements (Figure 2.1(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 [IMIM08] is not adapted to strongly anisotropic elements. The perceptually-based approach of [BBT*06] is also limited to the synthesis of simple strokes and irregular patterns. Similarly, bounding boxes [HLT*09] or sampling [MWT11, XCW14] reduce control on adjacency (Figure 2.1(a)). The proxy geometries introduced in [LGH13] 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 [YBY*13] for arrangements of tiles, but we are looking for a more general approach.

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 [1] propose methods for tiling small user-drawn arrangements. More complex stand-alone algorithms can synthesize uniform distributions efficiently [HHD03, LD05]. 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 are usually not easily extendable. Typically Figure 2.1(d) would be hard to do with such approaches because it mixes regular and random distributions.

2.3 Procedural Modeling

In this section, we present several inspiring procedural methods, coming from fields other 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 [Lin68] were used early in computer graphics to model plants [PL96]. 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 [SGSG71, WWSR03, MWH*06]. Like more general context-dependent growth systems [WZS98, MM12], 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 [Kap10] and Escher construction operators [Hen02]. These models are specific to their respective application fields, which limits their expressiveness. However they are similar to our

1. www.colourlovers.com/seamless

approach in the sense that they locally apply geometric transformations to an initial partition. Our approach targets general stationary arrangements.

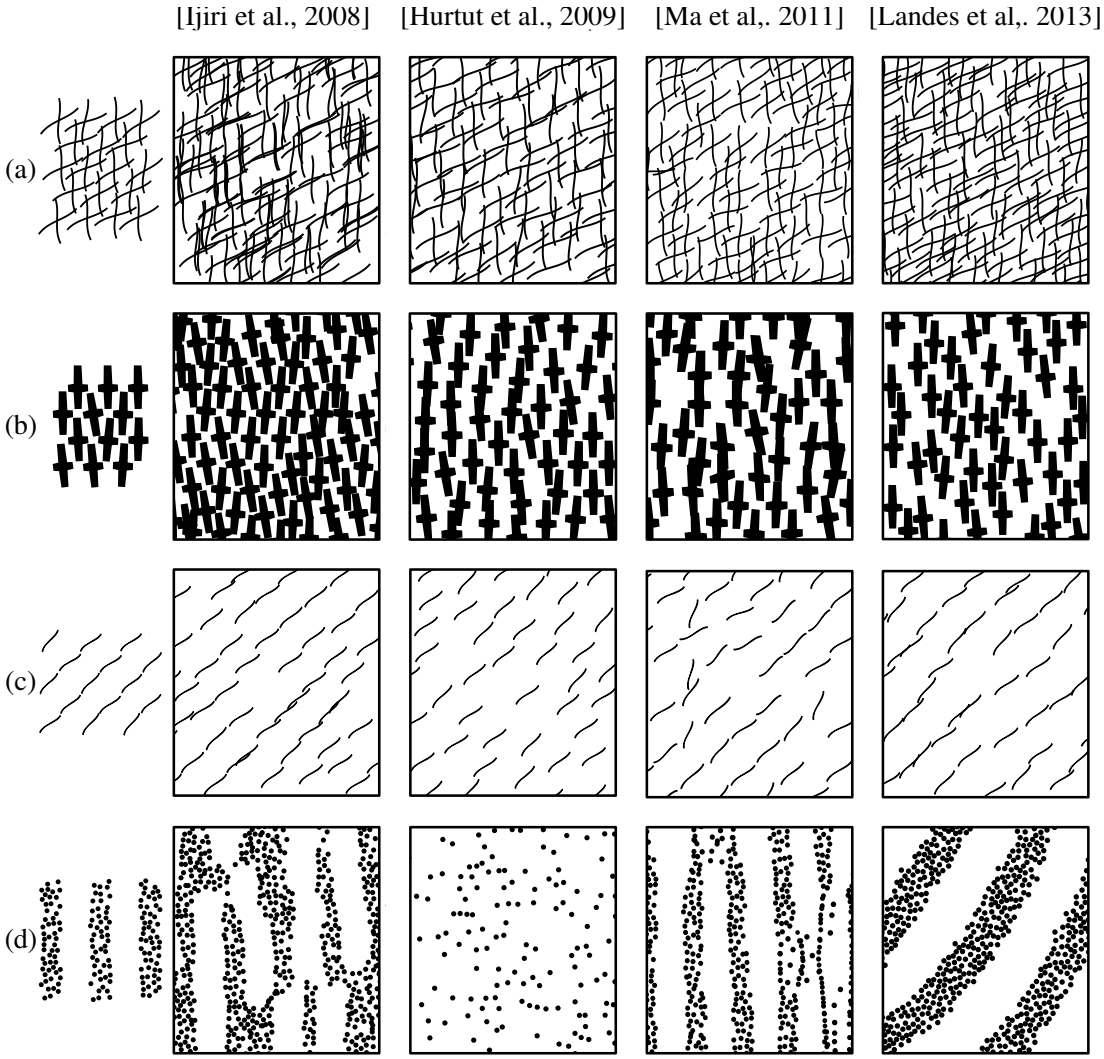


FIGURE 2.1 – ***Example-based methods' limitations.*** *Input exemplars are shown on the left. Synthesized results from previous methods [IMIM08, HLT*09, MWT11, LGH13] are shown on the four right columns.* ***(a)*** *A bimodal hatching, explicitly cited as one of the last limitations in [HLT*09]. While 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 [MWT11], 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 [IMIM08] 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 [HLT*09, LGH13]. 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.*

CHAPTER

3

PROGRAMMABLE DESIGN OF STATIONARY ARRANGEMENTS

3.1 OVERVIEW

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 [Jul81] 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, for instance by 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 stationary arrangements (see for instance the subscale stripe arrangements in Figure 1.2(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.

Merging 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.

Overview example. Figure 3.1 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).

3.2 DATA STRUCTURE

We represent our arrangements of geometric elements as a collection of curves embedded into a planar map: a topological modeling tool introduced in [BG89] for representing drawings. This structure contains vertices (intersection points), edges (pieces of curves linking vertices) and faces (2D domains enclosed by edges). Spatial adjacency between these three types of cells can be easily handled, providing an easy access to precise topological relations between them 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 3.2). 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. They will typically be used to select a subset of cells when needed and are set by the user at the initialization step.

Face labels reconstruction. When modifying or combining planar maps, labelling has to be conserved. We adopt the same solution as [ASP07]. 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 [FHW12] and use exact arithmetic and geometry with rational coordinates to avoid any topological artifacts due to numerical imprecision.

```
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)
```

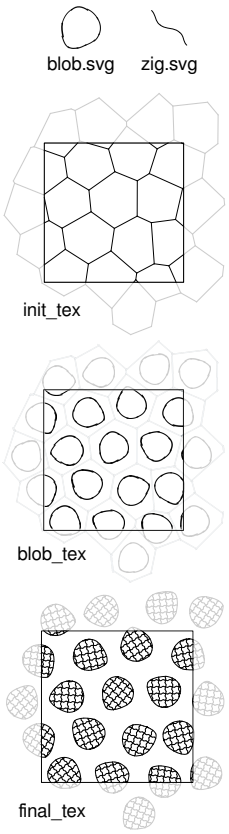
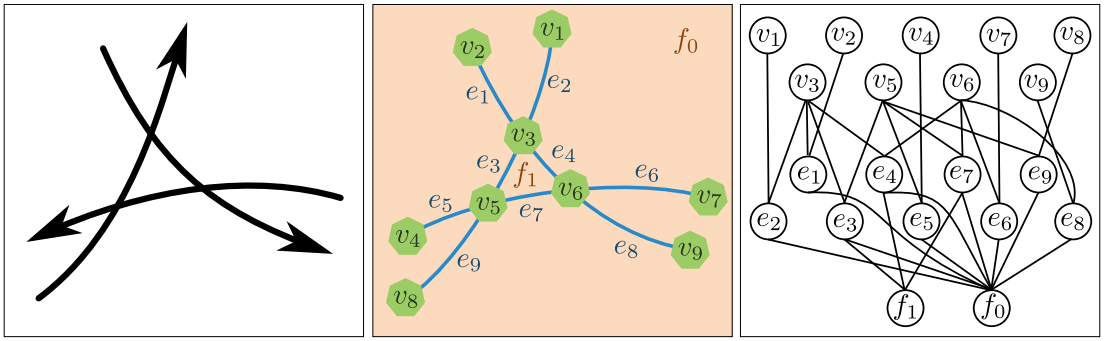


FIGURE 3.1 – *An example of a script and its output. Left:* *A script based on two imported SVG elements (a blob-like shape and a small stroke) and three user-defined local mappers to control local features.* ***Right:*** *The output is a two-scale arrangement. We show here two imported elements, followed by the two intermediary results and the final output. The square window represents the filled output region: Transparent lines are shown for clarity, yet they are not in the actual arrangement.*

3.3 PARTITIONING OPERATORS

The first step in the design of an arrangement is to choose a partition to define its global structure. Such partitions must be stationary and should hold a regular or non-regular global



Three oriented curves | Induced planar map | Incidence graph

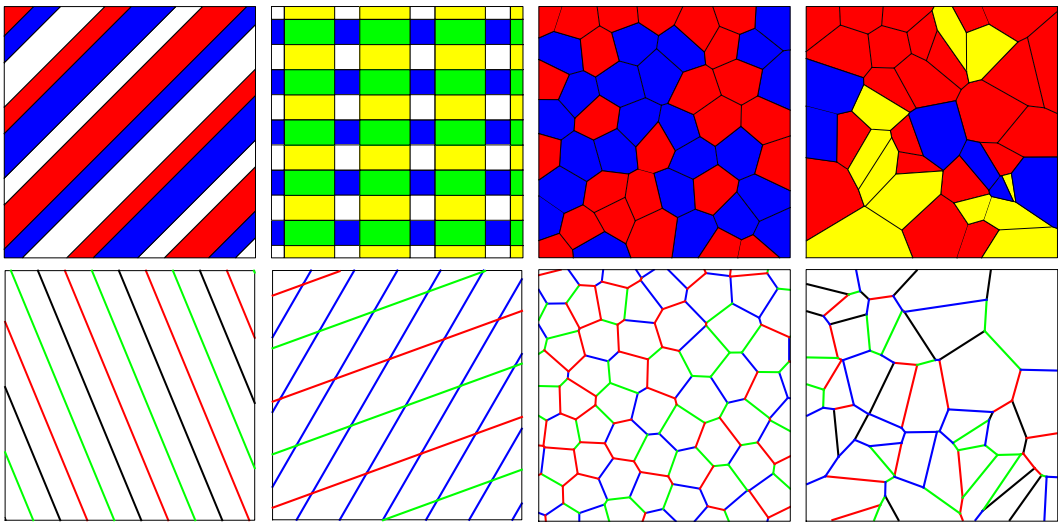
FIGURE 3.2 – ***Planar map representation. Left:*** *Three oriented curves.* ***Middle:*** *Induced planar map composed of nine vertices, nine edges and two faces. The face f_0 is unbounded. Edges are oriented accordingly to their originating curves. For example, e_1's source vertex is v_2 and its target 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.*

structure. These partitions will be extensively refined by defining local transformations using mappers, as presented in Section 3.4. 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.

We propose four partitioning operators that allow to design regular and non-regular partitions of the input region. These operators, in addition to a few others that let specify partition labels and properties, are recalled in the Table 3.2 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 3.3(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 3.3(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 3.3(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 probabilities used to randomly assign labels to faces and/or edges (Figure 3.3(c,d)).



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

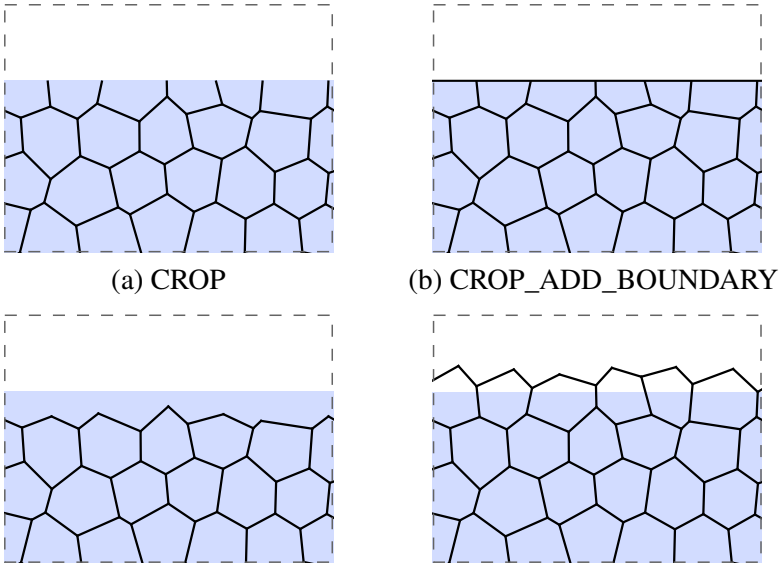
FIGURE 3.3 – ***Available types of partition.*** *When designing an arrangement, the first step is to choose a type of partition among four possible ones, whether it is a regular (a,b) or a non-regular partition (c,d). Colors denote assigned labels to faces (top) or edges (bottom). We vary the width 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.*

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 3.4). 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 prefers 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.

3.4 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.). Finally, a mapper 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 execu-



(a) CROP (b) CROP_ADD_BOUNDARY

(c) KEEP_INSIDE (d) KEEP_OUTSIDE

FIGURE 3.4 – *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).*

tion is handled by mapping operators. We provide one mapping operator per cell's type: "MapToVertices", "MapToEdges" and "MapToFaces" (Table 3.3 in appendix). A 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 used as input to another mapping operator in order to generate more complex patterns.

3.4.1 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.

3.4.2 Programming Mappers

We provide a set of low-level built-in operators specifically designed to program mappers, given in Table 3.4 of the appendix. All the examples shown in this chapter have been created with this simple operator set:

— *Incidence* operators are dedicated to access all the information stored in 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.
— *Label* 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 3.6 of the appendix).

3.4.3 Using Mappers

A typical use of mapping operators is to modify an original partition, for instance by removing or modifying some cells, then placing some new elements possibly controlling their adjacency. One can stop here or continue to map elements until reaching the desired arrangement. We show four examples in Figure 3.5 to illustrate the variety of effects a mapper allows to create on the partitions. In (a), a grid is locally modified to successively create triangular and hexagonal partitions that could themselves be used as starting points for other mappers. The second example (b) leverages labels and the precise matching of curve endpoints to create a puzzle-like brick wall arrangement. The third one (c) uses single point matching and location operators to create a uniform distribution of rosette shapes. In the last example (d), 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 3.1).

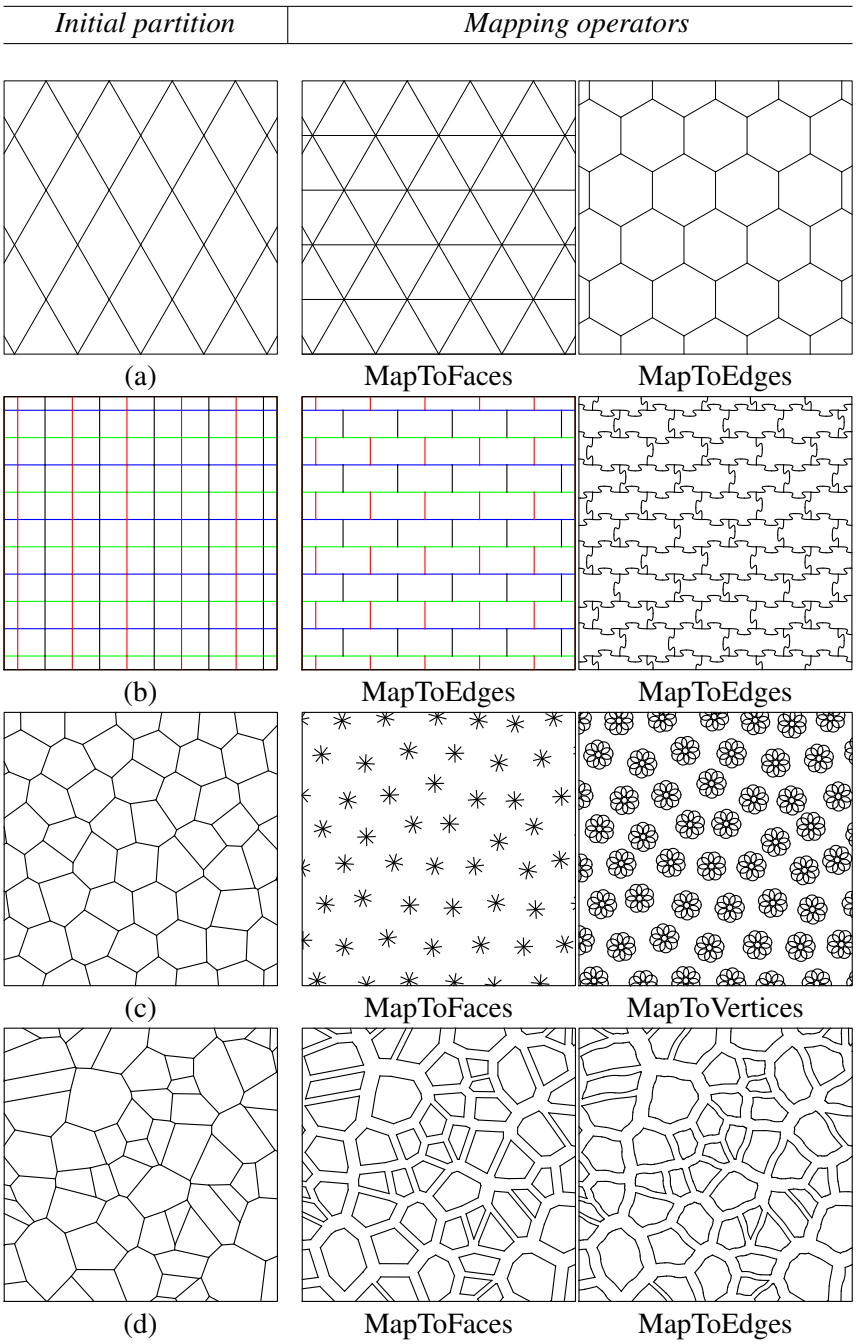


FIGURE 3.5 – *Using mappers.* *Four examples of mappers effect on initialized partitions.* ***(a)*** *A grid is first initialized using the "GridPartition" operator (left). A line is then inserted inside each face to obtain a subdivided triangular arrangement (middle). The hexagonal partition (right) is obtained by replacing edges by their duals: lines connecting the centroids of adjacent faces.* ***(b)*** *Each edge of an initial grid partition (left) 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).* ***(c)*** *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).* ***(d)*** *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).*

3.5 MERGING OPERATORS

Merging operators take two arrangements as inputs, and return one arrangement (Figure 3.6). They provide a simple way to mix simple arrangements to obtain complex patterns. We propose three different merging operators (Table 3.5):

— *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 3.6(d)).

— *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 3.6(e-h)).

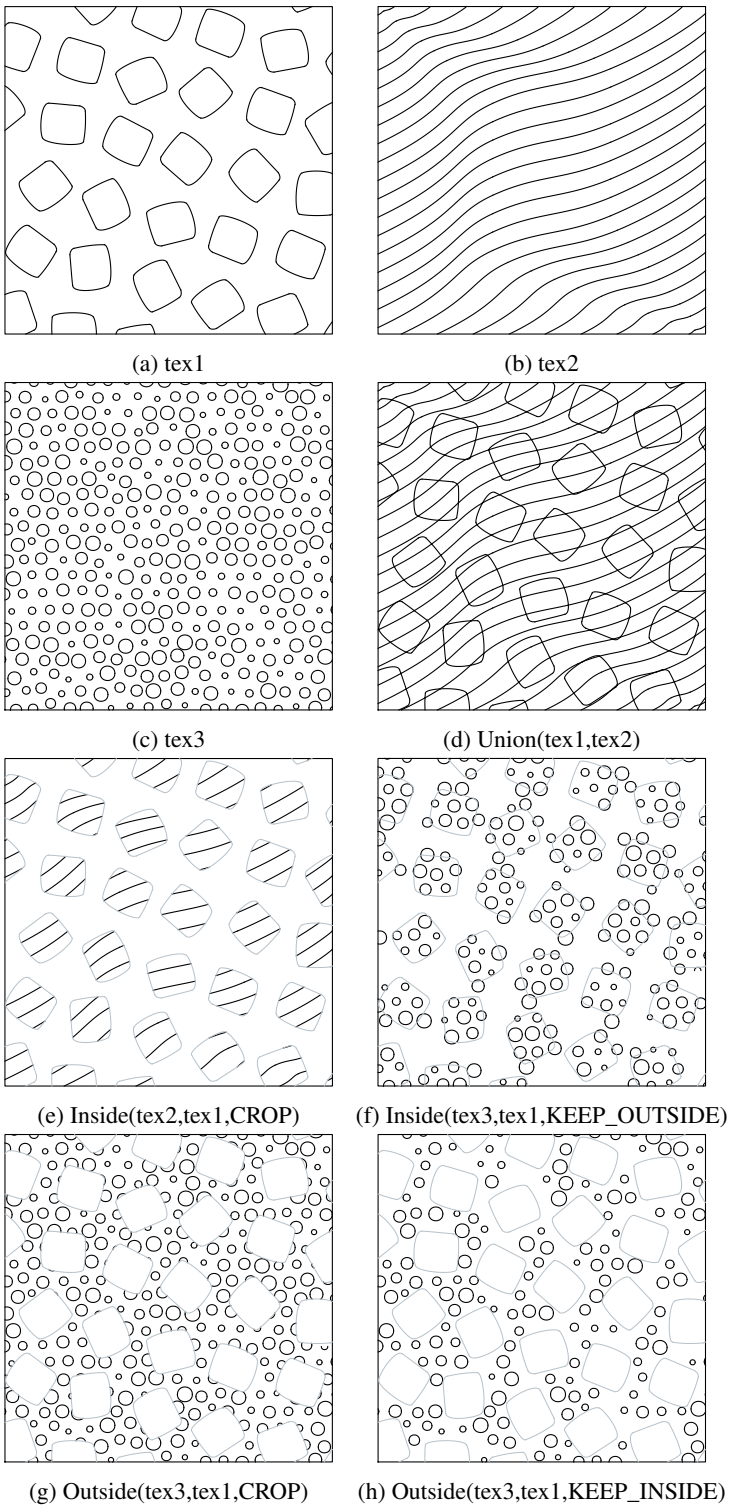


FIGURE 3.6 – ***Merging multiple distributions. (a,b,c)*** *Three simple arrangements obtained via partitioning and mapping operators.* ***(d)*** *The Union operator overlaps its two input arrangements.* ***(e,f)*** *The Inside operator behaves like a mask, keeping only the edges from a first arrangement that fall inside the faces of a second one. The same border management options are proposed as for partitions (see Figure 3.4).* ***(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.*

3.6 Results and Validation

3.6.1 Results

Along the chapter 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 chapter, 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 chapter are included in appendix. Most results were generated in a few seconds (except Figures 1.2(b,d) and 3.9(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 [Coo84]. 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 3.7 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 3.1). 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 3.8 and 3.9 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.

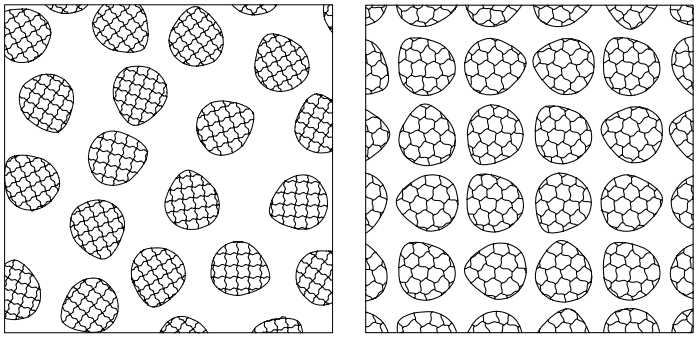
All the scripts producing these examples have less than 60 human-readable lines and they use only the operators given in appendix.

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 3.8 where a variety of arrangements are designed based on simplistic regular partitions. Figure 3.12 also shows that more complex tilings can be easily created such as wallpaper groups tilings.

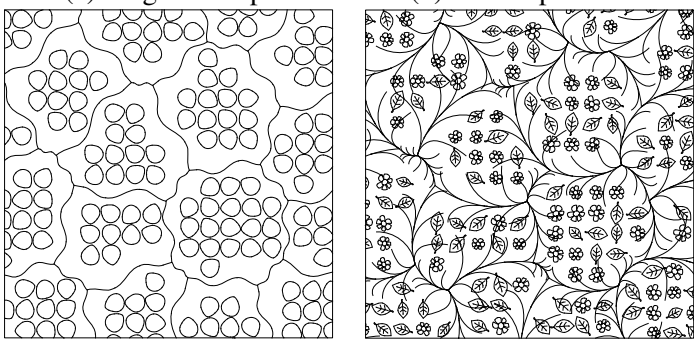
Expressiveness. All the examples shown in this section demonstrate that the arrangement properties we mentionned in Section 1 can be obtained with our set of operators: regular and non-regular arrangements, various elements adjacency relations such as contact or overlap, compositions of several arrangements and clusters of elements. Figure 3.10 shows that our approach overcomes the limitations of existing example-based techniques. In Figure 3.11, we show how labels can be used to create some interactions between combined arrangements allowing to create complex structures in a controllable way.

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.

Finally, Figure 3.13 shows a complete vector drawing, textured with seven scripts using our approach. The seven produced arrangements were then stylized and combined using Inkscape. This demonstrates how such a programmable approach could be used throughout a complete texture design tool.



(a) Original script (b) Switch partitions



(c) Switch arrangements (d) Vary mappings

FIGURE 3.7 – ***Script edition.*** *A design strategy can be to edit iteratively a starting arrangement.* ***(a)*** *Original two-scale arrangement from Figure 3.1.* ***(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.*

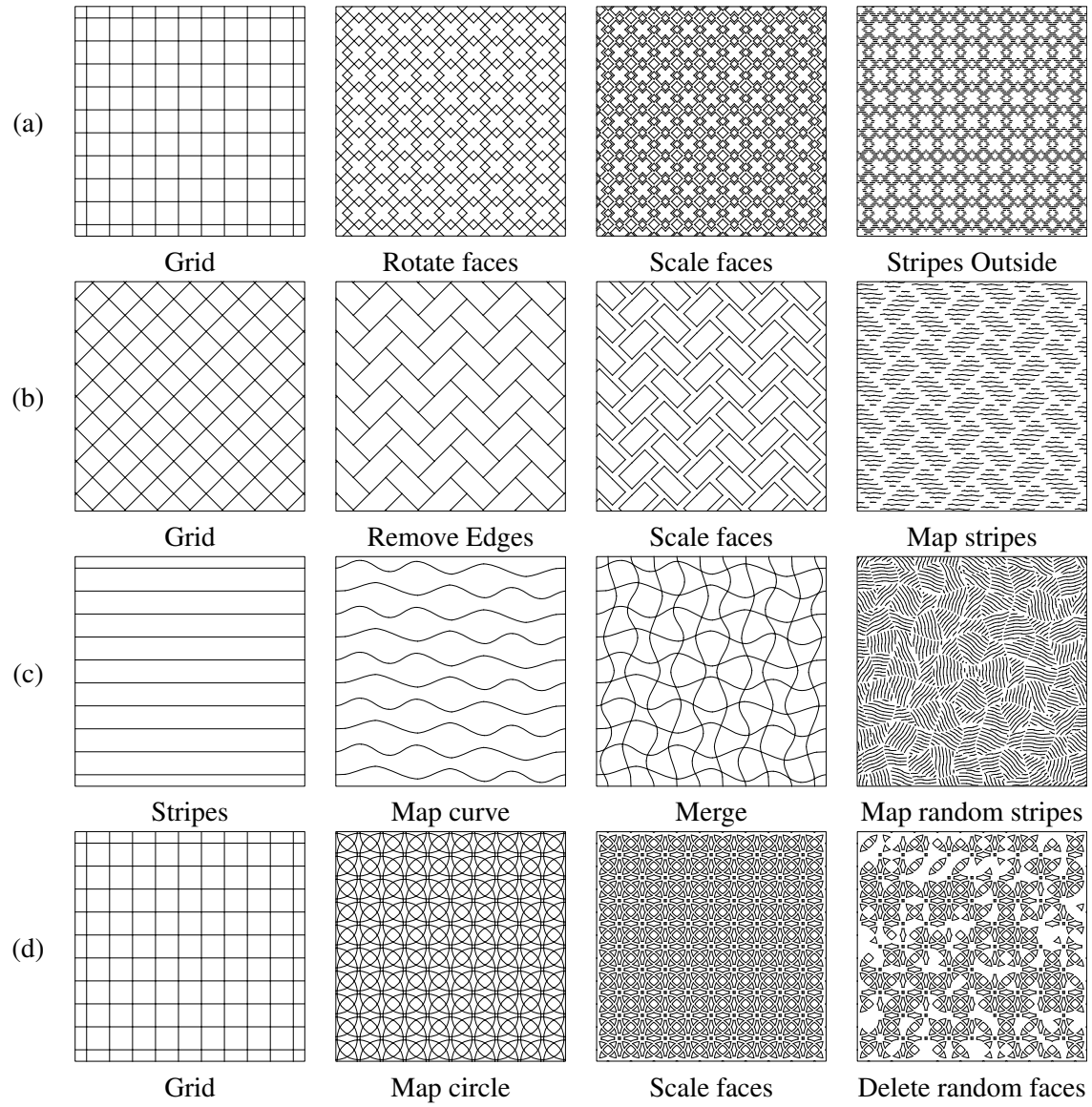


FIGURE 3.8 – ***Creating complex structures starting from regular partitions.*** *Each row shows some iterative design steps, starting from an initial regular partition.* ***(a,b,c)*** *Two-scale examples where the initial partition is refined in different ways, and the resulting regions filled with 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.*

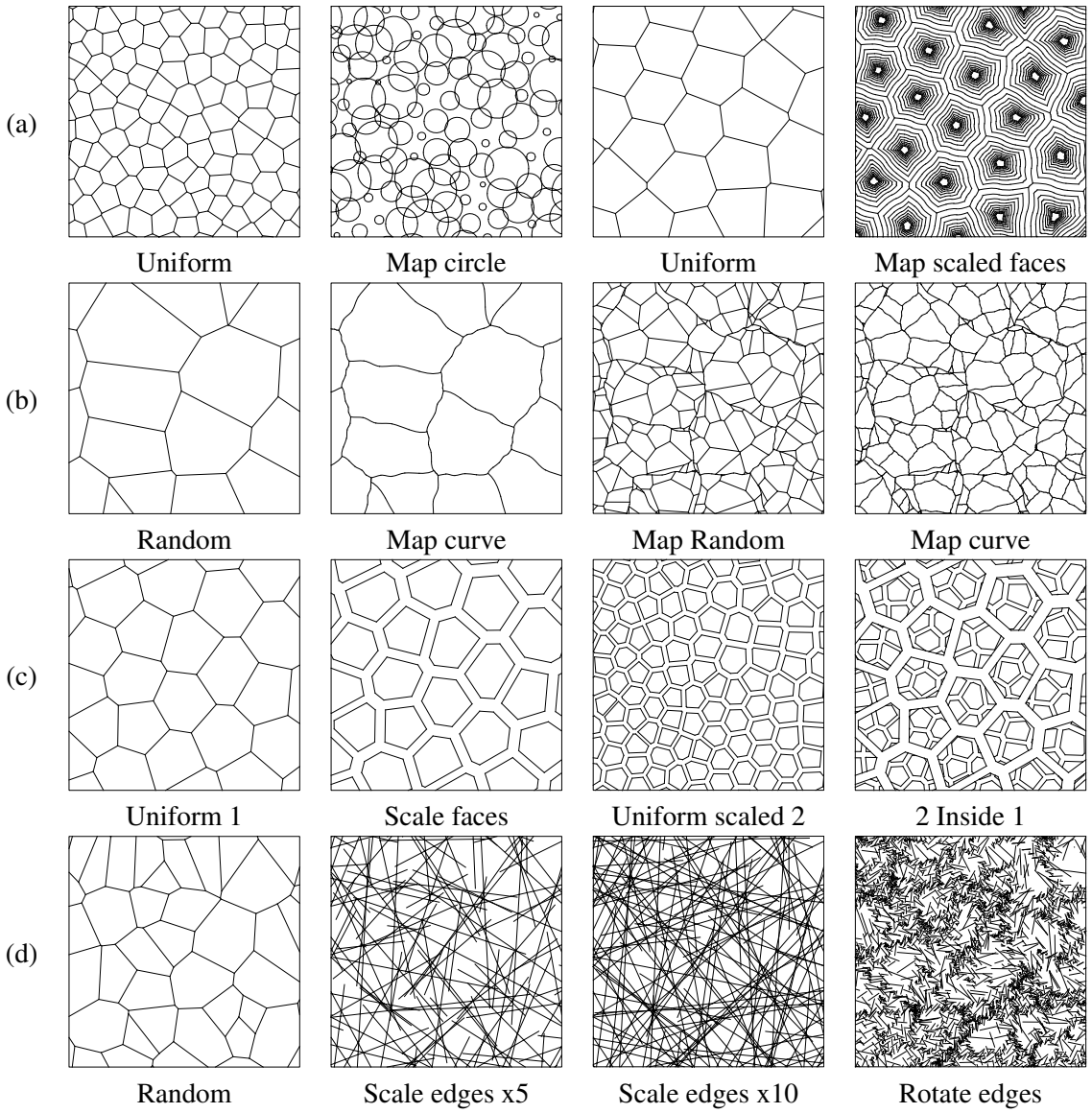


FIGURE 3.9 – ***Creating complex structures starting from non regular partitions.*** ***(a)*** *First row shows two examples starting from a uniform partition, yet with radically different final arrangements.* ***(b)*** *A random partition, after having mapped its edges with a curve, is recursively applied 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.*

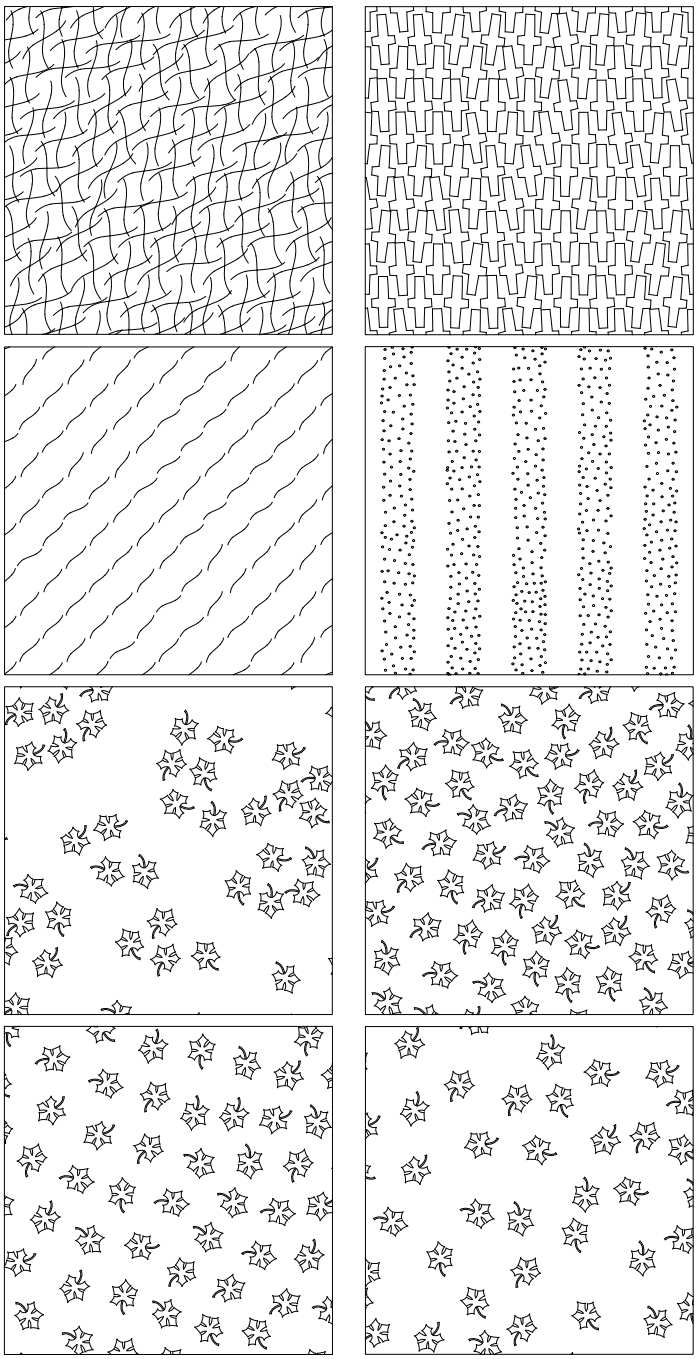


FIGURE 3.10 – ***Comparison with exemplar-based approaches.*** **Top:** *we go beyond by-example methods' limitations from Figure 2.1 by faithfully reproducing the target arrangements with our set of operators.* **Bottom:** *The evaluation protocol developed in [AKA13] showed that even expert designers do not usually agree on what should be the output arrangement based on one exemplar. 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 exemplar.*

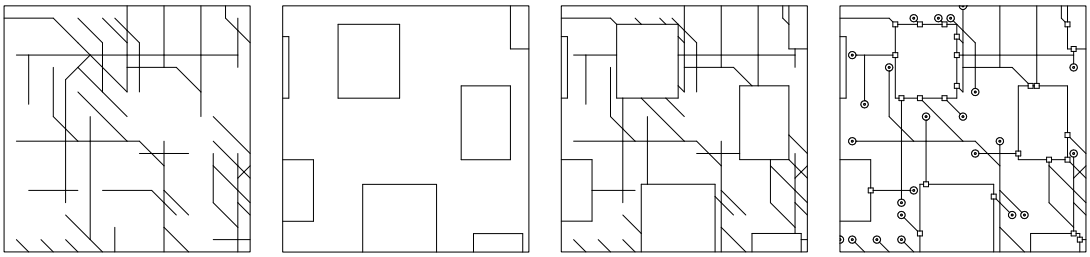


FIGURE 3.11 – ***Interactions between arrangements' scales.*** *This CPU-like arrangement is composed of two levels: chips and junctions connected together.* ***(a)*** *Junctions are created using SVG elements mapped on an initial random partition.* ***(b)*** *Chips are created using the same method and each chip is labelled.* ***(c)*** *These two arrangements are merged using the Outside operator.* ***(d)*** *A mapper uses the labeling information to generate two kinds of connectors depending on the vertices status (end points and chip junctions).*

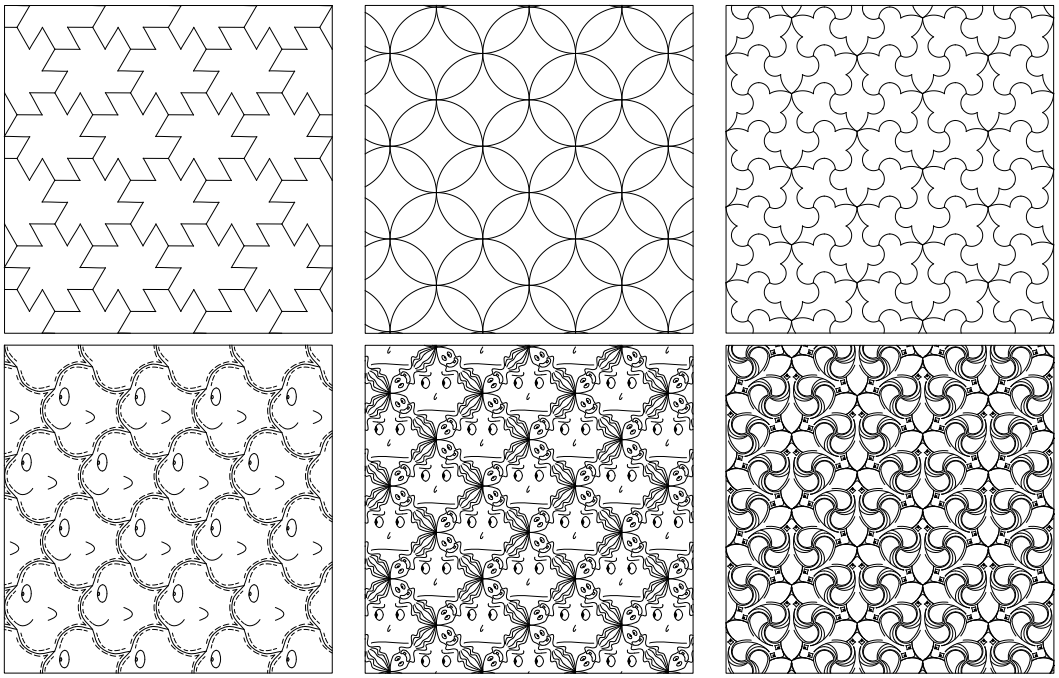


FIGURE 3.12 – ***Creating wallpaper groups tilings of the plane. (Top)*** *A wallpaper group P6 (left) is obtained from an hexagonal tiling (Figure 3.5(a, right)) followed by a mapper replacing each edge with a "Z"-shaped curve. A wallpaper group P4M (center) is obtained by replacing each edge of a tilted grid by a piece of circle. A wallpaper group P31M (right) is obtained by mapping three curves in each face of a triangular tiling (Figure 3.5(a, middle)) .* ***(Bottom)*** *Mapping more complex SVG elements allows Escher-like tilings to be created.*

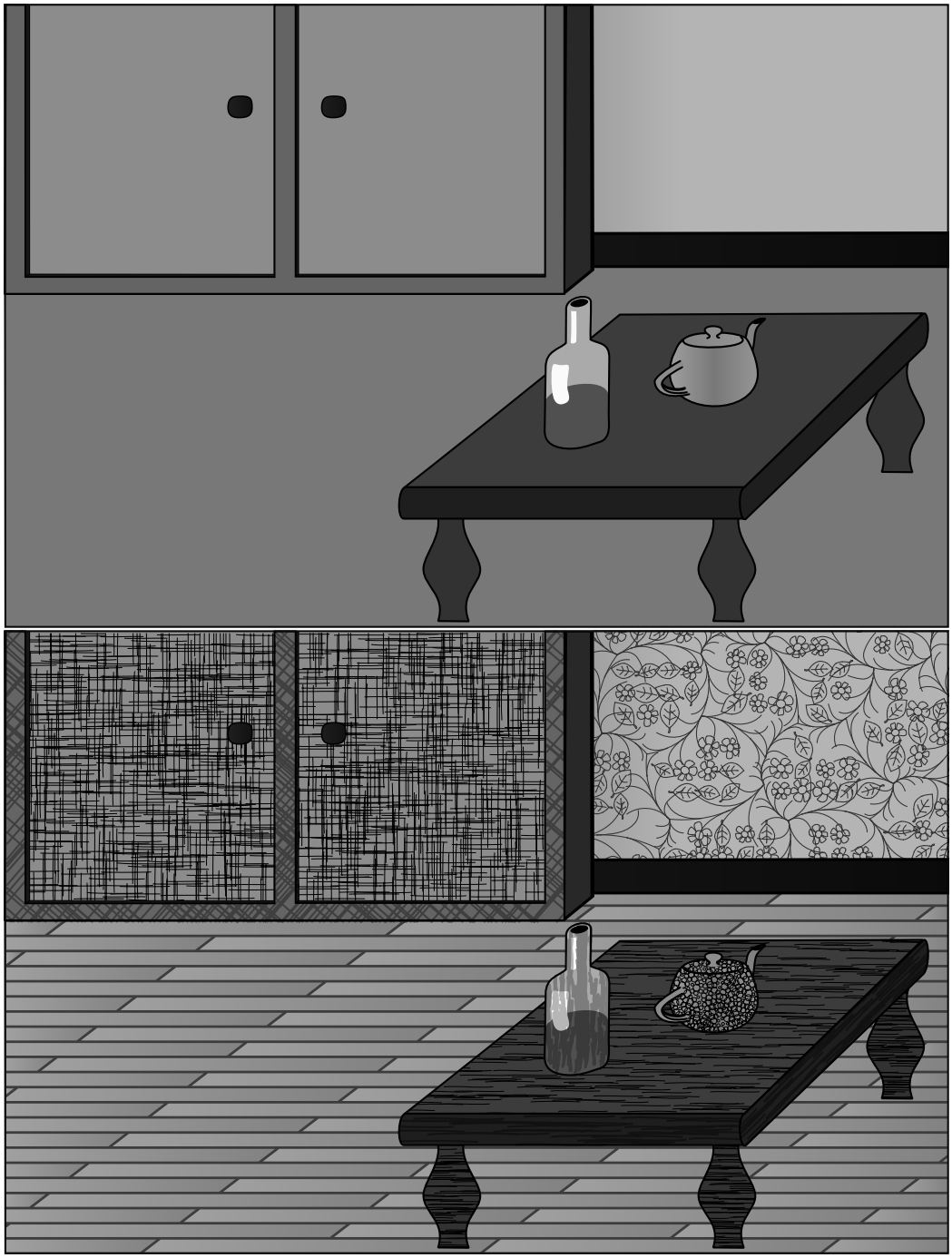


FIGURE 3.13 – *Texture-based illustration.* *(Top)* *A simple SVG drawing.* ***(Bottom)*** *The drawing is textured using seven scripts. The created paths can be grouped and imported in any vector drawing software. They can then be processed as any other SVG element. For example here, a single action was needed for filling all the parquetry slats with a horizontal linear gradient.*

3.6.2 User Study

We evaluated the practical use of our model by asking eight users to produce three arrangements as close as possible to three target examples we gave them. As detailed in the following, these users attended the same supervised tutorial session in order to get them familiar with our tool before working on their own in an unsupervised session. The tutorial, the script files and rendered arrangements are all available as supplemental materials.

Procedure

In order to obtain comparable results, the user-study has been conducted under the following procedure:

— All participants have backgrounds similar to technical directors. They have either a master degree in Computer Science or a title from a digital art school. They all had some experience with scripting before the study.

— **Tutorial (45min):** They all attended a 45 minutes **supervised** tutorial session, where they were introduced to the general principles of our model including partitions, mappers, and combining operators. The user was also provided with a Python script filled with working examples of textures. The user was invited to modify this file during the teaching part so as to get used with our operator syntax.

— **Sandbox (15min):** Users were next asked to produce two target textures, based on some given SVG elements, and some Python code snippets of partitions and mappers. The goal of this brief **supervised** session was to get users a step-further independent. This was their last chance to ask some questions about our tool before the unsupervised session.

— **Practice (3×15min):** This is the unsupervised part of the study. We gave users three manually drawn examples found on online photostocks (First column of Figure 3.14). Each participant was then asked to "use our tool during 15 minutes so as to produce a texture having an appearance as close as possible to the target". Each user was provided with the Python scripts that she/he used during the Sandbox, and two SVG elements: one "horseshoe" and one wavy curve. For each target example, we measured the number of times each user visualized intermediary results and we stored the resulting script and arrangement. We also asked the user to assign himself a mark between 1 and 10 that represents how satisfied she/he is with her/his result.

The study was followed by an oral discussion, based on the same set of questions for each user, in order to get qualitative feedback.

TABLE 3.1 – *User-study measures.* *Mean values / standard deviation of measures made on the eight participants.*

	Puzzles	Cracks	Waves
Satisfaction (out of 10)	9.1 / 1.5	7.8 / 1.8	7.8 / 1.5
Script length (in lines)	22 / 2.4	15.4 / 5.9	26 / 2.9
Number of operators used	2 / 0	4.4 / 1.6	6 / 1.4
Number of executions	3.1 / 1.0	2.4 / 0.5	3.6 / 0.7

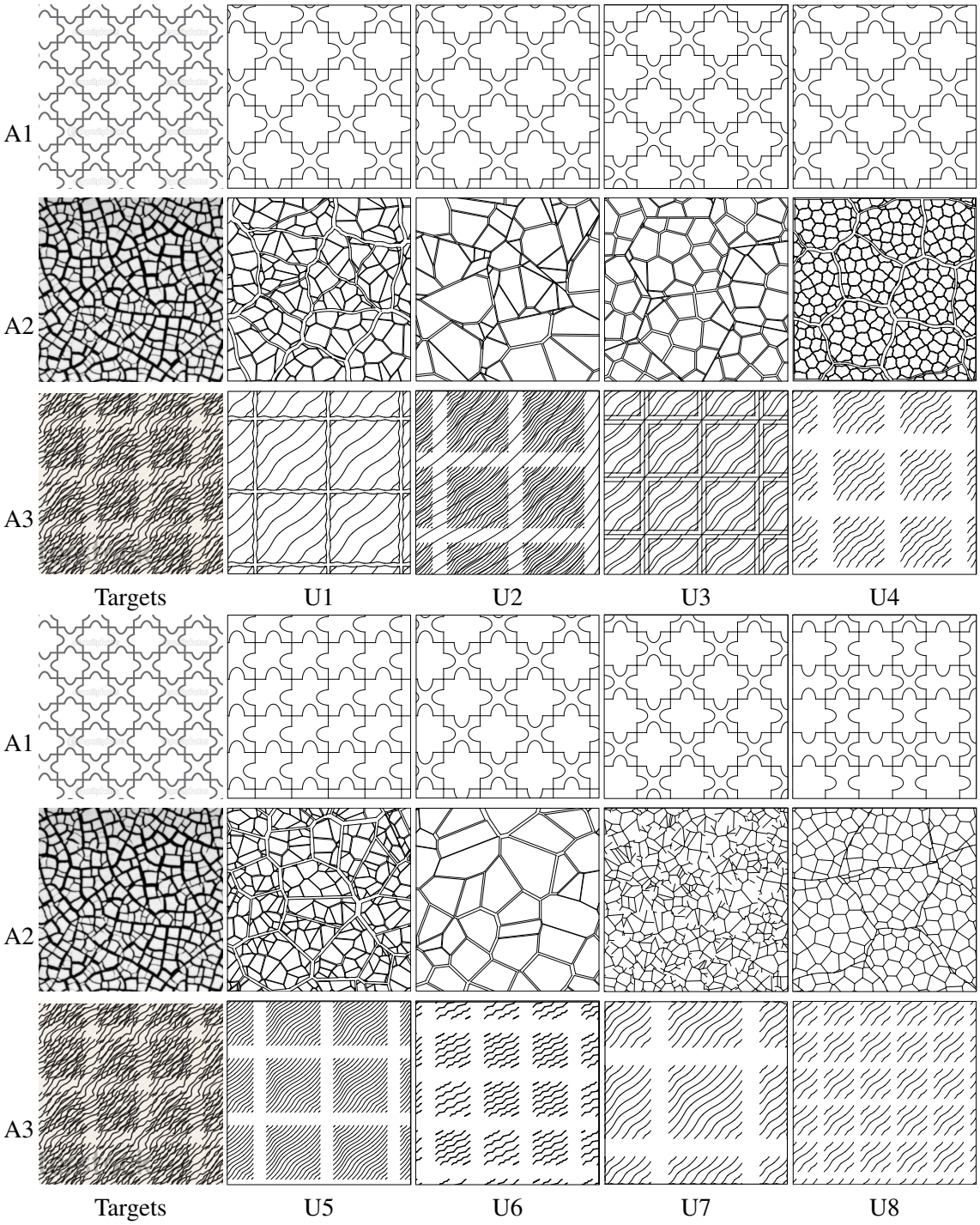


FIGURE 3.14 – ***Results from new users given a target image.*** *After a one-hour tutorial, we asked eight users to produce arrangements having an appearance as close as possible to the target images in the first column. Some of the results have been cropped in order to have comparable scales. Full results are given in the supplemental material.*

Practice Results

The three arrangements produced by the eight users during the practice session are shown in Figure 3.14. Table 3.1 gives the means and standard deviation of the satisfaction, script length, number of operators and number of executions for each example.

The results obtained after 15 minutes vary from being strongly similar to the target example (aside from the stroke grain) to being partly similar. All users except U5 achieved a perfect match with the target A1. Although visually simple, this regular puzzle-like target needs to use some labels to handle the orientation permutation of the stroke. Considering the second arrangement A2, the main variation among users' results comes from the chosen partition at each level, being either uniform (U3, U4, U8) or random, the latter leading to more faithful results. However, all participants except U6 and U7 managed to propose a two-levels arrangement. For the third arrangement A3, all participants also proposed a two-scales result. For this target, the main feature that users (except U2) did not manage to design under 15 minutes are the striped strokes that are traversing all the image.

After one hour of tutorial, the users mastered our proposed model enough to be able to divide input textures into structures that can independently be generated and combined using our operators. They usually took similar paths to generate their results. This is confirmed by the low standard deviations in script lengths and the number of operators used for each target listed in Table 3.1. All participants worked with rather small scripts, mostly between 15 and 30 lines, even for the waves example which embeds different level of structures. On average, they wrote 21 lines of code and composed 4 partitioning/mapping/combining operators in 15 minutes for a given texture. It is very important for a scripting tool to describe complex results in such a compact fashion because the most time-consuming scripting mistakes generally come when the script becomes too long. The users checked out for intermediary results every five minutes on average, which is sound for a scripting system.

These results demonstrate that our tool is simple to learn, intuitive and efficient when applied to practical tasks. All the positive points mentioned above are confirmed by the high satisfaction marks assigned by the users, as well as their feedbacks which are summarized in the next section.

Interview Summary

The last 15 minutes of the study consisted in a guided discussion. The supervisor took some notes for each question (see supplemental material). During the interview, all users declared that the target textures were easy to mentally decompose into our operator set and that it is easy to make these mental decompositions real by scripting using our system. Users said that they always kept tight control over their scripts, except U3 and U5 who mentioned that they once lost the sense of what the script was doing when manipulating multiscale partitions and labels. Half of the users pointed out that the labels were a bit hard to manage (U1, U2, U5 and U7) and that reducing the computation time would improve their experience (all users except U5 and U6). However, they all found that our operators are intuitive. In particular, some of the users mentioned that the model was very convenient because it reminded them other nodal tools (U2, U4 and U8). Two users (U4 and U8) especially liked that our tool encourages iterative design despite the computational time. Three users (U1, U2, U8) pointed a satisfying learning curve and the two latter found that they produced results that were surprisingly complex and aesthetic. All the users were positive about the overall experience of learning and practicing our tool.

3.7 OPERATOR LIST

We give here the list of our operators in respective tables: partition operators (Table 3.2), mapping operators (Table 3.3), mappers' built-in operators (Table 3.4), merging operators (Table 3.5), and other useful functions available anywhere in user scripts (Table 3.6).

TABLE 3.2 – *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 3.3 – *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 3.4 – ***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 3.5 – ***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 3.6 – ***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

CHAPTER

4

TILED PLANAR MAPS FOR INTERACTIVE DESIGN OF REGULAR AND SEMI-REGULAR ARRANGEMENTS

A major feedback of our user study is that execution time is the main pain point for the usability of our tool, in particular when users design arrangements through trials and errors (Section 3.6.2). This motivates the need for an arrangement synthesis method with interactive performances. Arrangements generated interactively also have broader applications: For example, they become potential assets for self-generating digital worlds such as in computer games and simulators. This chapter presents a method that demonstrates interactive performances at synthesizing a commonly used subclass of arrangements: regular and semi-regular arrangements. This method extends the design tool presented in Chapter 3.

Making Chapter 3's method faster is a hard problem. Arrangements represented as planar maps involve large amounts of geometry to store and process. For example a planar map with 10^5 cells usually takes over 100MB in RAM. Intense computational geometry algorithms have to be performed at each design iteration, such as detecting intersections between all the arrangement curves. Fortunately, these computations appear to be redundant when the arrangements exhibit regularities. Our goal here is to leverage these redundancies so as to craft a much faster synthesis method for (semi) regular arrangements. Such arrangements are frequently needed: For example manufactured surfaces and materials fall in this category.

The cases with most computation redundancies are the purely regular arrangements (also called periodic arrangements). Simply storing a regular arrangement can be strongly optimized because most of the geometry could be expressed as translations of a small group of cells. Furthermore when a regular arrangement is transformed into another regular arrangement (for

example with a non-random mapper), we can observe that the mapper is *commutative with translation*. This means that applying the mapper to a small group of cells should be sufficient for knowing the composition of the entire output arrangement.

Our method allows the user to design and visualize an infinite, regular arrangement which is represented behind the scenes by a *Tiled Planar Map* (TPM): A new data structure that contains a small set of cells that are sufficient for representing the entire arrangement. The user designs this infinite arrangement using the operators from Chapter 3. We introduce algorithms to compute TPMs that represent the output infinite arrangements for each operator. These algorithms allow to construct regular arrangements following Chapter 3's design rules, at interactive performances. Additionally we allow to randomize the regular arrangement at the end of the script, which makes it a semi-regular arrangement. The resulting design tool features the following advantages:

Interactive design. All operators execute at interactive performances. Those are independent from the arrangement size.

Parallel, constant-time point location. TPMs' point location is parallel and its computational cost does not depend on the point's spatial location. Consequently the following operations can be done in constant-time and in parallel:

— Finding which arrangement cell contains a given point (x, y).
— Computing the distance between a given point (x, y) and its nearest curve in the arrangement.
— Rendering the arrangement at any pixel (x, y).

Deterministic, parallel conversion to planar maps. A TPM can still be converted to a planar map filling a given region, in linear time regarding the number of duplicated cells. This is much faster than Chapter 3's quadratic synthesis method. Conversion to planar maps is still a deterministic process. Furthermore, it is also *local*: The content of the output planar map inside a small window is independent from the overall shape of the region to fill, as opposed to Chapter 3's method. This allows *parallel* conversion: the region to fill can be cut in any number of parts. Then, each of these parts can be handled in parallel. Let note that the position of each part does not affect the conversion cost. This allows seamless, on-the-fly generation of arrangements for a stream of windows, which is an important feature for interactive applications such as computer games.

Same design process. From the designers' perspective, arrangement creation follows the logics proved efficient in Chapter 3.

In the next sections we will give details about the TPM, how to implement Chapter 3's operators for TPMs, how to design procedural noise for creating semi-regular arrangements and finally how to export TPMs to planar maps. We will validate our approach by showing several arrangements produced side-by-side with the system from Chapter 3 and our new method. We will demonstrate the interactive usability of our new tool regarding the previous one by displaying the performance speed-ups we obtained.

4.1 THE TILED PLANAR MAP

The Tiled Planar Map is a structure that represents an infinite, regular arrangement by storing a small, sufficient set of planar map cells. The rest of the infinite arrangement is expressed as translations of these cells. Formally, a TPM T is a record of the following items:

— Two non-zero, non-colinear vectors $\mathbf{v}_1, \mathbf{v}_2$ which are the TPM's *periods*.
— A point $\mathbf{o}$ which is the TPM's *origin*.
— A planar map p following the definition from Section 3.2.

Additionally, these members are used to define:

— The *central tile* which is the parallelogram

$$t_{0,0} = (\mathbf{o}, \mathbf{o} + \mathbf{v}_1, \mathbf{o} + \mathbf{v}_1 + \mathbf{v}_2, \mathbf{o} + \mathbf{v}_2)$$

— The *tiles* which are the translated parallelograms

$$t_{i,j} = t_{0,0} + i\mathbf{v}_1 + j\mathbf{v}_2$$

Finally, the size of p is constrained by T's periods according to the following invariant: All faces of p are at least partially overlaping with the central tile $t_{0,0}$, and they are always entirely contained in the tiles $t_{i,j}$ with $i, j \in \{-1, 0, 1\}$. An example of TPM is displayed in Figure 4.1.

Each TPM T can be associated with an infinite planar map p^+ whose cells are exactly p's cells translated along multiples of $\mathbf{v}_1$ and $\mathbf{v}_2$. This means the following:

— For any cell $c \in p$ and all integers k_1, k_2, its translation $c + k_1\mathbf{v}_1 + k_2\mathbf{v}_2$ is a cell in p^+.
— Conversely, any cell in p^+ is the translation of a cell $c \in p$ along $k_1\mathbf{v}_1 + k_2\mathbf{v}_2$ for some integers k_1, k_2.

This infinite planar map p^+ corresponds to the actual arrangement designed by the user. The two properties above ensure that the TPM we process represents exactly p^+ at all design steps. Note that the small planar map contained in a TPM is usually larger than its tile so that we store at least one sample of each cell present in p^+ (see Figure 4.1). This involves slight redundancies in the stored geometry, but these redundancies are small enough to be harmless for our method's performances. This solution is also better than cutting p inside the central tile, which would break p's topology and for example add new vertices not desired by the user.

In the following sections we will consider each arrangement as being represented by a TPM.

4.2 OPERATORS ON TILED PLANAR MAPS

Here we describe implementations of Chapter 3's operators. Designers can use them the same way, except that they output TPMs instead of arrangements. However this implies no syntax change since any TPM can also be seen as an arrangement (a function which associates a planar map to each face).

4.2.1 TPM Partitions

As in Chapter 3, designing an arrangement always begins with a partition operator. This time, only the GridPartition operator is available since our method addresses regular structures with two non-colinear periods. This operator can still be parameterized for both grid directions, each one following an angle, a cycle of spacing values $s_1, \ldots, s_n$ and a cycle of labels

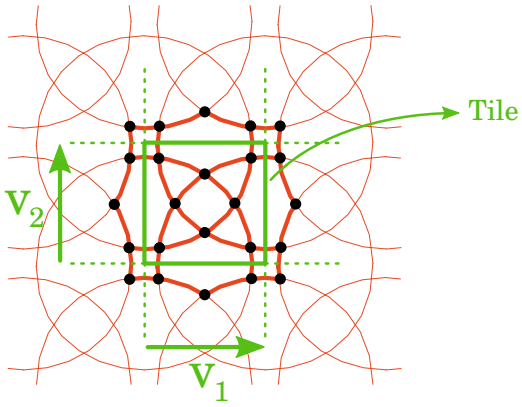


FIGURE 4.1 – ***Contents of a Tiled Planar Map.*** *Each TPM contains a small planar map (black vertices and thick orange edges) as well as two vectors* $\mathbf{v}_1, \mathbf{v}_2$ *that define how to tile this planar map. These vectors define the TPM's central tile (green square). Note that the planar map is usually larger than its tile so that the TPM contains at least one sample of each cell present in the arrangement. The rest of the arrangement (thin orange curves) is not stored and can be deduced from the TPM.*

$l_1, \ldots, l_m$ provided by the user. Given these informations, the output TPM has periods $\mathbf{v}_1$ and $\mathbf{v}_2$ aligned on provided angles. For each direction, we compute the length L of the corresponding period in function of the spacing values $s_1, \ldots, s_n$ and of the number of label values m:

$$L = \frac{LCM(n,m)}{n} \sum_{i=1}^{n} s_i$$

Where $LCM(n,m)$ refers to the Least Common Multiple of n and m. The output TPM's planar map is a grid computed as in Chapter 3's GridPartition operator and populated with the labels provided by the user.

4.2.2 TPM Mappers

Such as in Chapter 3, arrangements can be refined at will using mappers on vertices, edges or faces. These mappers follow the same programming rules as in Section 3.4. They also allow to use the same API, except for random functions which are forbidden in order to keep the arrangement regular. Mappers are applied to the whole arrangement using the corresponding operator (MapToVertices, MapToEdges or MapToFaces). The output arrangement can then be refined again or combined with other arrangements.

Behind the scene, we have to compute a new TPM that represents the output arrangement. For that, let note that a non-random mapper applied to a regular arrangement yields an other regular arrangement with *equal periods*. Note that this is also true in case of mappers using labels: the initial arrangement period already accounts for labels. Using this observation, we design an algorithm (Figure 4.2) independent from the mapper type (vertex, edge or face) which runs as follows. For an input mapper m and an input TPM T with planar map p and central tile $t_{0,0}$ representing the infinite arrangement p^+:

1. Apply m to all cells of p. We consider that m returns nothing for cells of the wrong type (for example, a vertex mapper returns nothing for faces) and we call the result $m(p)$.

2. Compute the set A of tiles $t_{i,j}$ overlapped by $m(p)$.
3. Compute the set B of couples (i', j') such that $m(p + i'\mathbf{v}_1 + j'\mathbf{v}_2)$ overlaps $t_{0,0}$. This set is obtained by applying a central symetry on A:
$$B = \{(i', j') = (-i, -j) | (i, j) \in A\}$$
4. Compute the *Neighborhood Influence Zone* NIZ which is the set of couples (i'', j'') such that $m(p + i''\mathbf{v}_1 + j''\mathbf{v}_2)$ overlaps one of the $t_{i,j}$ for $i, j \in \{-1, 0, 1\}$. This set is obtained by computing the convolution between $\{-1, 0, 1\}^2$ and B.
5. Compute the planar map
$$p' = \bigcup_{(i'',j'')\in NIZ} m(p) + i''\mathbf{v}_1 + j''\mathbf{v}_2$$
6. Compute the set of cells of p' that overlap $t_{0,0}$. These constitute the planar map of the output TPM $m(T)$. This output TPM has periods equal to T's.

Implementation of adjacency operators. In our method, mappers can use the same API as in Section 3.4 except for random functions. This API follows the same implementation. The only exception is adjacency operators: in the central planar map of a TPM, some cells are incident to only a subset of their real neighbors in the infinite arrangement: For example in Figure 4.1, border vertices are incident to only two edges whereas they have four real neighbours in the infinite arrangement. In order to reproduce the behavior of Chapter 3's adjacency operators, we need to find these additional neighbors and return them as well. This will return cells that are actually not stored anywhere (being outside the central planar map). We use the property that such cells can be expressed as translations of cells in the central planar map. We represent them as *shifted cells*, a simple structure that contains a reference to an actual cell c in the central planar map plus two counters k_1, k_2 that record which translation of c is designated. When applying an adjacency operator on c, we look for neighbors of c's translations along period multiples in the central planar map. Formally, if the TPM's periods are $\mathbf{v}_1$ and $\mathbf{v}_2$, the adjacency operator N on a shifted cell (c, i, j) is computed as following:
$$N(c, i, j) = \{(c', i + i', j + j') | c' \sim (c + i'\mathbf{v}_1 + j'\mathbf{v}_2)\}$$
where $\sim$ refers to the incidence relationship between two planar map cells. This operator can be extended to the TPM's actual cells as $N(c) = N(c, 0, 0)$. If any, copies are removed from this set using the equality relationship between shifted cells:
$$(c_1, i_1, j_1) = (c_2, i_2, j_2) \Leftrightarrow c_1 + i_1\mathbf{v}_1 + j_1\mathbf{v}_2 = c_2 + i_2\mathbf{v}_1 + j_2\mathbf{v}_2$$

4.2.3 TPM Combiners

Such as in Chapter 3, infinite arrangements can be combined using the operators Merge, Inside and Outside. These operators can still be parameterized with border management options. Note that the output arrangement is regular if and only if the input arrangements have aligned periods with a common multiple. Otherwise, their combination exhibits unique local configurations that "shift" slowly over space, without ever occuring more than one time [1]. Conversely, if

1. When periods are not aligned, one would need to multiply them with non-rational numbers in order to find a common multiple. This makes it impossible to find an integer number of period repetitions that ends up producing two occurences of the same configuration. Some exceptions work in theory but cannot be computed in practice: For example an axis-aligned grid with period 1 merged along a $\frac{\pi}{4}$-rotated grid with period $\sqrt{2}$ produces a regular arrangement, but it would imply to store $\sqrt{2}$ numerically and then to compute intersections between non-rational segments, which is not practical. A much simpler way to obtain this result is to write a mapper that cuts in two parts each face of the axis-aligned grid.

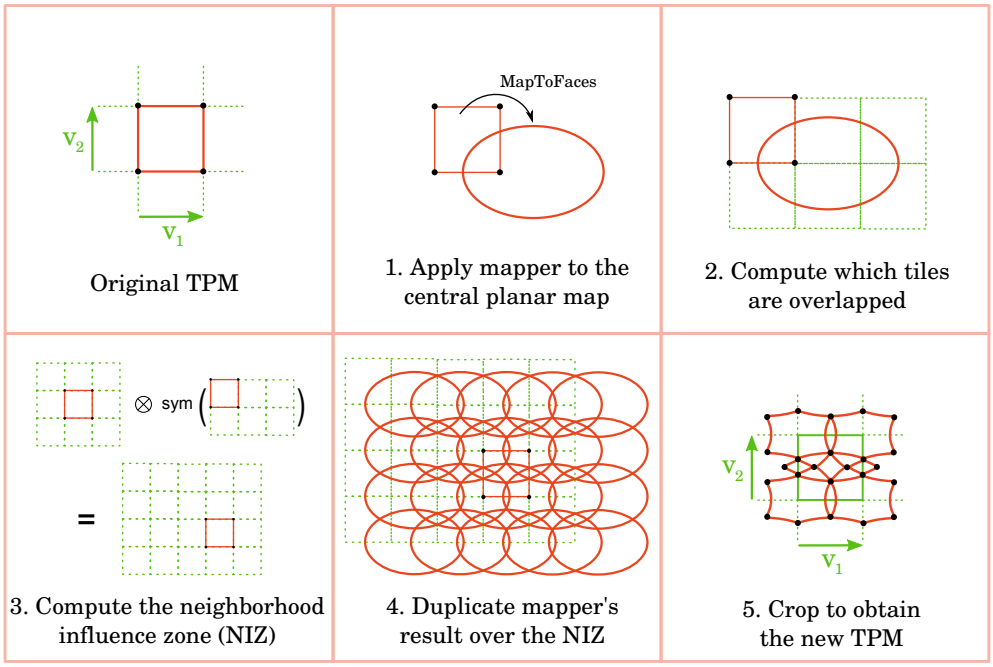


FIGURE 4.2 – ***Applying a mapper to a TPM.*** *When the user asks for applying a mapper to the infinite arrangement being designed, we have to compute the TPM respresenting the output infinite arrangement. We already know the output central tile since input and output arrangements have equal periods. Thus we just have to find all cells overlapping this central tile. From the TPM T representing the input arrangement (top left), we* **1.** *apply the user-defined mapper to all cells of T's planar map - in this example the mapper associates a shifted ellipse to each face;* **2.** *compute which tiles are overlapped by the mapper result. This result will allow us to determine conversely which cells of the output arrangement will overlap a given tile $t_{i,j}$;* **3.** *compute the Neighborhood Influence Zone, which is the set of tiles from where mapper results will overlap the central tile's 8-neighbors;* **4.** *group these mapper results in a single planar map, and* **5.** *keep cells from this planar map only when they overlap at least partially the central tile.*

input arrangements have aligned periods with common multiples then the output arrangement is periodic 's period is equal to the least of these common multiples. In practice, we check that the two input arrangements periods $(\mathbf{v}_1, \mathbf{v}_2)$ and $(\mathbf{v'}_1, \mathbf{v'}_2)$ respect the following property:

$$\exists k_1, k'_1, k_2, k'_2 \in \mathbb{N} | k_1\mathbf{v}_1 = k'_1\mathbf{v'}_1, k_2\mathbf{v}_2 = k'_2\mathbf{v'}_2$$

In this case we will use the Least Common Multiple notation: $\mathbf{LCM}(\mathbf{v}_1, \mathbf{v'}_1) = k_1\mathbf{v}_1$ and $\mathbf{LCM}(\mathbf{v}_2, \mathbf{v'}_2) = k_2\mathbf{v}_2$. These vectors are the output arrangement periods. From there, it is simple to find a TPM that represents the output arrangement. For input TPMs having planar maps p_1, p_2 and a masking operator O, our algorithm (Figure 4.3) runs the following steps:

1. Compute $\mathbf{v''}_1 = \mathbf{LCM}(\mathbf{v}_1, \mathbf{v'}_1)$ and $\mathbf{v''}_2 = \mathbf{LCM}(\mathbf{v}_2, \mathbf{v'}_2)$. These define the output central tile.
2. Compute the planar map p'_1 duplicated from p_1 over a 3×3 neighborhood of the output central tile. The same way, compute the planar map p'_2 duplicated from p_2 over a 3×3 neighborhood of the output central tile.
3. Compute the desired masking operator: $M = O(p'_1, p'_2)$

4. Finally, keep only cells of M that overlap at least partially the new central tile defined by $\mathbf{v''}_1$ and $\mathbf{v''}_2$.

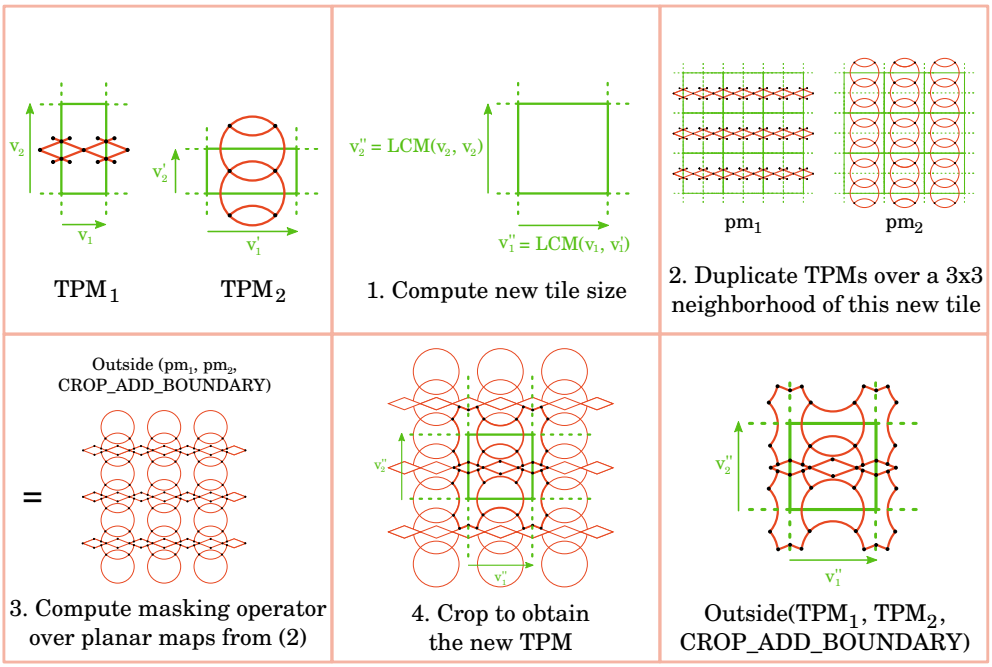


FIGURE 4.3 – ***Applying a combiner to TPMs.*** *When the user asks for applying a combiner to a couple of infinite arrangements, we have to compute the TPM respresenting the output infinite arrangement. This operation needs input arrangements to have periods with common multiples. If it is the case, from the two input planar maps (top left) we:* **1.** *compute the output central tile which dimensions are the least of these common multiples;* **2.** *duplicate the input planar maps along a 3x3 neighborhood of this new central tile;* **3.** *compute the desired masking operator on the bigger planar maps we obtained in 2. and* **4.** *keep the obtained cells only when they overlap the new central tile.*

4.2.4 Ghost Mapping

In our design tool, we allow to use procedural noise that changes purely regular arrangements into semi-regular arrangements. This noise is introduced with the same principle as mappers (Section 3.4). When users are done designing their desired regular structure, we allow them to write an ultimate mapper where random functions are allowed. When this mapper is applied to the arrangement, we do not compute a new planar map such as in Section 3.4's mapping process. Instead we keep a pointer to the mapper function. This pointer will be used to call the mapper on-the-fly at rendering/export time. Since this mapper does not have any influence on the rest of the script, its result not being usable, we call this process *ghost mapping*: an opportunity to add randomness and details without having to compute the involved costly intersection calculus.

Ghost mapping can be used for the same broad range as Section 3.4's mappers. This procedural noise thus opens possibilities beyond simple element jittering. One can add or remove new elements, add detail to the geometry, generate new shapes, etc. See Figure 4.5 for examples.

4.2.5 Conversion to Planar Maps

In the previous sections, we have seen that every planar map storing a regular arrangement can be represented more efficiently as a TPM. Conversely, every TPM can be converted back to a planar map covering a given region. Our algorithm is rather simple: The TPM's planar map is duplicated over all the region to fill (see Figure 4.4 for details about this duplication process). Then the result is cropped regarding a border management option given by the user. If any ghost mapper has been attached to the TPM, then it is called during the duplication process.

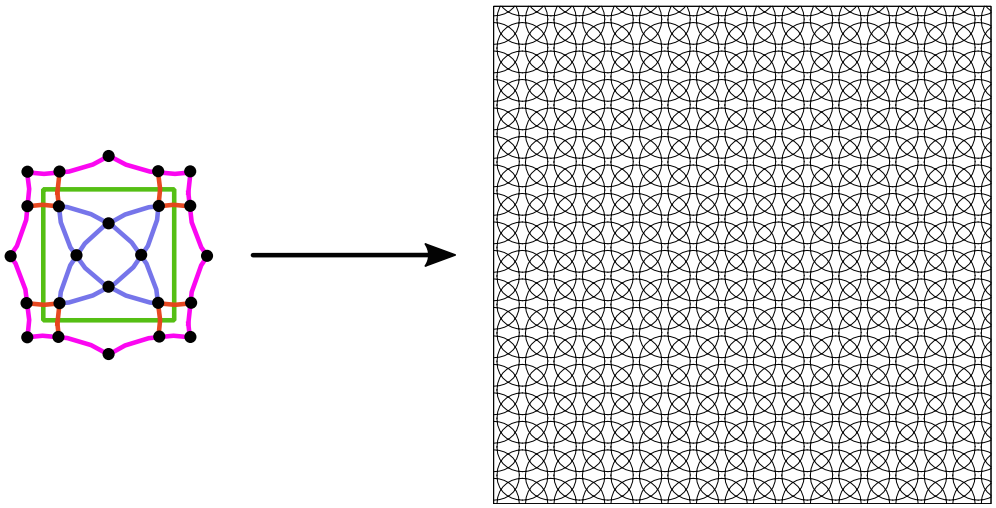


FIGURE 4.4 – ***Converting TPMs into planar maps.*** *Any TPM (left) can be converted back to a regular planar map. Each curve is processed in function of its position regarding the tile: Inner curves (blue) are duplicated every period* $(\mathbf{v}_1, \mathbf{v}_2)$*, border curves (orange) are duplicated every other period and outer curves (pink) are not duplicated since they are already inner curves of adjacent tiles. This process has linear complexity regarding the number of duplicated curves. In this example, the conversion took less than one second.*

Conversion from TPM to planar maps validates the full equivalence between this new synthesis method and Chapter 3's engine. Therefore our new method can be used as a drop-in replacement for all regular and semi-regular arrangements: For these targets it provides the same design features than Chapter 3's, plus interactive performances.

4.3 RESULTS AND VALIDATION

This section features performance measurements for comparison with Chapter 3 as well as several example outputs.

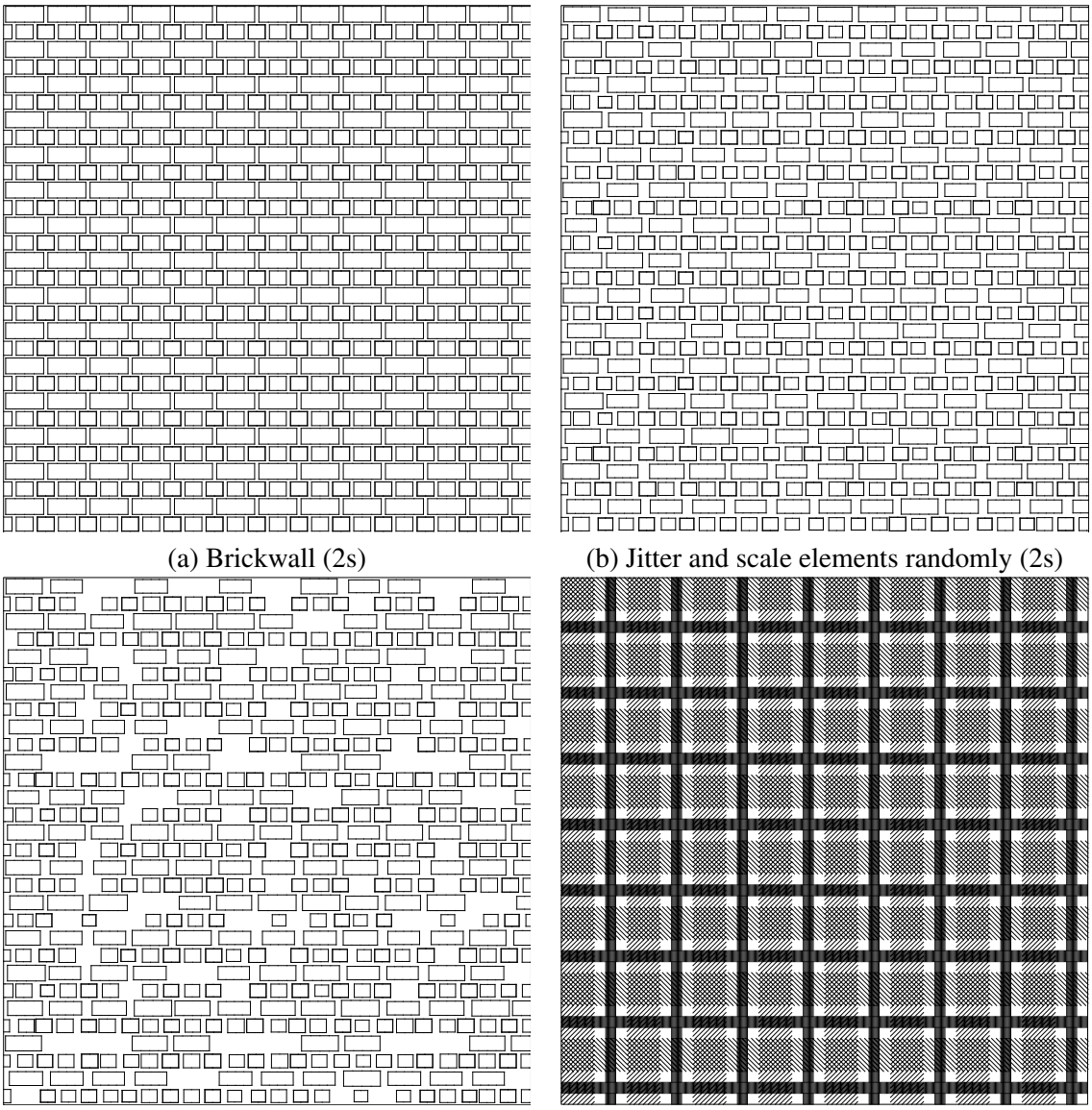
The performances obtained with our new approach are displayed in Table 4.1. These measures validate our bill of specifications. First, they validate that operators on TPMs actually execute in constant time regarding the size of the designed arrangement. Second, they show that converting TPMs to planar maps has indeed linear cost regarding the number of output cells. Third, they show that computing operators on TPMs and then converting the outputs is always much faster than using Chapter 3's engine. This last result validates that our new synthesis method can be used as a drop-in replacement of the previous engine for all regular and

semi-regular arrangements. Let note that the table says “aborted” for the previous engine’s result at 9000 synthesized cells because the process was killed before completion due to memory swap. This means that there is no hope to obtain the 9000 cells in reasonable time for this example.

TABLE 4.1 – *Performance comparisons* *for the brickwall texture (Figure 1.2(c)), generated for several window sizes. All time measurements are given in seconds for geometry operations computed on a single Intel(R) Core(TM) i7-3610QM CPU with frequency 2.30GHz and 4GB RAM.*

Amount synthesized	Operator computation	Conversion to planar map	Previous method
350 cells	1.92s	1.57s	6.5s
1400 cells	2.09s	4.16s	69.9s
2100 cells	1.99s	5.68s	162.39s
4200 cells	1.92s	9.45s	700.16s
9000 cells	1.99s	17.25s	aborted

A few results are shown in Figure 4.5. These results demonstrate how the same regular and semi-regular arrangements can be designed as with Chapter 3’s tool. Furthermore, we hope that this new approach to arrangement synthesis will allow users to go further and create even more complex designs now that performance is less of a limitation. In the case of semi-regular arrangements, tuning amplitude of random functions such as jittering usually takes several iterations before getting the desired look. The performance speed-up is even more of an advantage in these cases.



(a) Brickwall (2s)

(b) Jitter and scale elements randomly (2s)

(c) Delete elements randomly (2s)

(d) Fabric pattern (4s)

FIGURE 4.5 – ***Results produced with our new synthesis engine.*** *All these results demonstrate a speed-up of at least two orders of magnitude regarding Chapter 3's engine.* **(a)** *Reproduction of Figure 1.2(c). 3200 cells are synhesized.* **(b)** *Same example with ghost mapping. The mapper jitters and scales randomly the arrangement faces from (a).* **(c)** *Another example of ghost mapper that also deletes elements randomly in addition to jittering and scaling the others.* **(d)** *Reproduction of Figure 1.2(h).*

CHAPTER

5

APPLICATION TO CARTOGRAPHY

5.1 INTRODUCTION

Cartography is one of the oldest craftsmanship in human history, and also one of the most critical vectors of development for human culture, communication, trading and leisure. Maps, as real world representations, have been completely drawn by hand for centuries. Map production benefited from a significant speed-up with the invention of printing, but at least one exemplary of each map still had to be drawn exhaustively by hand. Drawing by hand a topographic map, representing the terrain based on its natural features such as relief or hydrography, and its artificial entities if any (buildings, roads), was a very labor-intensive task, due to the quantity of information to be represented [Duf66]. For example, the 1986 map of Austrian Alps took over 11000 hours of work [Jen04]. The past fifty years of scientific and technical advances in geographical information sciences, i.e. data acquisition, image processing, geographic database and cartography, brought topographic map series production to became digital and most of all automated. Institutional National Mapping Agencies, but also, for instance, Google Maps [1] and OpenStreetMap [2], took then part to the development of geographical information services in providing geographical data and cartography on-demand tools to users.

Nevertheless, a main difficulty related to the automation of the map series production process remains, for instance at the French National Mapping Agency: this process still relies on some non-automatic steps which could prevent from being able to update the map series. Certain parts of the terrain, i.e. rocks, scree, etc. in mountain areas, are currently represented using scans of old manual drawings. It implies now some problems while the French National Mapping Agency has to update geographical data and related maps: the melting and thus retreat of glaciers in mountain areas cannot be updated as long as we do not have automatic tools to generate those parts of the terrain.

1. `http://maps.google.com`
2. `http://www.openstreetmap.org`

Yet topographic maps are of primary importance for many users, in particular in mountainous areas: Terrain perception is the first information needed there for determining where are the hazardous zones, choose a path towards a given destination, etc. based on the understanding of the main morphological structures of the terrain (height, slope, roughness, ridges, valleys, etc.). Cartographers need a more automated and controllable production pipeline for topographic maps, which is not trivially feasible using state-of-the-art automated cartographic techniques: for example, the classic cartographic representation of terrain height are contour lines (lines of constant height). This representation is now generated automatically, but it is not practical when the terrain slopes are too steep. In these cases, the height step between two consecutive contour lines would be too big for providing relevant information. If one tries to display a bigger amount of contour lines, they would become too compressed and would make the map illegible. The case where terrain slopes are too big is very common in mountainous areas, especially in rocky zones where the slopes reach values around 70-80%. Moreover, in order to enhance the terrain perception, at some adapted scales, it would be more interesting to manage more expressive renderings of the relief. Therefore, these rocky zones, ubiquitous in mountains, require a dedicated depiction.

Traditionally, rocky areas are represented using hatchings whose density and orientation convey information about the orientation of rock surfaces (see Figure 5.1). Various hatching styles have been designed over centuries [Ali73], but the last map produced with this method was finished in 1956 [Gui05]. Since then, this artistic knowledge has not been transmitted to earlier generations of cartographers due to the extensive training periods needed. This overlong training time is yet another motivation for automating the production of mountain maps. In practice, it is challenging to generate hatching textures that convey the rock areas morphology as in hand-drawn maps. In particular some hatches correspond to actual terrain features such as ridges or valleys (also called *thalwegs*), whereas other hatches do not have this meaning and rather carry information through global properties of their distribution: density, orientation, randomness, etc. Finally, some properties of the hatches like length and randomness are influenced by the artistic style of the map. These should be made easy-to-tweak for applications such as map-on-demand.

These challenges call for a texture design tool that grants a lot of control over the hatching arrangement. Our programmable approach is a good candidate in such a context. In this chapter we describe how we use this approach to generate hatching textures that depict rocky areas in mountain maps. This depiction is expected to **1.** convey correctly the mountain morphology, and **2.** be easy to adapt for each viewer's particular needs. This part of my work has been done in collaboration with the French National Mapping Agency (IGN-France). In order to find a relevant design, we draw inspiration from maps they produced manually beforehand.

In practice, producing automatically hatching textures that represents mountain rocky areas implies to make these textures *spatially-varying*. We can observe in the manually-produced hatching shown in Figure 5.1(d,e) that the hatching indeed takes various density and orientation values over the map. Our texture design method described in Chapter 3 outputs purely stationary results, which does not permit to generate spatially-varying textures as is. In this chapter, we describe how we extend our method so as to handle spatial variations and we describe our hatching design for representing rocky areas. Figure 5.1 shows how the hatching textures are traditionally composited with shading layers to make the final map. Artistic shading is a research problem by itself, we will address hatching only in this chapter. Let

note that our method could also be applied to other cartography themes that involve textures: representation of vegetation, sand, sunken sea rocks, etc. We envision these usages as future, more thorough applications of our method to cartography.

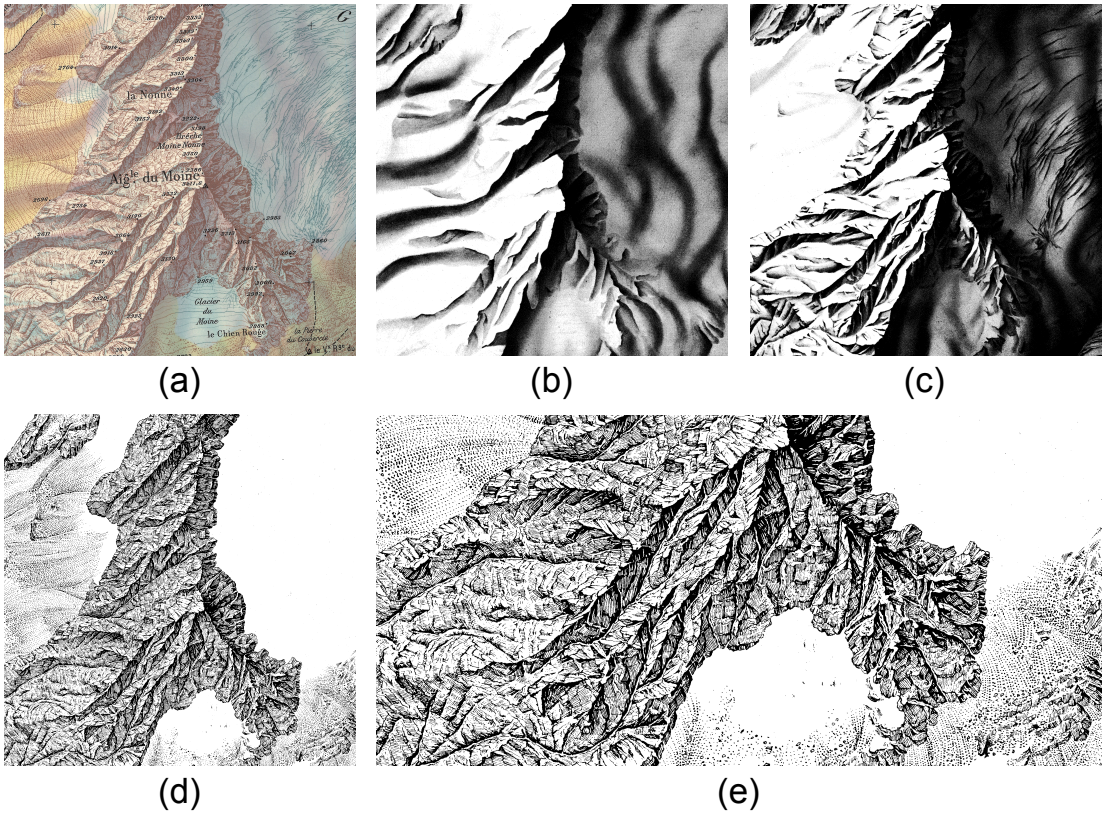


FIGURE 5.1 – ***Layers in a mountain map.*** *A complete map (a) for high mountain areas is traditionally composed of a number of black and white layers (b to e) that are composed using various color inks. Among these, several layers of artistic shading (b& c) and a layer of hatching for rocks (d) are major actors in the global look of the map. (e) shows a close-up view of the rock hatching layer. The goal of our project is to produce automatically rock hatching layers inspired by such existing maps. This map has ben produced between 1940 and 1956 by the French National Mapping Agency [Ali73].*

5.2 RELATED WORK

5.2.1 Artistic Mountains Maps

A lot of mountain maps used today still rely on manual designs. A good review of hand-drawn mountain maps is available in [Ali73]. Drawing these maps is a very labor-intensive task [Duf66, Jen04]. Another drawback of hand drawing is that the result is too dependent of each cartographer's personal style. For example, the French National Mapping Agency course for rock drawing gives a large freedom to the interpretation of each cartographer [Duf66]. Although this might not be a problem for other arts, it caused production issues when scaling up to the mapping of an entire country: this task generally involved many cartographers, which

made it difficult to maintain an uniform look between areas handled by each cartographer with his/her personal style.

These drawbacks motivated the development of computer-aided techniques for depicting the terrain, in particular rocky areas. Let note that hand-drawn maps still possess the highest rendering qualities of all existing techniques. Even though hand drawing is no longer a viable solution for worldwide mapping, these maps serve as a strong inspiration tool for our project.

5.2.2 Computational Depiction of Rocky Areas

The French National Mapping Agency has been looking for the automation of mountain maps production for years. An early example of a fully automated depiction of mountains has been realized and published in 2008 [LG08]. In this work, all traditional layers used in cartographic representations of mountains are automatically produced and assembled. However, the rock drawing layer was considered by the French National Mapping Agency as not conveying well enough the shape of rocky areas: it was carrying ambiguities about the rock shapes, passable zones and hazardous areas. This work continued with a better depiction of areas containing screes using an example-based texture synthesis method [HL11].

More recently, advances were made in the direction of imitating the Swiss rock drawing style [JGG*14, GH15] and apply it to creating a new topographic map for the mount Everest [LH15]. Here again, several layers of rendering are generated and assembled. The rock hatching layer is produced by drawing streamlines of constant height, with varying density [JL97]. This method produces results with a look close to the Swiss drawing style, but is based on a single predefined arrangement of hatches. This makes this method not flexible enough for applications such as map-on-demand. Furthermore, it was still not expressive enough for revealing all the relevant relief features such as structure lines, steep slope areas, inflexion points and saddles.

On the Computer Graphics side, works such as [BST09] have been achieved towards representing mountains in an artistic way. These works make good inspiration material even though they target aesthetic results rather than precise terrain morphology conveyance.

5.2.3 Spatially-Varying Element Textures in Computer Graphics

More generally, several methods presented in Chapter 2 aim at automatically producing spatially-varying arrangements. These methods are either not expressive or controllable enough for addressing the rendering of rocky mountain areas. Some of them are based upon a user-drawn exemplar [MWT11, IMIM08, KNBH12, AdPWS10, HLT*09, LGH13]. This does not fits our needs for precise control over the output design and for adaptation to varying user needs. Furthermore, most rock hatching arrangements exhibit precarious topological properties that these methods fail to reproduce faithfully, for example contact between hatches. The others propose to contract and stretch a predefined element arrangement in order to match precisely user-given density and orientation fields [SHS02, LWSF10]. Due to being constrained to such a predefined layout, these methods would be hard to use and adapt to the user demands. Finally, examplar-based nor layout-based methods provide enough control for generating hatches that

correspond exactly to ridges or valleys simultaneously with hatches that carry information through global properties of their distribution such as density, orientation or randomness.

5.3 Programmable Design of Spatially-Varying Arrangements

Hatching textures that represent rocky areas are *spatially-varying* textures: their density and orientation vary over the map. Such textures are beyond our definition of stationary outputs (see Chapter 3). Therefore we need to add an extension to our design method so as to add user-provided *control fields* that influence the texture over space. These control fields can come from any kind of data: bitmaps, close-form or implicit functions, etc.

Our approach needs to be flexible and allows each arrangement to be applied to any control field. Therefore, the system should allow to design arrangements and control fields separately. In practice, control fields are designed by the National Mapping Agency researchers which formalize the design rules used empirically by cartographers in the 40s. This research involves numerous iterations over the control fields. Designing the texture arrangement and its control fields simultaneously is not a situation specific to cartography: texture designers often manipulate both the texture and its control fields until they obtain the desired result [Dem01]. This is much easier to do when painting over some area of the control field modifies only the corresponding area in the output arrangement. This is often the case in texture design through shaders applied to bitmap data, but it is not the case for most element arrangement algorithms mentionned in Section 5.2.3 because these methods rely on global optimisations that run over the entire texture each time the control field is edited. With these methods, a small localized modification of the control field can yield changes in the entire arrangement. In our case, we want to guarantee *local control*: a small localized modification of the control field must yield only local changes in the output arrangement.

We achieve local control by using control fields inside mappers only, with special programming rules. These rules guarantee that there exists a window outside which modifications of the control field will not influence the mapper's result. This way we let the user free to influence many properties of the texture with control fields. The only exception is density, which can only be piecewise-constant in this model: In order to follow a density field, the user must start with a constant-density partition. Then he can use a mapper to make subscale textures with various densities in each face of this partition. This is not a problem for our application because the rock hatching designs used by map makers seem to have piecewise-constant densities. Therefore we let as further work the issue of partition algorithms that both follow density fields and guarantee local control.

5.3.1 Control Fields

In our approach, we define a control field as a function that associates a scalar or vector value to any point of the plane:In practice, control fields can be crafted from any kind of data (bitmap, close-form or implicit functions, etc) as long as they are associated with a function that follows the definition above. For example, a file containing bitmap data can be associated with an interpolation method.

It must be easy to apply any control field to any designed arrangement. Therefore, the arrangements must be designed separately from the definition of any particular control field. For that we define a *controlled arrangement* as a higher-order function:

$$f_1, f_2, \ldots, f_n \to A$$

where f_i refers to a control function and A refers to an arrangement - a function that associates a new planar map to any face. In this model, the programmer is able to design controlled arrangements without knowing which control field will be applied to them. Afterwards, any set of control fields can be fed as being $f_1, f_2, \ldots, f_n$. Let note that these definitions require no change in our implemented API since higher-order functions are handled natively in Python.

5.3.2 Controled (Higher-Order) Mappers

Similarly to controlled arrangements, we define a *controlled mapper* as being a higher-order function:

$$f_1, f_2, \ldots, f_n \to m$$

where f_i refers to a control function and m refers to a mapper. This way, programmers can design controlled mappers independently from any particular control field.

There are two programming rules for controlled mappers. First, they are the only place in the script where control fields can be used. Second, access the values of $f_1, f_2, \ldots, f_n$ only within a bounded range around the cell being processed by m. A good way to ensure that is to use only points of the plane that are inside faces accessed by m: there always exist a global upper bound to the arrangement cells' size, and m can access to neighbor cells only inside a bounded window (Section 3.4).

5.4 CARTOGRAPHIC DATA

In order to generate hatching textures that convey the rock areas morphology, we need to synthesize hatches that correspond to one-dimensional terrain features (ridges and valleys) plus hatches that carry surface information through global properties of their distribution (density, orientation and randomness). The National Mapping Agency provides the following cartographic data to represent this information:

— A *partition* of the terrain in faces to be textured separately. Face borders are tagged as being ridges, valleys or non-feature lines. Each face is expected to receive a hatching texture with roughly constant density and orientation.
— *Hatching orientation*: A face-wise constant control field which represents the overall orientation desired for each face's hatching.
— *Hatching density*: A face-wise constant control field representing the overall density desired for each face's hatching.
— *Terrain elevation*: A scalar control field which gives the measured elevation of the mountain in meters at any point.
— *Terrain slope*: A scalar control field which gives the slope of the mountain derived from the elevation at any point (in percent).

```
def test_orientfield():
    size = 1000
    # Controlled Mapper
    ellipse = ImportSVG("data/ellipse_long.svg")
    def face_to_ellipse(orient_field):
        def out_mapper(face):
            c = Centroid(face)
            return Rotate(MatchPoint(ellipse,
                BBoxCenter(ellipse), c), orient_field(
                c))
        return out_mapper
    # Controlled arrangement
    def controlled_ellipses(orient_field):
        props = IrregularProperties(500 / 1000000)
        part = UniformPartition(props, KEEP_OUTSIDE)
        tex = MapToFaces(face_to_ellipse(orient_field
            ), part)
        return tex
    # Orientation field
    center = Point(size/2, size/2)
    orient = lambda p: atan2((center - p).y(), (
        center - p).x())
    # Export
    ExportSVG(controlled_ellipses(orient), size)
```

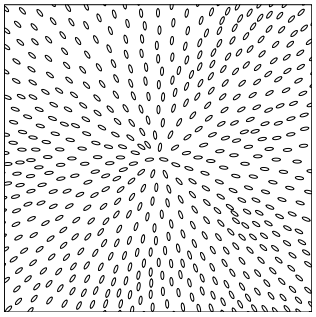


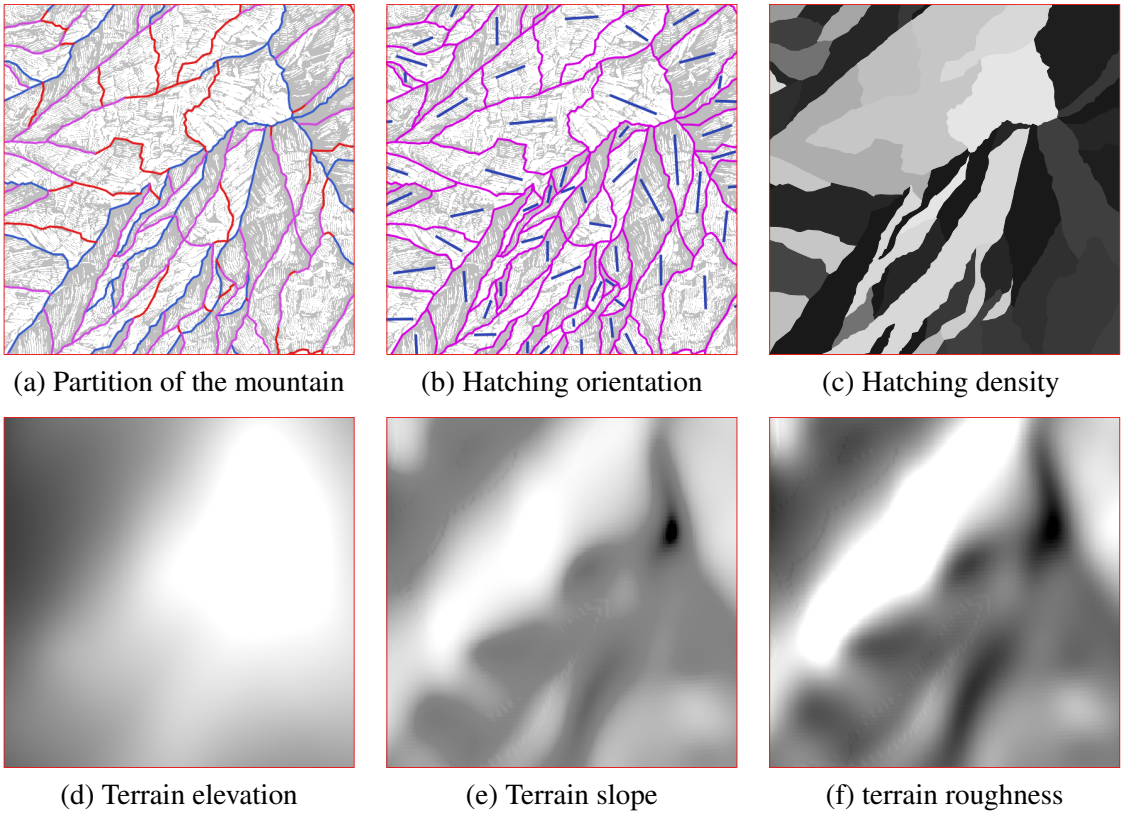
FIGURE 5.2 – *An example of script controlled with a control field. Left:* *A script based on an imported SVG element and a controlled mapper that orients this element along the input control field. This control field is defined as an analytic vector field that points towards the center of the image.* ***Right:*** *The SVG output of the example script.*

— *Terrain roughness*: A scalar control field which gives the roughness of the area derived from the slope at any point (normalized between 0 and 1).

Terrain elevation is measured using Light Detection And Ranging sensors placed on a plane flying over the area. Slope and roughness are computed from this data using classic differential operators. Getting a clean partition automatically is yet an open research topic because it involves to compute ridges and valleys of the height field with correct topology (in particular umbilics). For this reason ridges and valleys are extracted manually from the existing 1:10000 map of the Mont-Blanc which is also used as an inspiration for our design. Similarly, finding the hatching orientation and density that best convey the terrain morphology to human perception is another open research topic at the National Mapping Agency. In order to get the best possible results with our method, hatching orientation and density were extracted manually as well from the 1:10000 map. This information is expected to be automatically produced in the final pipeline.

5.5 DESIGN ITERATIONS

A number of iterations were done jointly with the National Mapping Agency before getting to our final design. In this section we describe and discuss our main design decisions over time. Our results after each decision are shown in Figure 5.4. These iterations demonstrate how our method was able to adapt to our design choices in terms of expressivity and controlability: It was always possible to put in practice our design intents and apply the needed changes.



(a) Partition of the mountain (b) Hatching orientation (c) Hatching density

(d) Terrain elevation (e) Terrain slope (f) terrain roughness

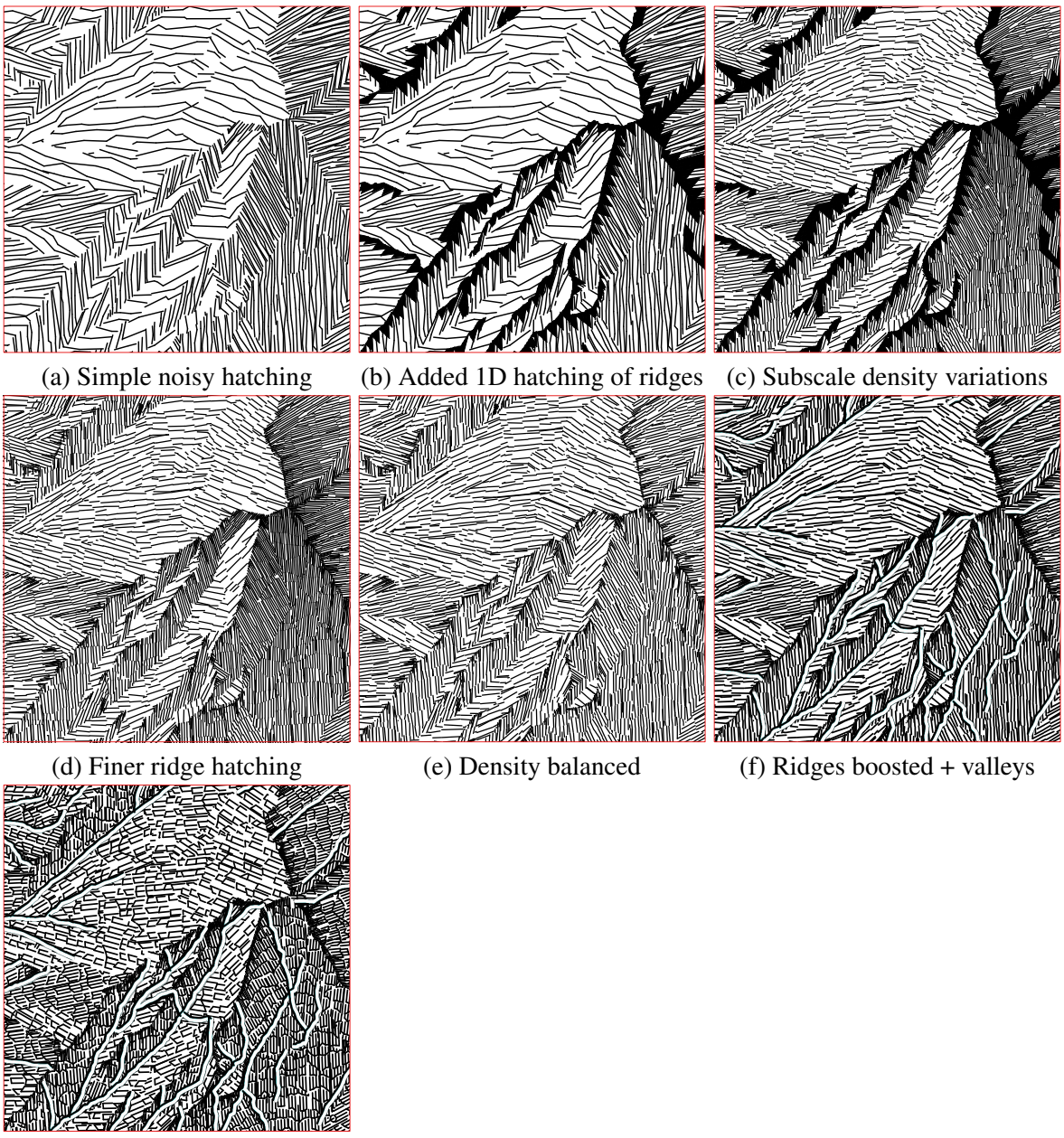
FIGURE 5.3 – ***Data used for rendering rocky areas.*** *This data was produced at the National Mapping Agency by extracting features from the 1:10000 Mont-Blanc map (shown with transparency in (a) and (b)).* **(a)** *A partition of the terrain into faces to fill independently with hatchings. Blue lines denote ridges, purple lines denote thalwegs (valleys) and red lines complete the partition. The rest of the data makes up the control fields used in our design:* **(b)** *In each face of the partition (purple), a blue line indicates the hatching orientation for this face.* **(c)** *Mean hatching density desired for each face, from very dense (black) to very sparse (white).* **(d)** *Terrain elevation captured from Light Detection And Ranging sensors, from 2200m (black) to 3400m (white).* **(e)** *Terrain slope derived from elevation, from 60% (black) to 90% (white).* **(f)** *Terrain roughness normalized between 0 and 1. The area represented here is the Aiguille du Moine in the French Alpes.*

We started with a hatching obtained by keeping one axis of a jittered grid. This hatching has constant density and orientation (except for the noise) in each face of the mountain partition. These density and orientation take their values in the control fields at the centroid of each partition face. Our second design choice was to add a thicker hatching texture falling from ridges in order to enhance the contrast between dark and lit faces such as recommended in [Duf66]. The next step was again to enhance slightly the contrast between dark and lit faces when the elevation is higher. For that we needed to make the hatching density vary inside each mountain face. We achieved this by making a first grid partition, jittering it and then fill each grid face with the first hatching pattern.

We obtained more subtle results by refining the hatchings on ridges: First, we shrinked the hatching shapes and we distributed them more sparsely. Then, we filled these shapes with a

subscale hatching distribution rather than just rendering them as uniform black areas. We also balanced the density of the main hatching pattern in order to avoid visual clutter near the top of the mountain. Still following [Duf66], we enhanced the ridge hatching on higher grounds and we added a rendering of valleys using QuantumGIS (QGIS), a geographical information system. Finally, we added hatchings in a second direction so as to render the terrain roughness by breaking the perceived hatching orientations.

The final design we obtained was considered a good starting point for representing rocky areas and convey their important features: ridges, valleys, saddles and slope directions. This result is planned to be improved again, judged by a panel of cartographers and finally included in a more automatic pipeline for producing topographics maps.



(a) Simple noisy hatching (b) Added 1D hatching of ridges (c) Subscale density variations

(d) Finer ridge hatching (e) Density balanced (f) Ridges boosted + valleys

(g) Added roughness effect

FIGURE 5.4 – ***Main design iterations.*** *Our first trial was a hatching obtained by keeping one axis of a jittered grid (a). Then we added a distribution of triangles and kept those touching ridges so as to enhance the contrast between dark and lit faces (b). The next step was to authorize density variations within a face (c) by partitioning the face and then applying the hatching texture. We chose to refine the ridge hatching shapes and to distribute them more sparsely (d), then to balance density in order to avoid visual clutter near the needle (e). We enhanced the ridge hatching on higher grounds and we added a rendering of valleys in post-processing (f), and we finally switched to bidirectional hatching so as to render the terrain roughness (g).*

CHAPTER

6

DISCUSSION

In this chapter we present our vision about our current research's limitations and the steps forward that we think impactful for further work in the area of arrangement design.

6.1 LIMITATIONS

Continuous density variations. The dotted stripes arrangement in Figure 3.10 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, even with our extension in Chapter 5. The only possible way to do something close to it would be to 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.

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 distribution containing the biggest amount possible of non-overlapping elements (see Figure 6.1).

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

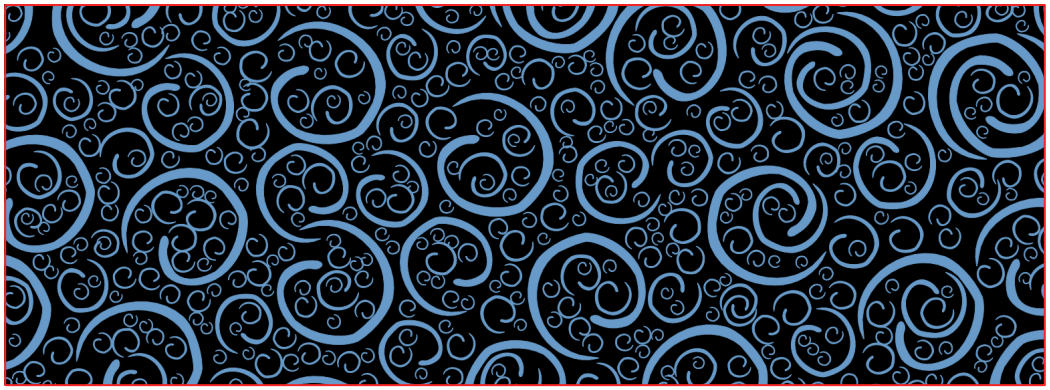


FIGURE 6.1 – *Arrangement defined through implicit control.* *This texture is based upon an arrangement defined as being a "maximum amount of non-overlapping elements". In practice, this arrangement is computed using dart throwing. This is a typical example of arrangement that would be difficult to obtain with our purely explicit approach.*

Planar maps. Since our internal representation is either a planar map or a Tiled Planar Map, we inherit all the limitations of these models. 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 [DRVDP14].

Performances. In Chapter 4 we have presented an extension that increases synthesis performances a lot for (semi) regular texture arrangements. This extension could be optimized further in order to create a no-latency interactive application. Fore example, operators could be computed on Tiled Planar Maps with minimal periods and labels could increase the period only at the moment their are used in a mapper. More importantly, irregular arrangements still need expensive computations to synthesize with our system. This also could be optimized a lot: For example, planar maps are a overly heavy representation for uniform distributions without overlap. Figure 6.2 shows such a distribution generated fastly using an optimized landmark point location which detects and remove collisions between objects, but which does not store them.

6.2 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 [AW90] 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

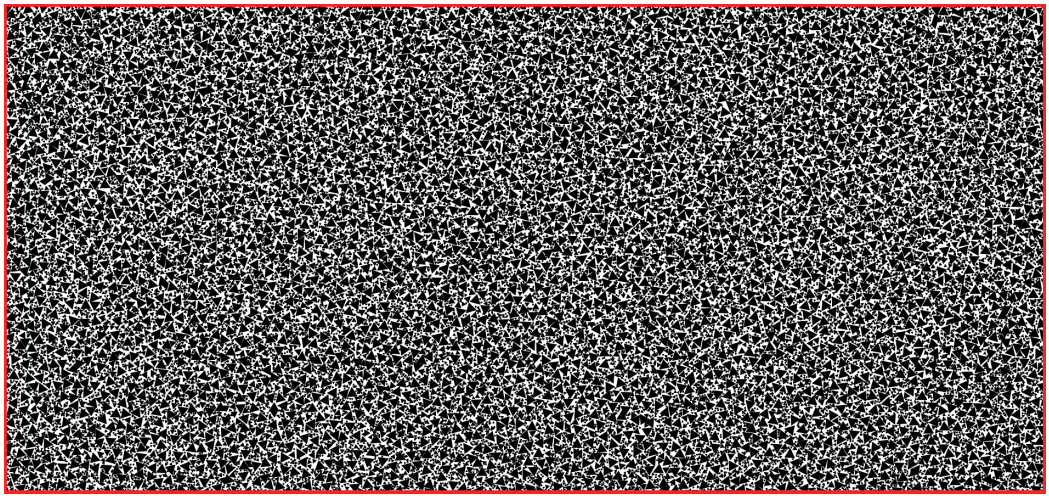


FIGURE 6.2 – *High-performance uniform distribution.* *This arrangement contains 32754 triangles and was generated in eight seconds using an optimized landmark point location.*

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 [GLLD12]. A full inverse *programmable approach* is probably too difficult since it would boil down to going 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.

6.3 TOWARDS A COMPLETE PROGRAMMABLE ILLUSTRATION PIPELINE

Our programmable approach addresses the problem of spatially *arranging* 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 [BA06]. 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 [MM12].

Continuous density variations. In addition to our extension that handles control fields in Chapter 5, it would be impactful to add partition operators that follow continuous density fields, especially for artists designing large backgrounds filled with a single texture (which is commonly done in comic books for example). One idea we experienced for UniformPartition and RandomPartition is to compute Voronoi diagrams from biased point distributions generated using dart throwing algorithms. An example of result we obtained so far is shown in Figure 6.3.

However, there is no guarantee that such an algorithm respects the property of local control needed for faster design iterations.

FIGURE 6.3 – *Texture-based background with continuous density variations.* *The arrangement in background cannot be designed with our current model. It was generated by computing Voronoi diagrams whose seeds are dart-thrown using a user-painted density function. This kind of result could be a target design for future systems.*

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 stylization of line-drawn 3D models [GTDS10, EWHS08]. This method would be a good candidate to extend our approach to stylization. Figure 6.4 gives a glimpse of the freedom let to the user when stylizing a geometric arrangement.

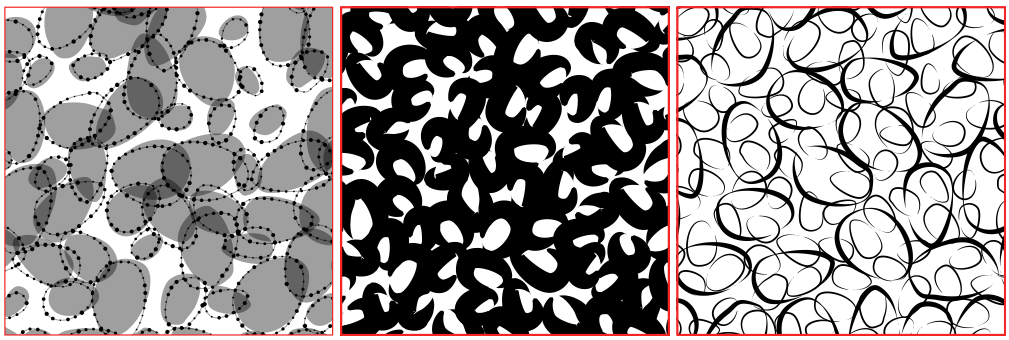


FIGURE 6.4 – *Examples of arrangement stylization.* *Three examples of stylization applied on geometric arrangements.*

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 [OBW*08] by adding color points mappers.

CHAPTER

7

CONCLUSION

In this thesis we presented a new way to design 2D arrangements of geometric elements, which is a major concern in the task of texture creation. We developped ideas and arguments about a *programming* description of arrangements, which we believe is a reliable way to carry the designer's intent. We also demonstrated how this approach is predictable and extensible.

Our experimentation tool focuses on planar map arrangements because they give maximum information to the programmer about the arrangement's geometry and topology, which is a powerful way to demonstrate a broad spectrum of possible results. However it also formalizes a canonical way to *think* arrangement design from the programmer's side: **1.** Decompose the arrangement into a number of scales and layers; **2.** Decide wether each arrangement's broad-scale organization is regular or not; **3.** Decide how the elements are arranged locally and which geometrical/topological details they exhibit. We believe this way of thinking could apply to designing geometric arrangements represented differently: triangulations, point clouds, meshes, etc.

Along this manuscript, we presented tools in which a user describes a texture by writting a script. However we envision that, for most users, this model will eventually serve as an internal representation only: Most user choices could actually be made through a graphical interface, which would be a more natural way to interact with the designed arrangement even for programming-friendly users. Partitions can be manually sketched and their parameters (angles, density) can be recognized automatically. A lot of mappers can be drawn manually over several example cells, then their source code should be generated automatically by the tool. Arrangements can be dragged-and-dropped from various layers so as to be merged. Such a tool potentially represents several PhDs worth of research in inverse procedural modelling, user interface and human-computer interaction. Yet, I like to think of this project as a step towards a helpful design method, which would capture reliably users' intents while exhibiting the same "magic" as examplar-based texturing tools.

APPENDIX

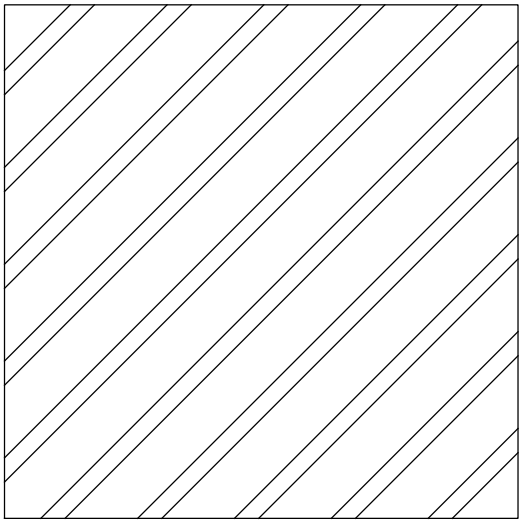
A

EXAMPLE PYTHON SCRIPTS

In this appendix we show the scripts used to synthesize all the arrangements in Chapter 3.

Stripes <1 sec

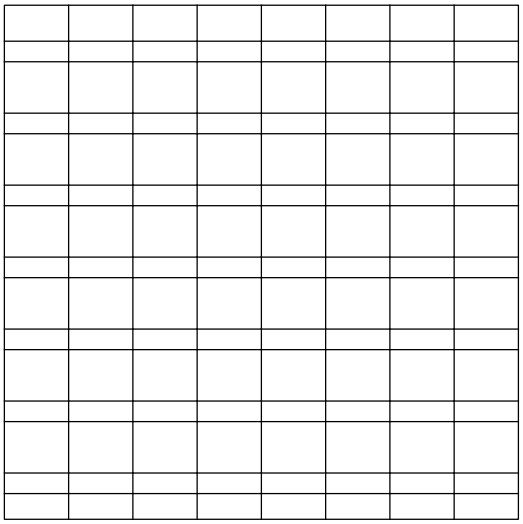
```
def stripes():
    # Texture size
    size  = 2000

    # Stripes angle=pi/4, widths=200 and 67
    lines = StripesProperties(pi/4,size/10,
        size/30)

    # Partition
    tex   = StripesPartition(lines)

    # Export result
    ExportSVG(tex,size)
```



Grid

<1 sec

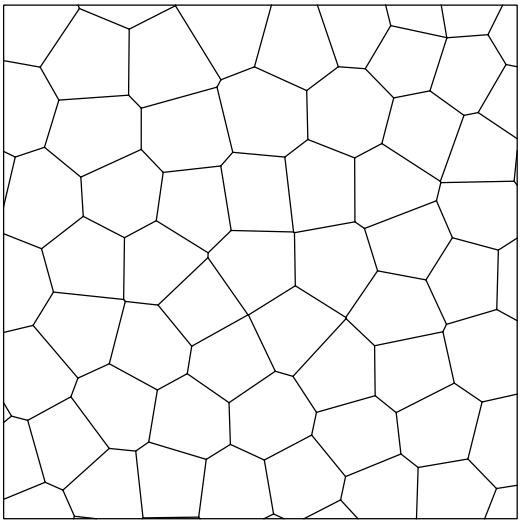
```
def grid():
    # Texture size
    size    = 2000

    # Horizontal stripes angle=0, widths=200
        and 80
    lines1 = StripesProperties(0,size/10,size
        /25)

    # Vertical stripes angle=pi/2, width=250
    lines2 = StripesProperties(pi/2,size/8)

    # Partition
    tex     = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Export result
    ExportSVG(tex,size)
```



Uniform

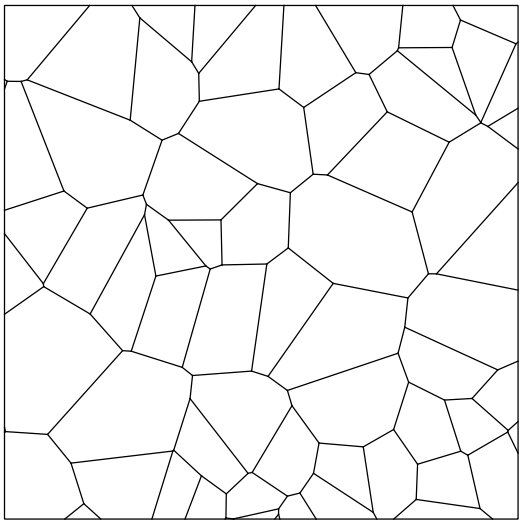
2 sec

```
def uniform():
    # Texture size
    size = 2000

    # Density ~= 50 faces
    prop = IrregularProperties(50/(size*size))

    # Partition
    tex  = UniformPartition(prop,KEEP_OUTSIDE)

    # Export result
    ExportSVG(tex,size)
```



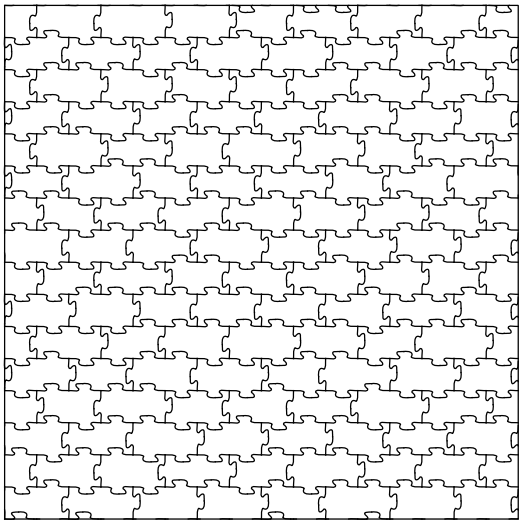
Random <1 sec

```
def random():
    # Texture size
    size = 2000

    # Density ~= 50 faces
    prop = IrregularProperties(50/(size*size))

    # Partition
    tex  = RandomPartition(prop,KEEP_OUTSIDE)

    # Export result
    ExportSVG(tex,size)
```



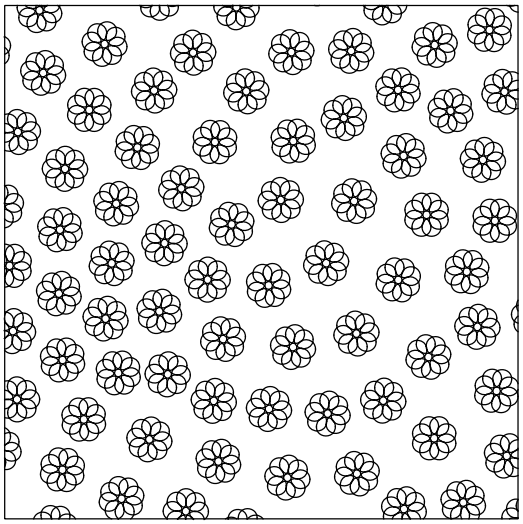
Puzzle 18 sec

```
def puzzle():
    size    = 2000
    zig     = ImportSVG("data/zig2.svg")

    # Grid partition, including edge labels
    lines1 = StripesProperties(0.0,size/16)
    lines2 = StripesProperties(pi/2.0,size/16)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: remove edges
    def grid_to_wall(edge):
        if((HasLabel(edge,"v1") and HasLabel(
            IncidentEdges(TargetVertex(edge)),"
            h1")) or
           (HasLabel(edge,"v2") and HasLabel(
               IncidentEdges(TargetVertex(edge)
               ),"h2"))):
            return Nothing()
        return ToCurve(edge)

    # Mapper: replace each edge by a curved
        line
    def edge_to_curve(edge):
        src_c = PointLabeled(zig,"start")
        dst_c = PointLabeled(zig,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        if Random(edge,0,1,0)<0.5:
            return MatchPoints(zig,src_c,dst_c,
                src_v,dst_v)
        else:
            return MatchPoints(zig,src_c,dst_c,
                dst_v,src_v)

    # Mapping operators
    wall_tex   = MapToEdges(grid_to_wall,
        grid_tex)
    puzzle_tex = MapToEdges(edge_to_curve,
        wall_tex)

    # Final texture
    ExportSVG(puzzle_tex,size)
```



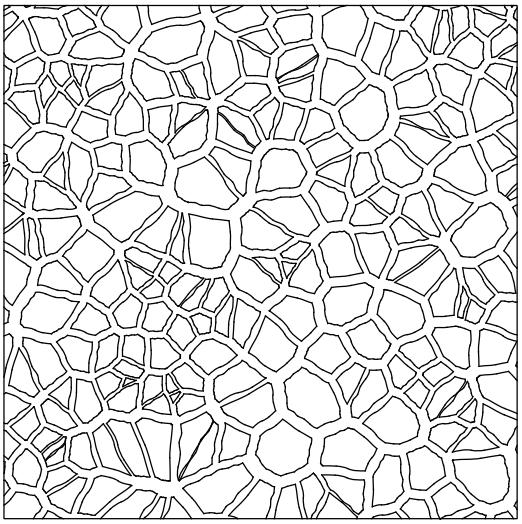
Rosettes 36 sec

```
def rosette():
    size      = 2000
    line      = ImportSVG("data/line.svg")
    circle    = ImportSVG("data/circle.svg")

    # Uniform partition (around 70 faces)
    props     = IrregularProperties(70/(size*
        size))
    part_tex = UniformPartition(props,
        KEEP_OUTSIDE)

    # Mapper: create a star in each face
    def face_to_star(face):
        rot    = Random(face,0.0,2.0*pi,0)
        elem1 = MatchPoint(line,BBoxCenter(line
            ),Centroid(face))
        elem2 = Append(Rotate(elem1,rot),Rotate
            (elem1,rot+pi/2))
        elem3 = Append(elem2,Rotate(elem2,pi/4)
            )
        return elem3

    # Mapper: place a circle on each vertex
    def vertex_to_circle(vertex):
        if(len(IncidentEdges(vertex))>1):
            return Nothing()
        return MatchPoint(circle,BBoxCenter(
            circle),Location(vertex))

    # Mapping operators
    stars_tex    = MapToFaces(face_to_star,
        part_tex)
    flowers_tex = MapToVertices(
        vertex_to_circle,stars_tex)

    # Final texture
    ExportSVG(flowers_tex,size)
```



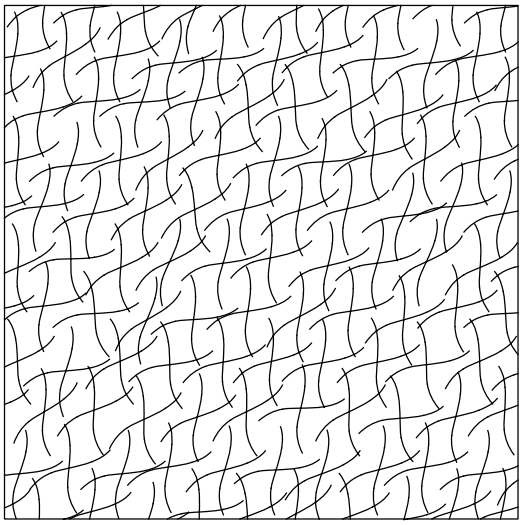
Cracks 35 sec

```
def cracks():
    size      = 2000
    lines     = [ImportSVG("data/line1.svg"),
                 ImportSVG("data/line2.svg"),
                 ImportSVG("data/line3.svg")]

    # Random partition (around 200 faces)
    props     = IrregularProperties(200/(size*
        size))
    part_tex = RandomPartition(props,
        KEEP_OUTSIDE)

    # Mapper: rescale each face
    def rescale_face(face):
        return Scale(Contour(face),0.8)

    # Mapper: replace each edge by a random
        curved line
    def edge_to_curve(edge):
        line   = lines[floor(Random(edge,0,len(
            lines),0))]
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    # Mapping operators
    reduced_tex = MapToFaces(rescale_face,
        part_tex)
    cracks_tex   = MapToEdges(edge_to_curve,
        reduced_tex)

    # Final texture
    ExportSVG(cracks_tex,size)
```



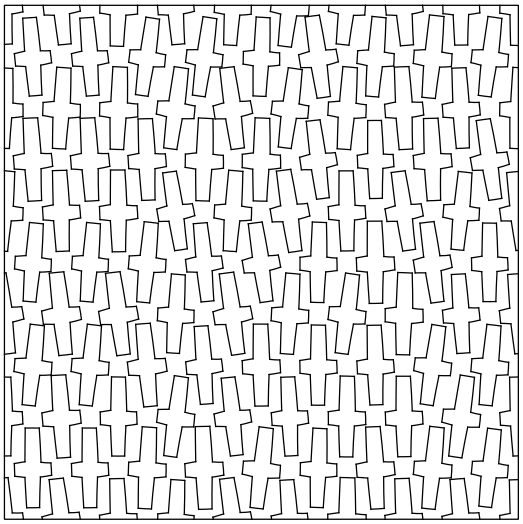
By example limitations: bimodal hatching 7 sec

```
def examplar_based_a():
    size    = 2000
    hatch   = ImportSVG("data/hatch.svg")

    # Grid partition, including face labels
    lines1 = StripesProperties(0,size/10)
    lines2 = StripesProperties(pi/2,size/20)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: place 2 hatches in one face on
        two
    def face_to_hatches(face):
        if ((HasLabel(face,"v1") and HasLabel(
            face,"h1")) or
            (HasLabel(face,"v2") and HasLabel(
                face,"h2"))):
            elem1 = Scale(hatch,Random(face
                ,2.6,2.7,1))
            elem2 = Rotate(elem1,pi/7+Random(
                face,-pi/12,pi/12,2))
            elem3 = Rotate(elem1,pi/2+Random(
                face,-pi/12,pi/12,3))
            elem4 = Append(elem2,elem3)
            w     = BBoxWidth(face)/7
            h     = BBoxHeight(face)/7
            pos   = Centroid(face)+Point(Random(
                face,-w,w,4),Random(face,-h,h,5)
                )
            return MatchPoint(elem4,BBoxCenter(
                elem4),pos)
        return Nothing()

    # Mapping operator
    hatch_tex = MapToFaces(face_to_hatches,
        grid_tex)

    # Final texture
    ExportSVG(hatch_tex,size)
```



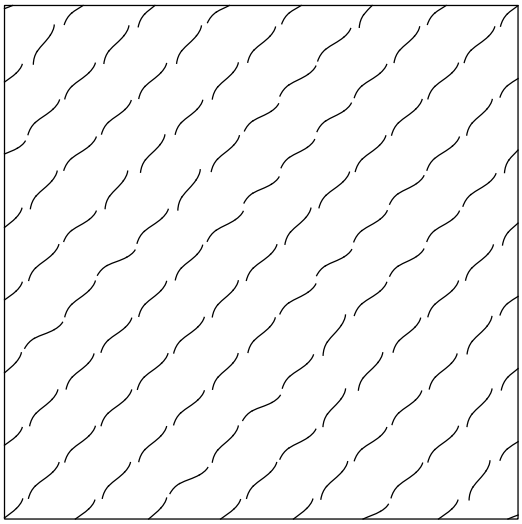
By example limitations: regular crosses 6 sec

```
def examplar_based_b():
    size    = 2000
    cross   = ImportSVG("data/cross.svg")

    # Grid partition, including face labels
    lines1 = StripesProperties(0,size/10)
    lines2 = StripesProperties(pi/2,size/18)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: place a cross in one face on two
    def face_to_hatches(face):
        if ((HasLabel(face,"v1") and HasLabel(
            face,"h1")) or
            (HasLabel(face,"v2") and HasLabel(
                face,"h2"))):
            elem1 = Scale(cross,Random(face
                ,1.35,1.45,2))
            elem2 = Rotate(elem1,Random(face,-pi
                /20,pi/20,4))
            return MatchPoint(elem2,BBoxCenter(
                elem2),BBoxCenter(face))
        return Nothing()

    # Mapping operator
    hatch_tex = MapToFaces(face_to_hatches,
        grid_tex)

    # Final texture
    ExportSVG(hatch_tex,size)
```



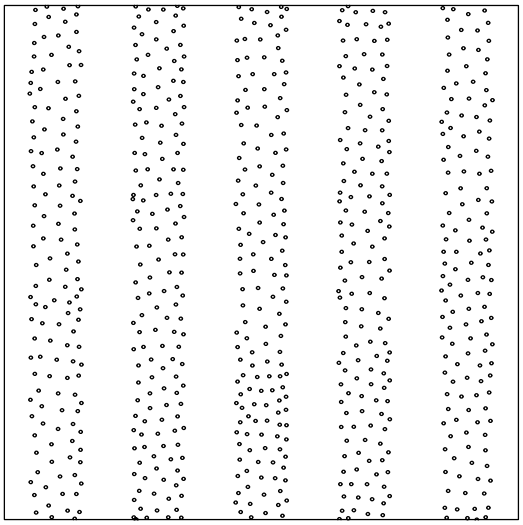
By example limitations: regular curves 4 sec

```
def examplar_based_c():
    size      = 2000
    hatch     = ImportSVG("data/hatch.svg")

    # Grid partition
    lines1    = StripesProperties(pi/4,size/10)
    lines2    = StripesProperties(pi/2+pi/4,
        size/10)
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: place a curved line in each face
    def face_to_hatch(face):
        src_p = BBoxCenter(hatch)
        dst_p = Centroid(face)
        elem1 = Scale(hatch,Random(face
            ,1.3,1.4,0))
        elem2 = Rotate(elem1,pi/4+Random(face,-
            pi/12,pi/12,1))
        return MatchPoint(elem2,src_p,dst_p)

    # Mapping operator
    hatch_tex = MapToFaces(face_to_hatch,
        grid_tex)

    # Final texture
    ExportSVG(hatch_tex,size)
```



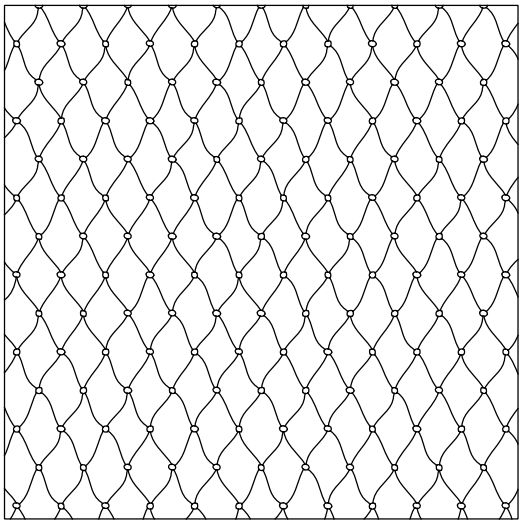
By example limitations: clusters 42 sec

```
def examplar_based_d():
    size    = 2000
    circle = ImportSVG("data/circle.svg")

    # Stripes partition, including face labels
    lines = StripesProperties(pi/2,size/10)
    SetFaceLabels(lines,"v1","v2")
    stripes_tex = StripesPartition(lines)

    # Mapper: place a circle in each face
    def face_to_circle(face):
        return Scale(MatchPoint(circle,
            BBoxCenter(circle),Centroid(face))
            ,0.02)

    # Mapper: creates a dot pattern in one
        face on two
    def face_to_circles(face):
        if HasLabel(face,"v2"):
            props = IrregularProperties(100/(
                BBoxWidth(face)*BBoxHeight(face)
                ))
            part   = UniformPartition(props,
                CROP_ADD_BOUNDARY)
            circ   = MapToFaces(face_to_circle,
                part)
            return circ(face)
        return Nothing()

    # Mapping operator
    texture = MapToFaces(face_to_circles,
        stripes_tex)

    # Final texture
    ExportSVG(texture,size)
```



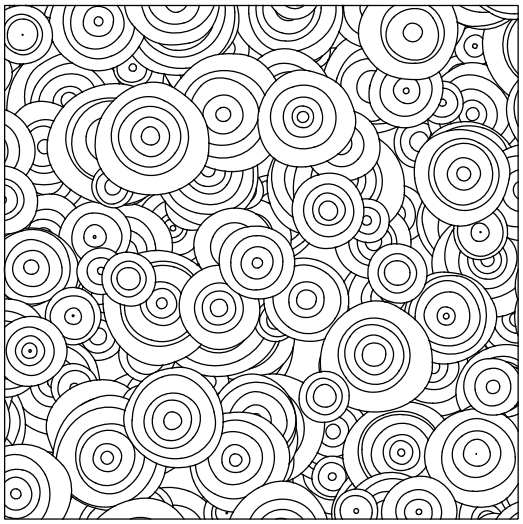
Teaser: example a 25 sec

```
def teaser_a():
    # Load multiple curve and blob shapes
    curves = [ImportSVG("data/curve1.svg"),
              ImportSVG("data/curve2.svg"),
              ImportSVG("data/curve3.svg"),
              ImportSVG("data/curve4.svg")]
    blobs  = [ImportSVG("data/blob1.svg"),
              ImportSVG("data/blob2.svg"),
              ImportSVG("data/blob3.svg"),
              ImportSVG("data/blob4.svg")]
    size = 2000

    # Mapper: replace each edge by a random curve
    def edge_to_curve(edge):
        curve = curves[floor(Random(edge,0,len(curves),0))]
        src_c = PointLabeled(curve,"bottom")
        dst_c = PointLabeled(curve,"top")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(curve,src_c,dst_c,src_v,dst_v)

    # Mapper: replace each vertex by a random blob
    def vertex_to_blob(vertex):
        blob = blobs[floor(Random(vertex,0,len(blobs),0))]
        return MatchPoint(blob,BBoxCenter(blob),Location(vertex))

    # Grid Partition
    lines1    = StripesProperties(pi/3,150)
    lines2    = StripesProperties(-pi/3,150)
    grid_tex  = GridPartition(lines1,lines2,KEEP_OUTSIDE)

    # Mapping operators
    curve_tex = MapToEdges(edge_to_curve,grid_tex)
    blob_tex  = MapToVertices(vertex_to_blob,grid_tex)

    # Combining
    final_tex = Union(blob_tex,Outside(curve_tex,blob_tex,CROP))

    # Final texture
    ExportSVG(final_tex,size)
```



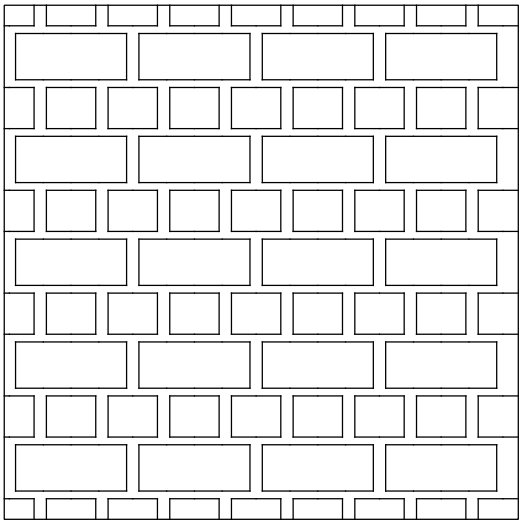
Teaser: example b 2 min, 9 sec

```
# create a texture with non-overlapping concentric circles
def circles_in_circles(density):
    cir = [ImportSVG("data/circle1.svg"),
           ImportSVG("data/circle2.svg"),
           ImportSVG("data/circle3.svg")]

    # Uniform partition
    props  = IrregularProperties(density)
    part   = UniformPartition(props, KEEP_OUTSIDE)

    # Mapper: place circles in each face
    def face_to_circles(face):
        all_circ  = Nothing()
        cur_scale = Random(face,0.3,0.8,0);

        # Perturb shape and scale for each circle
        i = 0
        while cur_scale>0.0:
            circle    = cir[floor(Random(face,0,len(circles),i))]
            cur_circ  = Scale(MatchFace(circle, face),cur_scale)
            cur_circ  = Rotate(cur_circ,Random(face,0,2*pi,i+10))
            all_circ  = Append(all_circ,cur_circ)
            cur_scale = cur_scale-Random(face,0.09,0.2,i+20)
            i = i+1

        return all_circ

    return MapToFaces(face_to_circles,part)

def teaser_b():
    size = 2000

    # Create a first layer of circles
    tex = circles_in_circles(15/(size*size))

    # Combine layers
    for i in range(0,9):
        tmp = circles_in_circles(15/(size*size))
        tex = Union(tmp,Outside(tex,tmp,CROP))

    ExportSVG(tex,size)
```



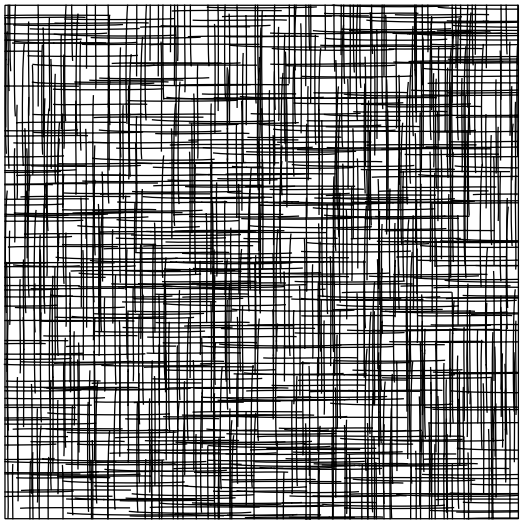
Teaser: example c 1 sec

```
def teaser_c():
    size    = 2000

    # Grid partition, including edge labels
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,120)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2","v3","v4")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: remove edges containing specific
        label sequences
    def remove_edge(edge):
        if HasLabel(edge,"h1") or HasLabel(edge
            ,"h2"):
            return ToCurve(edge)
        if HasLabel(IncidentEdges(TargetVertex(
            edge)),"h1"):
            if HasLabel(edge,"v1"):
                return ToCurve(edge)
            else:
                return Nothing()
        else:
            if HasLabel(edge,"v2") or HasLabel(
                edge,"v4"):
                return ToCurve(edge)
            else:
                return Nothing()

    # Mapper: rescale each face
    def shrink_face(face):
        if HasLabel(IncidentEdges(face),"v1"):
            return Scale(Contour(face),0.9)
        else:
            return Scale(Contour(face),0.8)

    # Mapping operator
    wall_tex    = MapToEdges(remove_edge,
        grid_tex)
    shrink_tex = MapToFaces(shrink_face,
        wall_tex)

    # Final texture
    ExportSVG(shrink_tex,size)
```



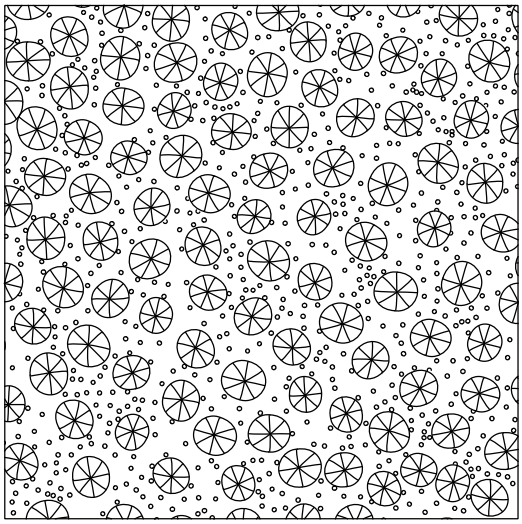
Teaser: example d 3 min, 40 sec

```
def teaser_d():
    # Load several curves
    curves    = [ImportSVG("data/bighatch1.svg"),
                 ImportSVG("data/bighatch2.svg"),
                 ImportSVG("data/bighatch3.svg"),
                 ImportSVG("data/bighatch4.svg")]
    size      = 2000

    # Uniform partition (with around 1000 faces)
    props     = IrregularProperties(1000/(size*size))
    init_tex  = UniformPartition(props, KEEP_OUTSIDE)

    # Mapper: place a vertical or horizontal curve on each face
    def face_to_hatches(face):
        curve = curves[floor(Random(face,0,len(curves),0))]
        if Random(face,0,1,1)>0.5:
            curve = Rotate(curve,pi/2)
        return MatchPoint(curve,BBoxCenter(curve),Centroid(face))

    # Mapping operator
    hatch_tex = MapToFaces(face_to_hatches, init_tex)

    # Final texture
    ExportSVG(hatch_tex,size)
```



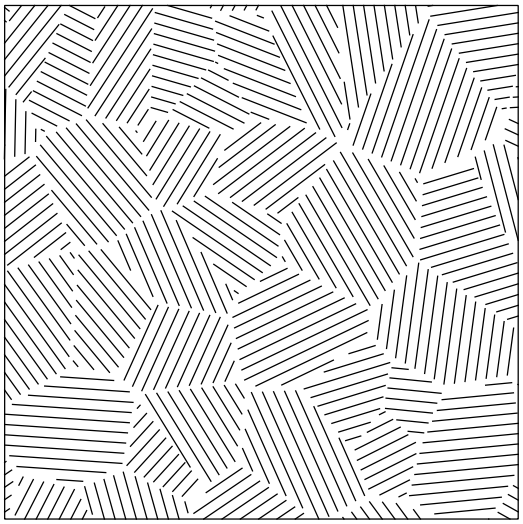
Teaser: example e 20 sec

```
def teaser_e():
    size     = 2000
    wheel    = ImportSVG("data/wheel1.svg")
    stipple  = ImportSVG("data/stipple1.svg")

    # Uniform partition (around 100 faces)
    props1   = IrregularProperties(100/(size*
        size))
    tex1     = UniformPartition(props1,
        KEEP_OUTSIDE)

    # Random partition (around 1200 faces)
    props2   = IrregularProperties(1200/(size*
        size))
    tex2     = RandomPartition(props2,
        KEEP_OUTSIDE)

    # Mapper: place a wheel in each face
    def face_to_wheel(face):
        w = Scale(Rotate(wheel,Random(face,0,2*
            pi,0)),Random(face,0.8,1,1))
        return MatchPoint(w,BBoxCenter(w),
            Centroid(face))

    # Mapper: place a stipple in each face
    def face_to_stipples(face):
        s = Scale(Rotate(stipple,Random(face
            ,0,2*pi,0)),Random(face,0.9,1,1))
        return MatchPoint(s,BBoxCenter(s),
            Centroid(face))

    # Mapping operators
    tex_wheel    = MapToFaces(face_to_wheel,
        tex1)
    tex_stipples = MapToFaces(face_to_stipples
        ,tex2)

    # Combining operators
    texture = Union(tex_wheel,Outside(
        tex_stipples,tex_wheel,KEEP_INSIDE))

    # Final texture
    ExportSVG(texture,size)
```



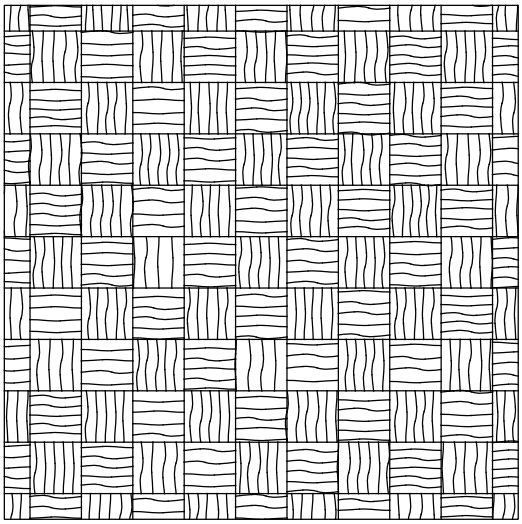
Teaser: example f 3 sec

```
def teaser_f():
    # Random partition (around 30 faces)
    size      = 2000
    props     = IrregularProperties(30/(size*
        size))
    init_tex = RandomPartition(props,
        KEEP_OUTSIDE)

    # Mapper: rescale each face
    def scale_map(face):
        return Scale(Contour(face),0.95)

    # Mapper: produces stripes on each face
    def hatch_map(face):
        angle = Random(face,0,2*pi,1)
        lines = StripesProperties(angle,40)
        return StripesPartition(lines)(face)

    # Mapper: remove boundary edges
    def border_map(edge):
        if IsBoundary(edge):
            return Nothing()
        return ToCurve(edge)

    # Mapping operators
    tex1 = MapToFaces(scale_map,init_tex)
    tex2 = MapToFaces(hatch_map,tex1)
    tex3 = MapToEdges(border_map,tex2)

    # Final texture
    ExportSVG(tex3,size)
```



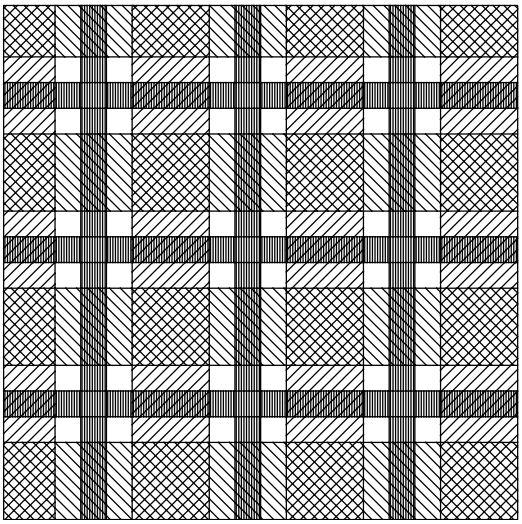
Teaser: example g 6 sec

```
def teaser_g():
    curves = [ImportSVG("data/line1.svg"),
              ImportSVG("data/line2.svg")]

    # Grid partition, including face labels
    size   = 2000
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    gridl_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: replace each edge by a random
        curve
    def edge_to_curve(edge):
        if IsBoundary(edge):
            return ToCurve(edge)
        curve = curves[floor(Random(edge,0,len(
            curves),0))]
        src_c = PointLabeled(curve,"start")
        dst_c = PointLabeled(curve,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        if Random(edge,0,1,1)<0.5:
            return MatchPoints(curve,src_c,dst_c
                ,src_v,dst_v)
        else:
            return MatchPoints(curve,dst_c,src_c
                ,src_v,dst_v)

    # Mapper: creates stripes in each face
    def face_to_stripes(face):
        width = BBoxWidth(face)/Random(face
            ,4,6,0)
        theta = 0
        if((HasLabel(face,"h1") and HasLabel(
            face,"v1")) or
           (HasLabel(face,"h2") and HasLabel(
               face,"v2"))):
            theta = pi/2
        lines   = StripesProperties(theta,width
            )
        stripes = StripesPartition(lines)
        return MapToEdges(edge_to_curve,stripes
            )(face)

    # Mapping operator
    tiled_tex = MapToFaces(face_to_stripes,
        gridl_tex)

    # Final texture
    ExportSVG(tiled_tex,size)
```



Teaser: example h 26 sec

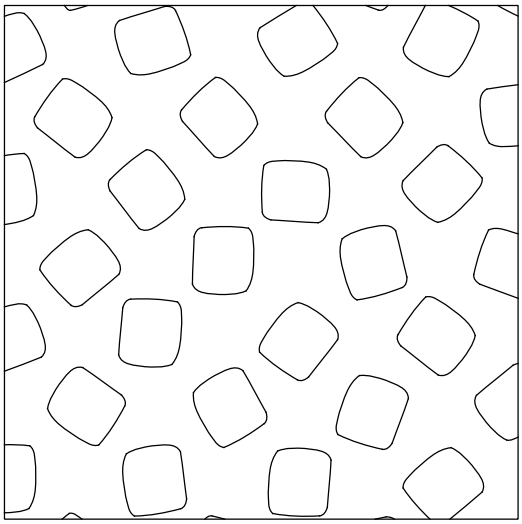
```
def teaser_h():
    size    = 2000

    # Stripe partition
    lines1 = StripesProperties(0.0,300)
    SetFaceTags(lines1,"h1","h2")
    stripes1 =  StripesPartition(lines1)

    # Stripe partition
    lines2 = StripesProperties(pi/2,300)
    SetFaceLabels(lines2,"v1","v2")
    stripes2 =  StripesPartition(lines2)

    # Grid partition with labels
    lines3 = StripesProperties
        (0.0,100,100,100,300)
    SetFaceLabels(lines3,"h1","h2","h3","h4")
    lines4 = StripesProperties(pi
        /2,100,100,100,300)
    SetFaceLabels(lines4,"v1","v2","v3","v4")
    grid = GridPartition(lines3,lines4,
        KEEP_OUTSIDE)

    # Create a stripe partition with specified
        angle and width
    def hatch(angle,width):
        lines = StripesProperties(angle,width)
        return StripesPartition(lines)

    # Mapper: create -pi/4 stripes in each
        face
    def hatch_map_stripes1(f):
        angle = -pi/4
        if HasLabel(f,"h1"):
            return hatch(angle,30)(f)
        else:
            return Nothing()

    # Mapper: create pi/4 stripes in each face
    def hatch_map_stripes2(f):
        angle = pi/4
        if HasLabel(f,"v1"):
            return hatch(angle,30)(f)
        else:
            return Nothing()

    # Mapper: create pi/2 stripes in specific
        faces
    def hatch_map_grid(f):
        angle = pi/2
        if HasLabel(f,"h1"):
            return hatch(angle,10)(f)
        elif HasLabel(f,"v1"):
            return hatch(angle,10)(f)
        else:
            return Nothing()

    # Mapping operators
    tex1 = MapToFaces(hatch_map_stripes1,
        stripes1)
    tex2 = MapToFaces(hatch_map_stripes2,
        stripes2)
    tex4 = MapToFaces(hatch_map_grid,grid)

    # Merging operators
    tex3 = Union(tex1,tex2)
    tex5 = Union(tex3,tex4)

    ExportSVG(tex5,size)
```



Combining a: texture 1

1 sec

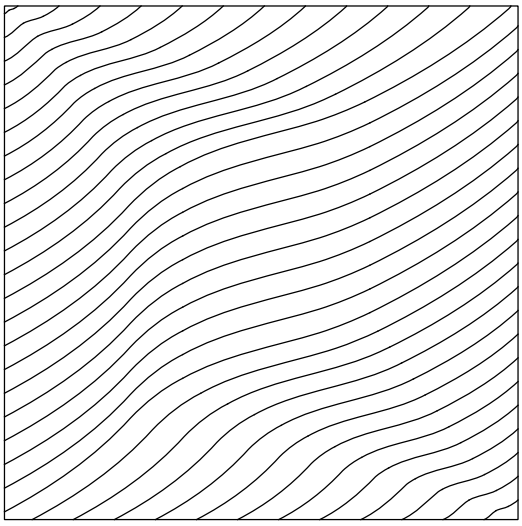
```
def squares(theta,width):
    # Grid partition with given width and
        orientation
    lines1    = StripesProperties(theta,width)
    lines2    = StripesProperties(theta+pi/2.0,
        width)
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)
    square    = ImportSVG("data/square.svg")

    # Mapper: place a rounded square in each
        face
    def face_to_square(face):
        return Rotate(MatchFace(square,face),
            Random(face,0.0,2.0*pi,1))

    # Return the texture genrated via the
        mapping operator
    return MapToFaces(face_to_square,grid_tex)

def combine_a():
    tex1 = squares(pi/4,400)
    ExportSVG(tex1,2000)
```



Combining b: texture 2

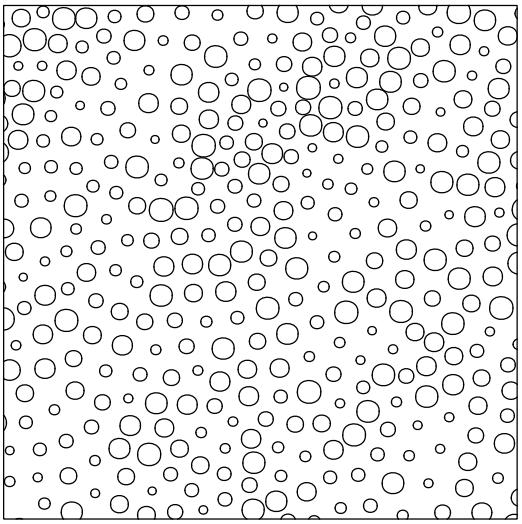
1 sec

```
def curves(theta,width):
    # Stripes partition with given width and
        orientation
    props    = StripesProperties(theta,width)
    stripes = StripesPartition(props)
    line     = ImportSVG("data/line6.svg")

    # Mapper: replace each edge by a curve
    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    # Return the texture genrated via the
        mapping operator
    return MapToEdges(line_to_curve,stripes)

def combine_b():
    tex2 = curves(pi/6,80)
    ExportSVG(tex2,2000)
```



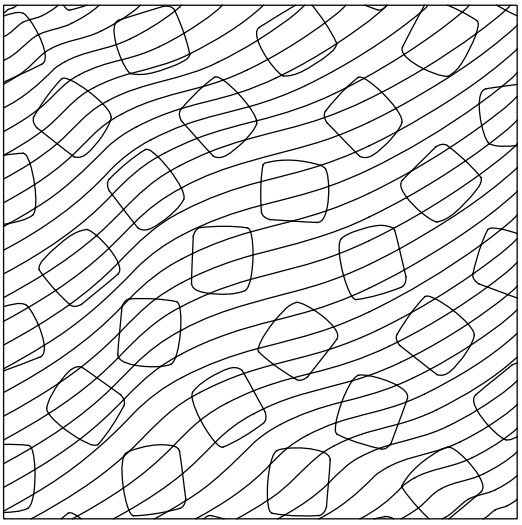
Combining c: texture 3 13 sec

```
def circles(density):
    # Uniform partition with given density
    props  = IrregularProperties(density)
    part   = UniformPartition(props,
         KEEP_OUTSIDE)
    circle = ImportSVG("data/circle.svg")

    # Mapper: place a circle in each face
    def face_to_circle(face):
        src_p = BBoxCenter(circle)
        dst_p = Centroid(face)
        return Scale(MatchPoint(circle,src_p,
             dst_p),Random(face,0.05,0.15,0))

    # Mapping operator
    return MapToFaces(face_to_circle,part)

def combine_c():
    tex3 = circles(8.5e-5)
    ExportSVG(tex3,2000)
```



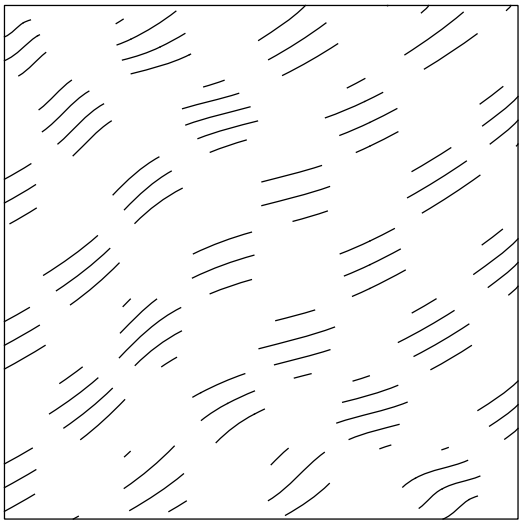
Combining d: merging tex1 and tex2 4 sec

```
def combine_d():
    tex1 = squares(pi/4,400)
    tex2 = curves(pi/6,80)
    texf = Union(tex1,tex2)
    ExportSVG(texf,2000)
```



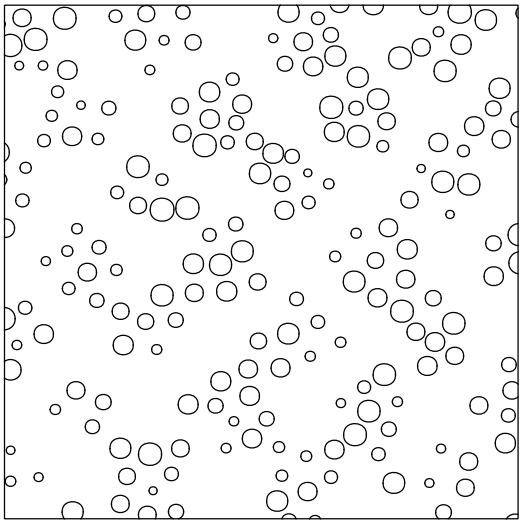
Combining e: tex2 inside tex1 (crop) 3 sec

```
def combine_e():
    tex1 = squares(pi/4,400)
    tex2 = curves(pi/6,80)
    texf = Inside(tex2,tex1,CROP)
    ExportSVG(texf,2000)
```



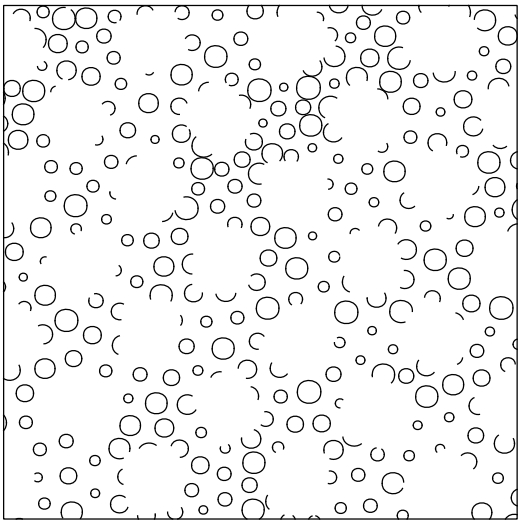
Combining f: tex3 inside tex1 (keep outside) 15 sec

```
def combine_f():
    tex1 = squares(pi/4,400)
    tex3 = circles(8.5e-5)
    texf = Inside(tex3,tex1,KEEP_OUTSIDE)
    ExportSVG(texf,2000)
```



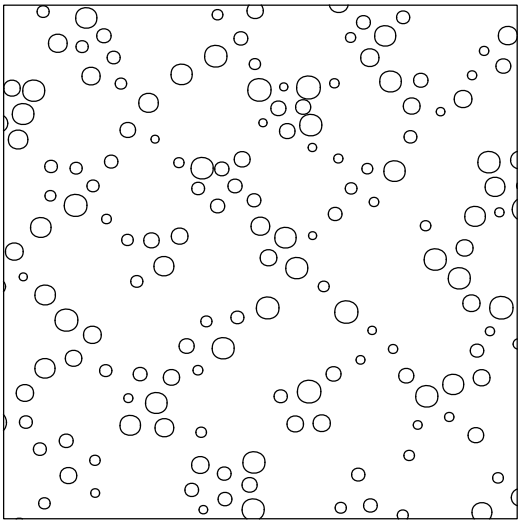
Combining g: tex3 outside tex1 (crop) 15 sec

```
def combine_g():
    tex1 = squares(pi/4,400)
    tex3 = circles(8.5e-5)
    texf = Outside(tex3,tex1,CROP)
    ExportSVG(texf,2000)
```



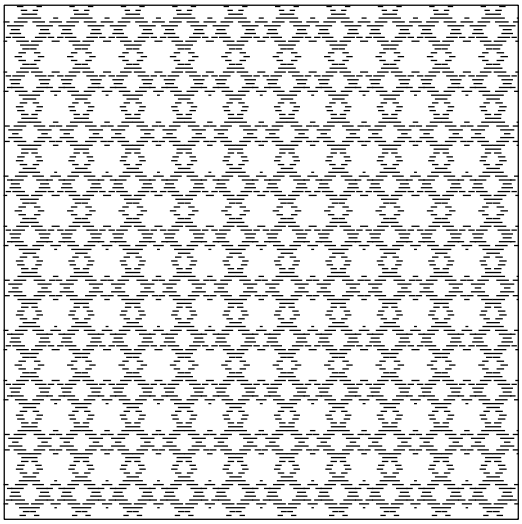
Combining h: tex3 outside tex1 (keep inside) 15 sec

```
def combine_h():
    tex1 = squares(pi/4,400)
    tex3 = circles(8.5e-5)
    texf = Outside(tex3,tex1,KEEP_INSIDE)
    ExportSVG(texf,2000)
```



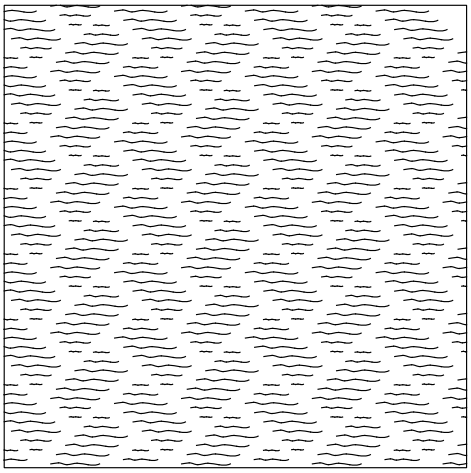
Regular result (a) 30 sec

```
def complex_regular01():
    size    = 2000

    # Grid partition
    lines1 = StripesProperties(0.0,200)
    lines2 = StripesProperties(pi/2.0,200)
    grid    = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Stripe partition
    lines = StripesProperties(0,15)
    hatch = StripesPartition(lines)

    # Mapper: rotate faces with a pi/4 angle
    def rotate_map(f):
        return Rotate(Contour(f), pi/4)

    # Mapper: scale each face with a factor
        0.8
    def scale_map(f):
        return Scale(Contour(f), 0.8)

    # Mapping operators (rotate/scale)
    tex1 = MapToFaces(rotate_map, grid)
    tex2 = MapToFaces(scale_map, tex1)

    # Combining operator
    tex3 = Outside(hatch,tex2,CROP)

    ExportSVG(tex3,size)
```



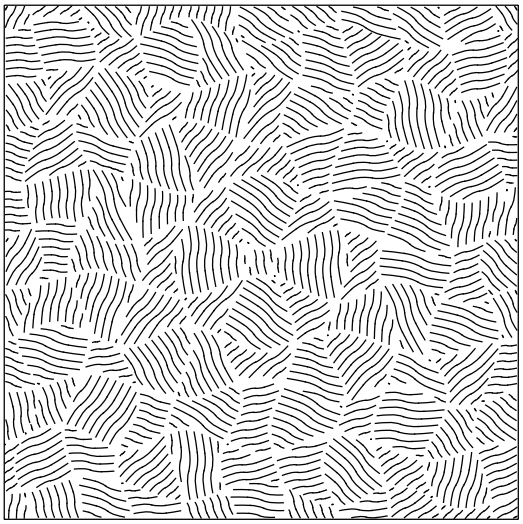
Regular result (b) 39 sec

```
def complex_regular02():
    size = 2000
    line = ImportSVG("data/line3.svg")

    # Grid partition and labels
    lines1 = StripesProperties(pi/4.0,200)
    lines2 = StripesProperties(-pi/4.0,200)
    SetEdgeTags(lines1,"h1","h2","h3","h4")
    SetEdgeTags(lines2,"v1","v2","v3","v4")
    grid = GridPartition(lines1,lines2,KEEP_OUTSIDE)

    # Mapper: keep or remove edge
    def grid_to_V_mapper(edge):
        if HasTag(edge,"v1") and HasTag(
            IncidentEdges(TargetVertex(edge)),"h1"):
            return Nothing()
        if HasTag(edge,"v2") and HasTag(
            IncidentEdges(TargetVertex(edge)),"h2"):
            return Nothing()
        if HasTag(edge,"v3") and HasTag(
            IncidentEdges(TargetVertex(edge)),"h3"):
            return Nothing()
        if HasTag(edge,"v4") and HasTag(
            IncidentEdges(TargetVertex(edge)),"h4"):
            return Nothing()
        if HasTag(edge,"h1") and HasTag(
            IncidentEdges(TargetVertex(edge)),"v3"):
            return Nothing()
        if HasTag(edge,"h2") and HasTag(
            IncidentEdges(TargetVertex(edge)),"v4"):
            return Nothing()
        if HasTag(edge,"h3") and HasTag(
            IncidentEdges(TargetVertex(edge)),"v1"):
            return Nothing()
        if HasTag(edge,"h4") and HasTag(
            IncidentEdges(TargetVertex(edge)),"v2"):
            return Nothing()
        return ToCurve(edge)

    # Mapper: rescale face
    def scale_map(f):
        return Scale(Contour(f), 0.8)
    # Mapper: replace each edge by a curve
    def map_curve_to_edge(edge):
       start = PointTagged(line,"start")
       end   = PointTagged(line,"end")
       if IsBoundary(edge):
          return Nothing()
       else:
          return MatchPoints(line,start,end,Location
              (SourceVertex(edge)),Location(
              TargetVertex(edge)))
    # Create a stripe partition
    def hatch(angle):
       lines = StripesProperties(angle,40)
       return StripesPartition(lines)
    # Mapper: create stripes in each face
    def hatch_map(f):
       angle = 0.0
       return hatch(angle)(f)
    # Mapping operator: select grid edges
    tex1 = MapToEdges(grid_to_V_mapper,grid)
    # Mapping operator: rescale faces
    tex2 = MapToFaces(scale_map, tex1)
    # Mapping operator: place hatches faces
    tex3 = MapToFaces(hatch_map, tex2)
    # Mapping operator: replace hatches by smooth
        curves
    tex4 = MapToEdges(map_curve_to_edge, tex3)

    ExportSVG(tex4,size)
```



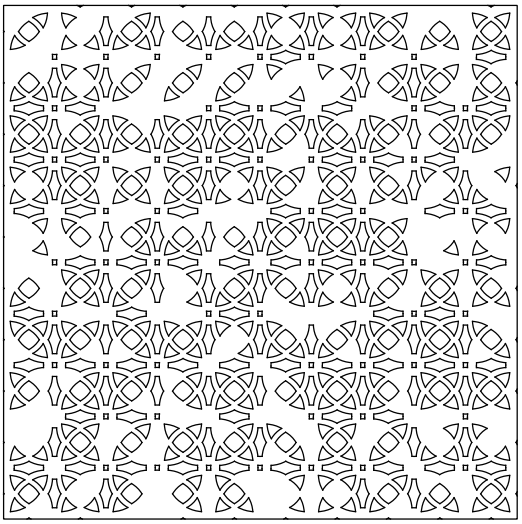
Regular result (c) 30 sec

```
def complex_regular03():
    size    = 2000
    line = ImportSVG("data/line4.svg")
    line2 = ImportSVG("data/line2.svg")

    # Stripe partitions and their labels
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"h1","h2")
    stripes1 = StripesPartition(lines1)
    stripes2 = StripesPartition(lines2)

    # Mapper: replace each edge by a curve
    def map_curve_to_edge(edge):
        start = PointTagged(line,"start")
        end   = PointTagged(line,"end")
        if IsBoundary(edge):
            return ToCurve(edge)
        elif  HasLabel(edge,"h1"):
            return MatchPoints(line,start,end,
                Location(SourceVertex(edge)),
                Location(TargetVertex(edge)))
        else:
            return MatchPoints(line,end,start,
                Location(SourceVertex(edge)),
                Location(TargetVertex(edge)))

    # Mapper: replace each edge by a curve
    def map_curve_to_edge2(edge):
        start = PointTagged(line2,"start")
        end   = PointTagged(line2,"end")
        if IsBoundary(edge):
            return Nothing()
        else:
            return MatchPoints(line2,start,end,
                Location(SourceVertex(edge)),
                Location(TargetVertex(edge)))

    # Mapper: create stripes in each face with
        random orientations
    def hatch_map(f):
        angle = Random(f,0,2*pi,0)
        lines = StripesProperties(angle,30)
        hatch = StripesPartition(lines)
        return hatch(f)

    # Mapper: rescale each face
    def scale_map(f):
        return Scale(Contour(f), 0.9)

    # Mapping / combining operators
    tex1 = MapToEdges(map_curve_to_edge,
        stripes1)
    tex2 = MapToEdges(map_curve_to_edge,
        stripes2)
    tex3 = Union(tex1,tex2)
    tex4 = MapToFaces(scale_map,tex3)
    tex5 = MapToFaces(hatch_map,tex4)
    tex6 = MapToEdges(map_curve_to_edge2,tex5)

    ExportSVG(tex6,size)
```



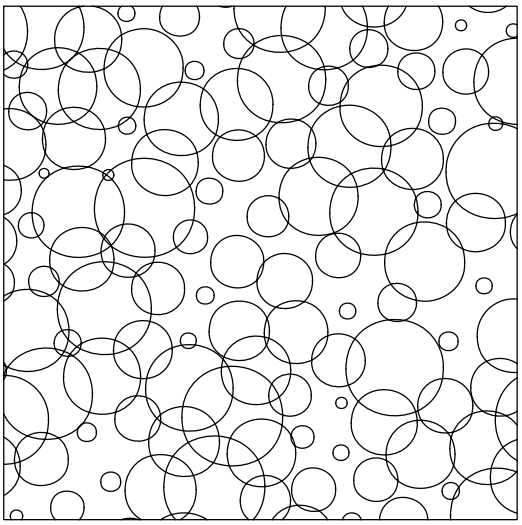
Regular result (d) 25 sec

```
def complex_regular04():
    size   = 2000
    circle = ImportSVG("data/circle.svg")

    # Grid partition
    lines1 = StripesProperties(pi/2.0,200)
    lines2 = StripesProperties(pi,200)
    partition1 = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: place a circle in each face
    def circle_map(f):
        return Scale(MatchFace(circle,f), 2.5)

    # Mapper: rescale face
    def scale_map(f):
        return Scale(Contour(f), 0.65)

    # Mapper: keep or remove face
    def delete_map(f):
        r = Random(f,0,1,2)
        if r<0.3:
            return Nothing()
        else:
            return Contour(f)

    # Mapping operators
    tex1 = MapToFaces(circle_map,partition1)
    tex2 = MapToFaces(scale_map,tex1)
    tex3 = MapToFaces(delete_map,tex2)

    ExportSVG(tex3,size)
```



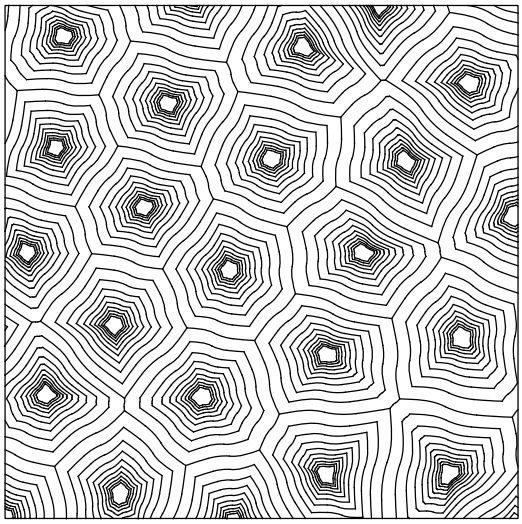
Non regular result (a) left 6 sec

```
def complex_non_regular01():
    size   = 2000
    circle = ImportSVG("data/circle.svg")

    # Uniform partition
    density = 100/(size*size)
    props = IrregularProperties(density)
    partition= UniformPartition(props,
        KEEP_OUTSIDE)

    # Mapper: place a circle in each face
    def circle_map(f):
        return Scale(MatchFace(circle,f),
            Random(f,0.2,2.0,2))

    # Mapping operator
    tex1 = MapToFaces(circle_map,partition)

    ExportSVG(tex1,size)
```



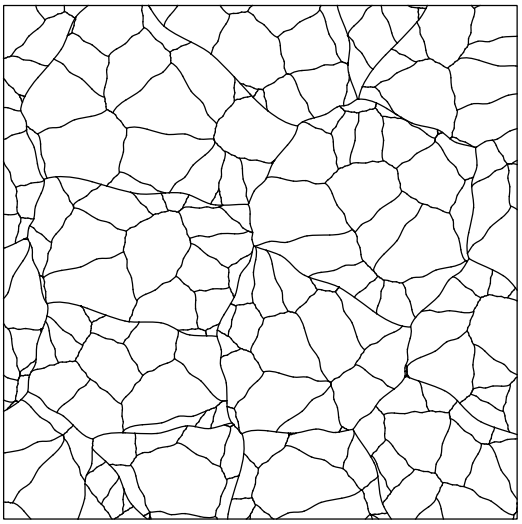
Non regular result (a) right 29 sec

```
def complex_non_regular02():
    size    = 2000
    line = ImportSVG("data/line2.svg")

    # Uniform partition
    density = 20/(2000*2000)
    props = IrregularProperties(density)
    partition= UniformPartition(props,
        KEEP_OUTSIDE)

    # Mapper: replace each edge by a curve
    def map_curve_to_edge(edge):
        start = PointTagged(line,"start")
        end   = PointTagged(line,"end")
        return MatchPoints(line,start,end,
            Location(SourceVertex(edge)),
            Location(TargetVertex(edge)))

    # Mapper: rescale a face
    def scale_map(f):
        return Scale(Contour(f), 0.8)

    # Mapping operators (iteratively rescale
        faces)
    l1 = MapToEdges(map_curve_to_edge,
        partition)
    l2 = MapToFaces(scale_map,l1)
    l3 = MapToFaces(scale_map,l2)
    l4 = MapToFaces(scale_map,l3)
    l5 = MapToFaces(scale_map,l4)
    l6 = MapToFaces(scale_map,l5)
    l7 = MapToFaces(scale_map,l6)
    l8 = MapToFaces(scale_map,l7)
    l9 = MapToFaces(scale_map,l8)
    l10 = MapToFaces(scale_map,l9)

    # Combining operators: merge all
        arrangements
    tex = Union(Union(Union(Union(Union(Union(
        Union(Union(Union(l1,l2),l3),l4),l5),
        l6),l7),l8),l9),l10)

    ExportSVG(tex,size)
```



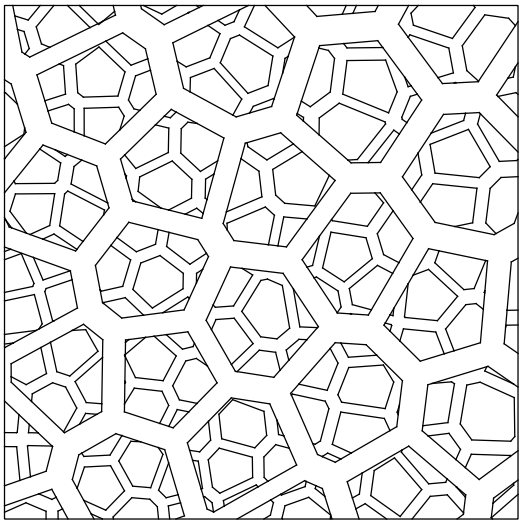
Non regular result (b) 12 sec

```
def complex_non_regular03():
    size    = 2000
    line = ImportSVG("data/line3.svg")

    # Random partition
    density = 10/(size*size)
    props = IrregularProperties(density)
    partition = RandomPartition(props,
        KEEP_OUTSIDE)

    # Mapper: replace internal edges by curves
    def map_curve_to_edge(edge):
        start = PointTagged(line,"start")
        end   = PointTagged(line,"end")
        if IsBoundary(edge):
            return ToCurve(edge)
        else:
            return MatchPoints(line,end,start,
                Location(SourceVertex(edge)),
                Location(TargetVertex(edge)))

    # Mapper: create a random partition in
        each face
    def partition_map(f):
        density2 = 100/(size*size)
        props2 = IrregularProperties(density2)
        partition2 = RandomPartition(props2,
            CROP_ADD_BOUNDARY)
        return partition2(f)

    # Mapping operators
    tex1 = MapToEdges(map_curve_to_edge,
        partition)
    tex2 = MapToFaces(partition_map, tex1)
    tex3 = MapToEdges(map_curve_to_edge,tex2)

    ExportSVG(tex3,size)
```



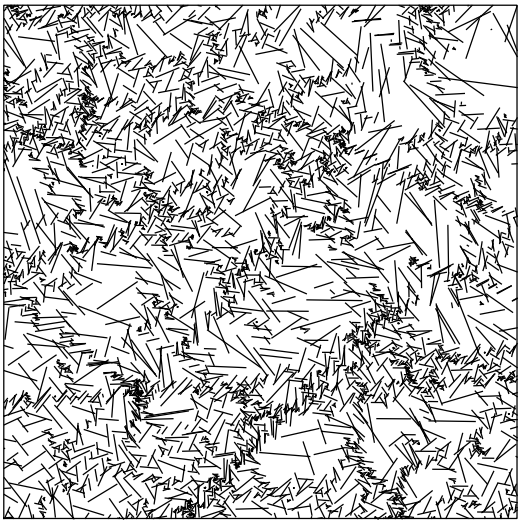
Non regular result (c) 30 sec

```
def complex_non_regular04():
    size    = 2000

    # Uniform partition
    density1 = 20/(size*size)
    props1 = IrregularProperties(density1)
    partition1 = UniformPartition(props1,
        KEEP_OUTSIDE)

    # Uniform partition
    density2 = 90/(size*size)
    props2 = IrregularProperties(density2)
    partition2 = UniformPartition(props2,
        KEEP_OUTSIDE)

    # Mapper: rescale faces
    def scale_map(f):
        return Scale(Contour(f), 0.8)

    # Mapper: create a uniform partition in
        each face
    def partition2_map(f):
       return MapToFaces(scale_map,partition2)
           (f)

    # Mapping operators
    tex1 = MapToFaces(scale_map,partition1)
    tex2 = MapToFaces(scale_map,partition2)

    # Combining operators
    tex3 = Inside(tex2,tex1,CROP)
    tex4 = Union(tex1,tex3)

    ExportSVG(tex4,size)
```



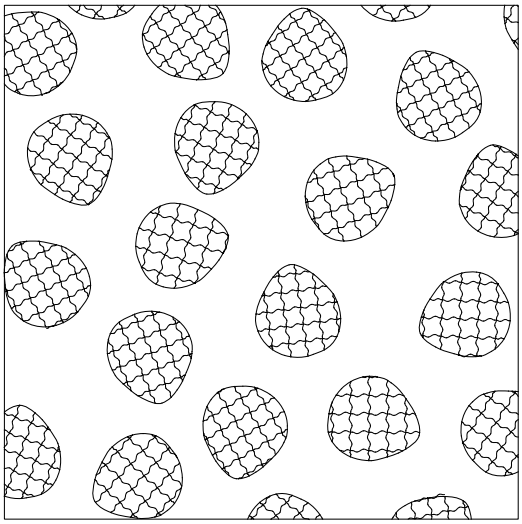
Non regular result (d) 1 min, 28 sec

```
def complex_non_regular05():
    size    = 2000

    # Random partition
    density = 30/(2000*2000)
    props = IrregularProperties(density)
    partition= RandomPartition(props,
        KEEP_OUTSIDE)

    # Mapper: rotate edge
    def rotate_e(e):
        return Rotate(ToCurve(e), pi/4)

    # Mapper: rescale edge
    def scale_map_5(e):
        return Scale(ToCurve(e), 5.0)

    # Mapper: rescale edge
    def scale_map_10(e):
        return Scale(ToCurve(e), 10.0)

    # Mapping operators
    tex1 = MapToEdges(scale_map_5,partition)
    tex2 = MapToEdges(scale_map_10,partition)
    tex3 = MapToEdges(rotate_e,tex2)

    ExportSVG(tex3,size)
```



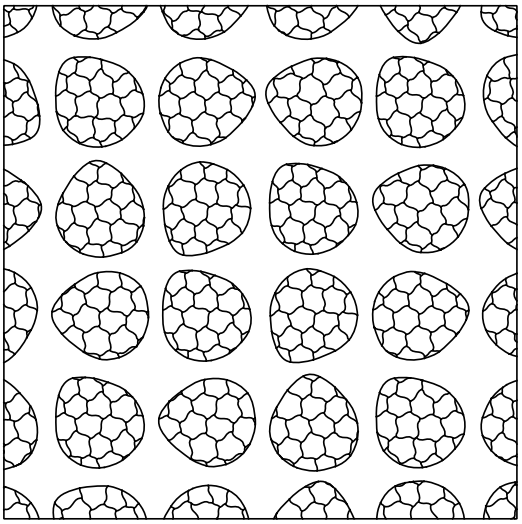
Script edition (a) original script 25 sec

```
def overview_orig():
    size = 2500
    blob = ImportSVG("data/blob.svg")
    zig  = ImportSVG("data/zig.svg")

    # Mapper: place a blob in each 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: replace each 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)

    def create_blob_tex():
        # Uniform partition
        props    = IrregularProperties
            (11/(2000*2000))
        init_tex = UniformPartition(props,
            KEEP_OUTSIDE)

        # Mapping operators
        return MapToFaces(map_blob_to,init_tex)

    # Mapper: generate a texture in each face
    def create_zig_tex(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_tex = GridPartition(lines1,lines2,
            CROP_ADD_BOUNDARY)

        # Mapping operator
        texture2 = MapToEdges(map_curve_to,
            init_tex)
        return texture2(face)

    init_tex  = create_blob_tex()
    final_tex = MapToFaces(create_zig_tex,
        init_tex)

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



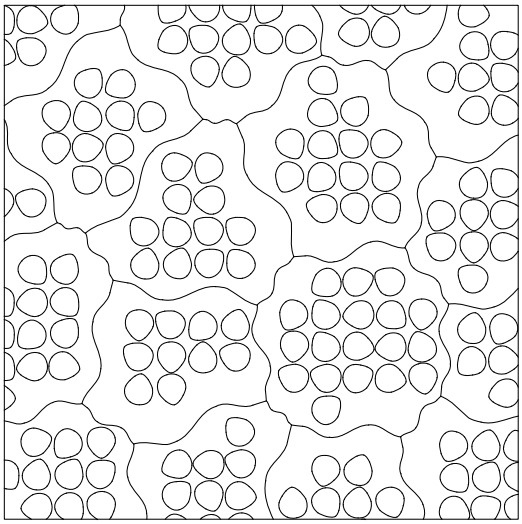
Script edition (b) switch partitions 52 sec

```
def overview_var1():
    size = 2500
    blob = ImportSVG("data/blob.svg")
    zig  = ImportSVG("data/zig.svg")

    # Mapper: place a blob in each 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: replace each 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)

    def create_blob_tex():
        # Grid partition
        theta     = 0
        width     = size/4
        lines1    = StripesProperties(theta,
            width)
        lines2    = StripesProperties(theta+pi
            /2,width)
        init_tex = GridPartition(lines1,lines2,
            KEEP_OUTSIDE)

        # Mapping operators
        return MapToFaces(map_blob_to,init_tex)

    # Mapper: generate a texture in each face
    def create_zig_tex(face):
        # Uniform partition
        props     = IrregularProperties(11/(
            BBoxWidth(face)*BBoxHeight(face)))
        init_tex = UniformPartition(props,
            CROP_ADD_BOUNDARY)

        # Mapping operator
        texture2 = MapToEdges(map_curve_to,
            init_tex)
        return texture2(face)

    init_tex  = create_blob_tex()
    final_tex = MapToFaces(create_zig_tex,
        init_tex)

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



Script edition (c) switch arrangements 12 sec

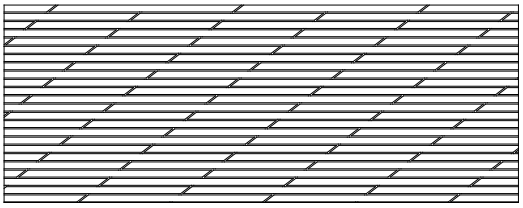
```
def overview_var2():
    size = 2500
    blob = ImportSVG("data/blob.svg")
    zig  = ImportSVG("data/zig.svg")

    # Mapper: place a blob in each 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: replace each 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)

    def create_blob_tex(face):
        # Grid partition
        theta     = 0 #Random(face,0,2*pi,0)
        width     = size/16
        lines1    = StripesProperties(theta,
            width)
        lines2    = StripesProperties(theta+pi
            /2,width)
        init_tex = GridPartition(lines1,lines2,
            KEEP_INSIDE)

        # Mapping operators
        return MapToFaces(map_blob_to,init_tex)
            (face)

    # Mapper: generate a texture in each face
    def create_zig_tex():
        # Uniform partition
        props     = IrregularProperties(11/(size
            *size))
        init_tex = UniformPartition(props,
            KEEP_OUTSIDE)

        # Mapping operator
        return MapToEdges(map_curve_to,init_tex
            )

    init_tex  = create_zig_tex()
    final_tex = Union(init_tex,MapToFaces(
        create_blob_tex,init_tex))

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



Script edition (d) vary mappings 56 sec

```
def overview_var3():
    size    = 2500
    flower = ImportSVG("data/flower.svg")
    leaf    = ImportSVG("data/leaf.svg")
    zig     = ImportSVG("data/zig7.svg")

    # Mapper: place a blob in each face
    def map_blob_to(face):
        new_flower = Scale(Rotate(flower,Random
            (face,0,2*pi,0)),Random(face
            ,0.7,1,1))
        new_leaf = Scale(Rotate(leaf,Random(
            face,0,2*pi,2)),Random(face
            ,0.7,1,3))
        if Random(face,0,1,4)>0.5:
            return MatchPoint(new_flower,
                BBoxCenter(new_flower),Centroid(
                face))
        else:
            return MatchPoint(new_leaf,
                BBoxCenter(new_leaf),Centroid(
                face))

    # Mapper: replace each 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)

    def create_blob_tex(face):
        # Grid partition
        theta     = 0
        width     = size/16
        lines1    = StripesProperties(theta,
            width)
        lines2    = StripesProperties(theta+pi
            /2,width)
        init_tex = GridPartition(lines1,lines2,
            KEEP_INSIDE)

        # Mapping operators
        return MapToFaces(map_blob_to,init_tex)
            (face)

    # Mapper: generate a texture in each face
    def create_zig_tex():
        # Uniform partition
        props     = IrregularProperties(11/(size
            *size))
        init_tex = UniformPartition(props,
            KEEP_OUTSIDE)

        # Mapping operator
        return MapToEdges(map_curve_to,init_tex
            )

    init_tex  = create_zig_tex()
    final_tex = Union(init_tex,MapToFaces(
        create_blob_tex,init_tex))

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

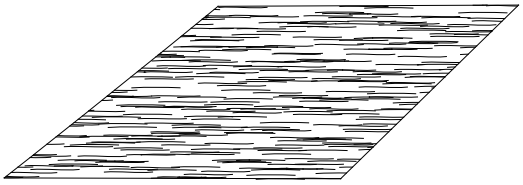
Parquetry 20 sec



```
def parquetry():
    # Label filtering: transforms a grid
        partition into
    # parquetry slats with space in between
    def parquetry_structure(e):
        faces = Faces()
        faces.push_back(LeftFace(e))
        faces.push_back(RightFace(e))
        if not LeftFace(e).is_bounded_face()\
        or not RightFace(e).is_bounded_face():
            return ToCurve(e)
        if HasTag(faces, "v1") and HasTag(faces
            , "v2") and HasTag(faces, "h1")\
        or HasTag(faces, "v2") and HasTag(faces
            , "v3") and HasTag(faces, "h1")\
        or HasTag(faces, "v3") and HasTag(faces
            , "v4") and HasTag(faces, "h1")\
        or HasTag(faces, "v4") and HasTag(faces
            , "v1") and HasTag(faces, "h1")\
        or HasTag(faces, "v1") and HasTag(faces
            , "v2") and HasTag(faces, "h3")\
        or HasTag(faces, "v2") and HasTag(faces
            , "v3") and HasTag(faces, "h3")\
        or HasTag(faces, "v3") and HasTag(faces
            , "v4") and HasTag(faces, "h3")\
        or HasTag(faces, "v4") and HasTag(faces
            , "v1") and HasTag(faces, "h3")\
        or HasTag(faces, "h1") and HasTag(faces
            , "h2") and HasTag(faces, "v1")\
        or HasTag(faces, "h2") and HasTag(faces
            , "h3") and HasTag(faces, "v1")\
        or HasTag(faces, "h3") and HasTag(faces
            , "h4") and HasTag(faces, "v3")\
        or HasTag(faces, "h4") and HasTag(faces
            , "h1") and HasTag(faces, "v3")\
        or HasTag(faces, "v2") and HasTag(faces
            , "v3") and HasTag(faces, "h2")\
        or HasTag(faces, "v3") and HasTag(faces
            , "v4") and HasTag(faces, "h2")\
        or HasTag(faces, "v4") and HasTag(faces
            , "v1") and HasTag(faces, "h4")\
        or HasTag(faces, "v1") and HasTag(faces
            , "v2") and HasTag(faces, "h4"):
            return Nothing()
        return ToCurve(e)

    # Grid partition
    my_scale = 50
    interstice_size = 0.2*my_scale
    lines1    = StripesProperties(0.0,1*
        my_scale,interstice_size,1*my_scale,
        interstice_size)
    SetFaceLabels(lines1, "h1", "h2", "h3", "
        h4")
    lines2    = StripesProperties(pi/6+pi/32+pi
        /100,8*my_scale, interstice_size,8*
        my_scale, interstice_size)
    SetFaceLabels(lines2, "v1", "v2", "v3", "
        v4")
    part = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)
    # Refinement
    struct = MapToEdges(parquetry_structure,
        part)
    # Export
    frame = ImportSVG("data/new_carpet_outline
        .svg")
    ExportSVGinDomain(struct, frame)
```

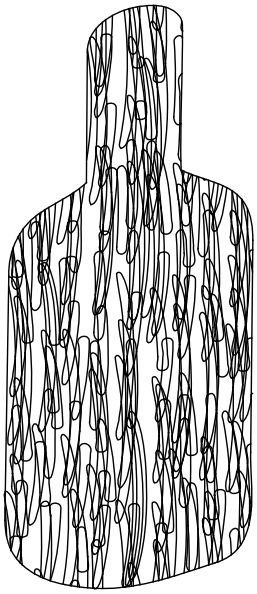
Table 45 sec

```
def table():
    # Import elements
    hatch1 = ImportSVG("data/smoothhatch1.svg")
    hatch2 = ImportSVG("data/smoothhatch2.svg")
    hatch3 = ImportSVG("data/smoothhatch3.svg")
    hatches = [hatch1, hatch2, hatch3]
    # Parameter for modifying hatch orientation
    side_angle = 0

    # Mapper: place a hatch on a vertex
    def place_hatch(v):
        randx = Random(v, -20, 20, 1)
        randy = Random(v, -80, 80, 2)
        loc = Location(v) + Point(randx, randy)
        elem0 = hatches[floor(Random(0,len(hatches)))]
        elem  = Rotate(elem0, side_angle)
        return MatchPoint(Scale(elem, 1.0), BBoxCenter(elem),loc)

    # Grid partition
    linesA   = StripesProperties(side_angle + pi/3.0,60)
    linesB= StripesProperties(side_angle + 2.0*pi/3.0, 150)
    part = GridPartition(linesA, linesB, KEEP_OUTSIDE)
    # Apply the mapper
    texture = MapToVertices(place_hatch, part)
    # Fill the outline
    frame = ImportSVG("data/outline_table_topside.svg")
    ExportSVGinDomain(texture,frame)
```



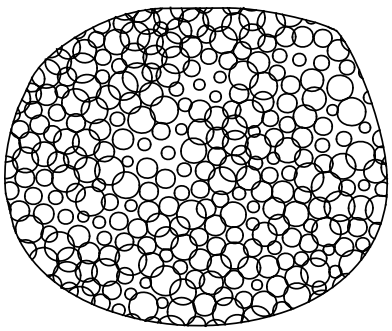
Bottle 11 sec

```
    frame = ImportSVG("data/outline_bottle.svg")
    bubble1 = ImportSVG("data/bubble1.svg")
    bubble2 = ImportSVG("data/bubble2.svg")
    bubble3 = ImportSVG("data/bubble3.svg")
    bubble4 = ImportSVG("data/bubble4.svg")
    bubbles = [bubble1, bubble2, bubble3, bubble4]
    side_angle = pi / 2.0

    # Mapper: place a bubble on a vertex
    def place_bubble(v):
        randx = Random(v, -20, 20, 1)
        randy = Random(v, -40, 40, 2)
        loc = Location(v) + Point(randx, randy)
        elem0 = bubbles[floor(Random(0,len(bubbles)))]
        elem  = Rotate(elem0, side_angle)
        return MatchPoint(Scale(elem, 1.0), BBoxCenter(elem),loc)

    # Grid partition
    linesA   = StripesProperties(side_angle + pi/3.0,60)
    linesB= StripesProperties(side_angle +2* pi/3.0, 60)
    part = GridPartition(linesA, linesB, KEEP_OUTSIDE)
    # Apply the mapper
    texture = MapToVertices(place_bubble, part)
    # Fill the outline
    ExportSVGinDomain(texture, frame)
```



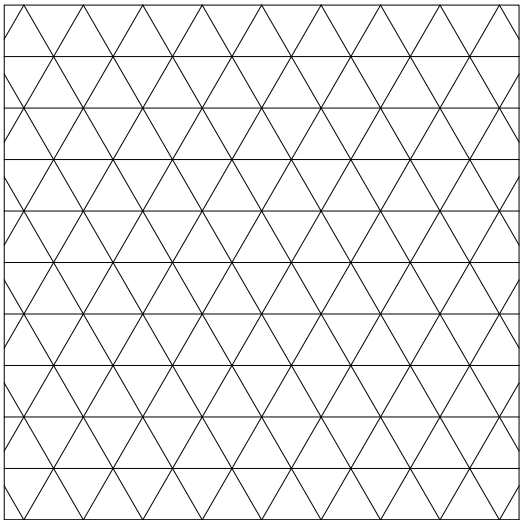
Teapot 22 sec

```
def teapot():
    density = 3.0e-4
    circle = ImportSVG("data/circle.svg")

    # Mapper: place a circle in each face
    def face_to_circle(face):
        src_p = BBoxCenter(circle)
        dst_p = Centroid(face)
        return Scale(MatchPoint(circle,src_p, dst_p),Random(face,0.05,0.15,0))

    # Uniform partition with given density
    props  = IrregularProperties(density)
    part   = UniformPartition(props, KEEP_OUTSIDE)
    # Apply the mapper
    tex3 = MapToFaces(face_to_circle,part)
    # Fill the outline
    frame = ImportSVG("data/outline_teapot.svg")
    ExportSVGinDomain(tex3, frame)
```



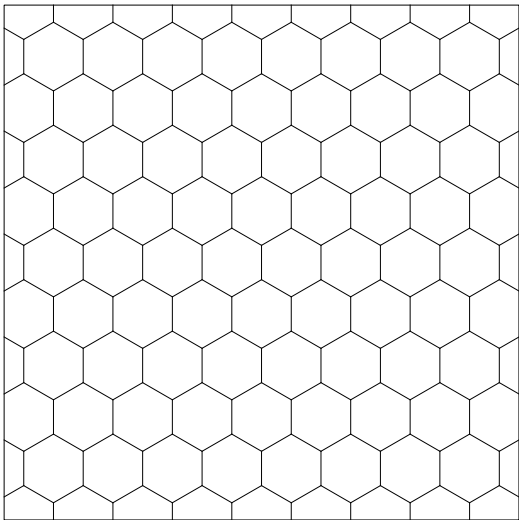
Triangles 4 sec

```
# Mapper: splits in two a unique face from a grid partition
def split(face):
    are_incident = lambda e,f: f == LeftFace(e) or f == RightFace(e)
    v1 = 0
    v2 = 0
    for v in IncidentVertices(face):
        score_source = 0
        score_target = 0
        for e in IncidentEdges(v):
            if v == SourceVertex(e) and are_incident(e, face):
                score_source += 1
            if v == TargetVertex(e) and are_incident(e, face):
                score_target += 1
        if score_source == 2:
            v1 = v
        if score_target == 2:
            v2 = v
    if v1 == 0
        return Nothing()
    seg = ImportSVG("segment.svg")
    start_seg = PointLabeled(seg, "start")
    end_seg = PointLabeled(seg, "end")
    return MatchPoints(seg, start_seg, end_seg, Location(v1), Location(v2))

def example_triangles():
    # Grid partition
    props1 = StripesProperties(pi/3.0, 200)
    props2 = StripesProperties(-pi/3.0, 200)
    grid = GridPartition(props1, props2, KEEP_OUTSIDE)
    # Apply mapper and merge the result with the original grid
    triangles = Union(grid, MapToFaces(split, grid))
    # Export final texture
    ExportSVG(triangles, 2000)
```



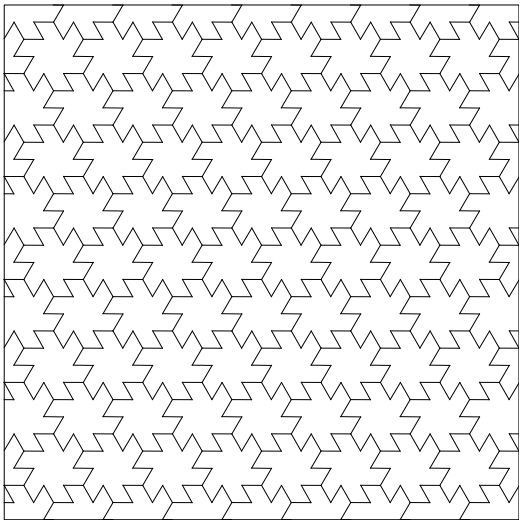
Bee hive 5 sec

```
# Mapper: returns a segment linking the centroids of the faces incident to e
def dual(e):
    if e.is_outside():
        return Nothing()
    seg = ImportSVG("segment.svg")
    start_seg = PointLabeled(seg, "start")
    end_seg = PointLabeled(seg, "end")
    return MatchPoints(seg, start_seg, end_seg, Centroid(LeftFace(e)), Centroid(RightFace(e)))
def example_hexa():
    # Grid partition
    props1 = StripesProperties(pi/3.0, 200)
    props2 = StripesProperties(-pi/3.0, 200)
    grid = GridPartition(props1, props2, KEEP_OUTSIDE)
    # Split faces (from example ''Triangles'')
    triangles = Union(grid, MapToFaces(split, grid))
    # Take the duals of all edges
    hexa = MapToEdges(dual, triangles)
    # Export final tetxure
    ExportSVG(hexa, 2000)
```



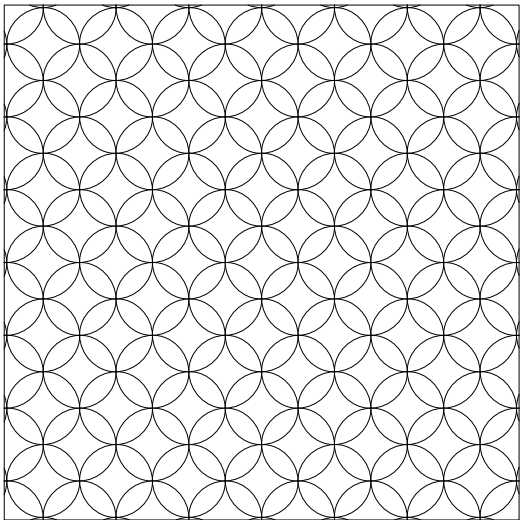
Wallpaper group P6 48 sec

```
def example_p6():
    # Import SVG element
    line = ImportSVG("data/p6_elt.svg")
    # Mapper: replaces an edge by the imported SVG element
    def line_to_curve(edge):
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)
    props1 = StripesProperties(pi/3.0, 200)
    props2 = StripesProperties(-pi/3.0, 200)
    grid = GridPartition(props1, props2, KEEP_OUTSIDE)
    # Split faces (from example ''Triangles'')
    triangles = Union(grid, MapToFaces(split, grid))
    # Take duals of all edges (from example ''Bee hive'')
    hexa = MapToEdges(dual, triangles)
    # Replace edges by Z-shaped curves
    p6 = MapToEdges(line_to_curve, hexa)
    # Export final texture
    ExportSVG(p6, 2000)
```



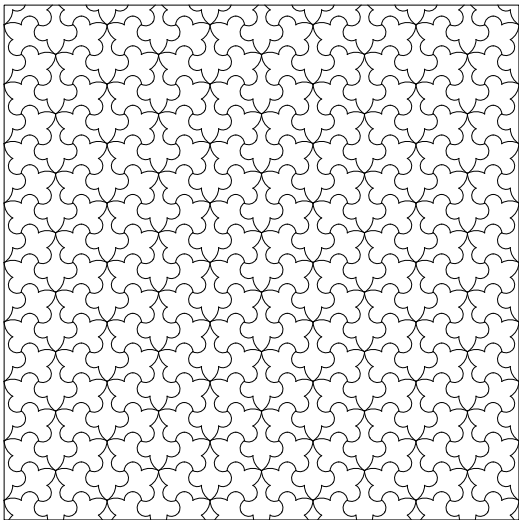
Wallpaper group P4M 24 sec

```
def example_p4m():
    # Import SVG element
    line = ImportSVG("data/p4m_elt.svg")
    # Mapper: replaces an edge by the
        imported SVG element
    def line_to_curve(edge):
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)
    # Mapper: replaces an edge by a rotation
        of the imported SVG element
    def line_to_curve_inv(edge):
        dst_c = PointLabeled(line,"start")
        src_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)
    # Grid partition
    props1 = StripesProperties(pi/4.0, 200)
    props2 = StripesProperties(-pi/4.0, 200)
    grid = GridPartition(props1, props2,
        KEEP_OUTSIDE)
    # Take the results of both mappers
    p4m = Union(MapToEdges(line_to_curve,
        grid), MapToEdges(line_to_curve_inv,
        grid))
    # Export final texture
    ExportSVG(p4m, 2000)
```



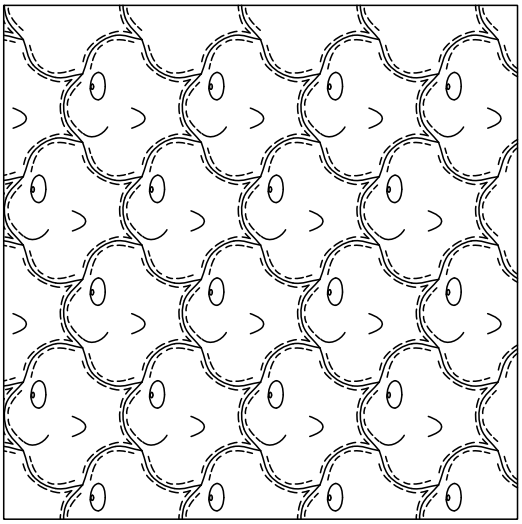
Wallpaper group P31M 50 sec

```
def example_p31m():
    line0 = ImportSVG("data/p31m_elt.svg")
    line0_sym = ImportSVG("data/p31m_elt_sym.svg")
    # Mapper: subdivides a triangle face in three pseudo-triangles
    def subdivide(face):
        out = Nothing()
        line = line0
        nb_left = 0
        for e in IncidentEdges(face):
            if face == LeftFace(e):
                nb_left = nb_left + 1
        if nb_left > 1:
            line = line0_sym
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        dst_v = Centroid(face)
        for v in IncidentVertices(face):
            src_v = Location(v)
            curve = MatchPoints(line,src_c,dst_c,src_v,dst_v)
            out = Append(out, curve)
        return out
    # Grid partition
    props1 = StripesProperties(pi/6.0, 200)
    props2 = StripesProperties(pi-pi/6.0, 200)
    grid = GridPartition(props1, props2, KEEP_OUTSIDE)
    # Split faces (from example ``Triangles'')
    triangles = Union(grid, MapToFaces(split, grid))
    # Subdivide triangle faces
    p31m = MapToFaces(subdivide, triangles)
    # Export final texture
    ExportSVG(p31m, 2000)
```



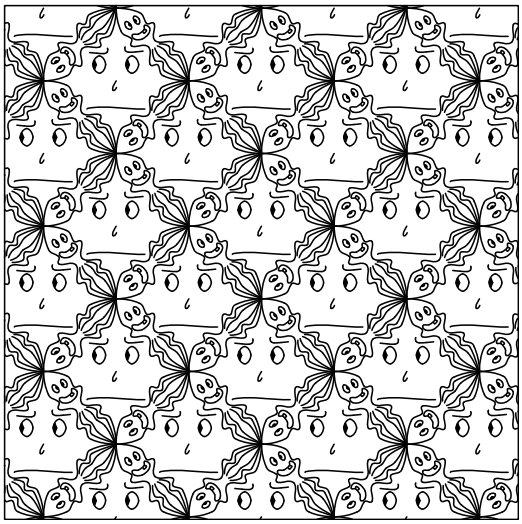
Escher P6 38 sec

```
def example_p6_escher():
    # Imported SVG elements
    line = ImportSVG("data/p6_elt_complex3.
        svg")
    center_elt = ImportSVG("data/
        p6_elt_complex3_center.svg")
    # Mapper: replaces an edge by a piece of
        a fish outline
    def line_to_curve(edge):
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)
    # Mapper: places a fish interior inside a
         face
    def face_to_center_elt(f):
        centers_center = PointLabeled(
            center_elt,"end") + Point(20,0)
        return MatchPoint(center_elt,
            centers_center, Centroid(f))
    # Grid partition
    props1 = StripesProperties(pi/3.0, 200)
    props2 = StripesProperties(-pi/3.0, 200)
    grid = GridPartition(props1, props2,
        KEEP_OUTSIDE)
    # Split faces (from example ``Triangles
        '')
    triangles = Union(grid, MapToFaces(split,
         grid))
    # Take duals of all edges (from example
        ``Bee hive'')
    hexa = MapToEdges(dual, triangles)
    # Replace edges by fish outlines
    p6 = MapToEdges(line_to_curve, hexa)
    # Place fish interiors
    p6_center = MapToFaces(face_to_center_elt
        , hexa)
    # Export final texture
    final = Union(p6, p6_center)
    ExportSVG(final, 1000)
```



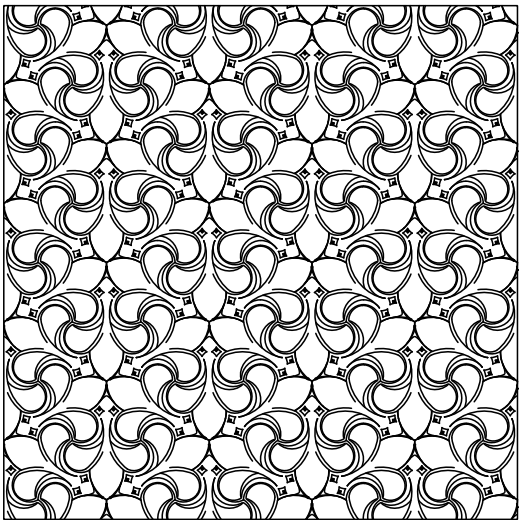
Escher P4M 43 sec

```
def example_p4m_escher():
    # Imported SVG elements
    octopus = ImportSVG("data/
        p4m_elt_complex1.svg")
    guy = ImportSVG("data/p4m_elt_complex2.
        svg")
    # Mapper: places an octopus on an edge
    def line_to_octopus(edge):
        src_c = PointLabeled(octopus,"start")
        dst_c = PointLabeled(octopus,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(octopus,src_c,
            dst_c,src_v,dst_v)
    # Mapper: places a guy on a face's center
    def face_to_guy(face):
        center_guy = PointLabeled(guy,"start"
            )
        c = Centroid(face)
        return MatchPoint(guy,center_guy, c)
    # Grid partition
    props1 = StripesProperties(pi/4.0, 200)
    props2 = StripesProperties(-pi/4.0, 200)
    grid = GridPartition(props1, props2,
        KEEP_OUTSIDE)
    # Merge octopuses and guys
    p4m = Union(MapToEdges(line_to_octopus,
        grid), MapToFaces(face_to_guy, grid))
    # Export final texture
    ExportSVG(p4m, 1000)
```



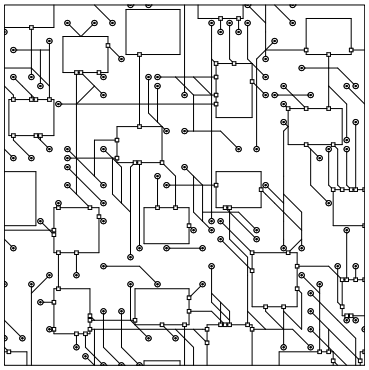
Escher P31M 35 sec

```
def example_p31m_escher():
    # Imported SVG elements
    line0 = ImportSVG("data/p31m_elt_complex.
        svg")
    line0_sym = ImportSVG("data/
        p31m_elt_complex_sym.svg")
    # Mapper: subdivides each triangle face
        into three pseudo-triangles
    def subdivide(face):
        out = Nothing()
        line = line0
        nb_left = 0
        for e in IncidentEdges(face):
            if face == LeftFace(e):
                nb_left = nb_left + 1
        if nb_left > 1:
            line = line0_sym
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        dst_v = Centroid(face)
        for v in IncidentVertices(face):
            src_v = Location(v)
            curve = MatchPoints(line,src_c,
                dst_c,src_v,dst_v)
            out = Append(out, curve)
        return out
    # Grid partition
    props1 = StripesProperties(pi/6.0, 200)
    props2 = StripesProperties(pi-pi/6.0,
        200)
    grid = GridPartition(props1, props2,
        KEEP_OUTSIDE)
    # Split faces (from example ``Triangles
        '')
    triangles = Union(grid, MapToFaces(split,
         grid))
    # Subdivide resulting triangle faces
    p31m = MapToFaces(subdivide, triangles)
    # Export final texture
    ExportSVG(p31m, 1000)
```



Microprocessor 54 sec

```
def processor():
    # Global parameters
    size = 2000
    step = 50

    # ———————— Level 1 ————————
    densities = [0.002, 0.00175, 0.00125, 0.00225, 0.001]
    # Paths and density indexes
    elts_densities = [["net1.svg",0],["net2.svg",0],["net3.
        svg",1],["net4.svg",0],["net5.svg",1],["net6.svg"
        ,1],["net7.svg",2],["net8.svg",1],["net9.svg",0],["
        net10.svg",3],["net11.svg",3],["net12.svg",4]]

    # Define a new mapper for each 'elt'
    def mapper(elt):
        def new_face_mapper(face):
            pt = Centroid(face)
            # Clamp position for regular silicium look
            clamp = lambda x: step * floor(x/step)
            pt = Point(clamp(pt.x()), clamp(pt.y()))
            return MatchPoint(elt, PointLabeled(elt, "start"
                ), pt)
        return new_face_mapper

    # dst: distributes randomly elt e with density d
    dst = lambda e, d: MapToFaces(mapper(e), RandomPartition
        (IrregularProperties(d), KEEP_OUTSIDE))

    # Distribute all elements
    level1 = lambda f: Nothing()
    for e_d in elts_densities:
        d = densities[e_d[1]]
        level1 = Merge(level1,dst(ImportSVG(e_d[0]),d))

    # ———————— Level 2 ————————
    ships = ["ship1.svg","ship2.svg","ship3.svg","ship4.svg"
        ,"ship5.svg","ship6.svg"]

    # Mapper: generates and places a ship on the face
    def face_to_ship(face):
        ship = ImportSVG(ships[floor(Random(face,0,6,0))])
        ship = Scale(ship, floor(Random(face,0,4,0)))
        pt = Centroid(face)
        # Clamp position for regular silicium look
        clamp = lambda x: step * floor(x/step)
        pt = Point(clamp(pt.x()), clamp(pt.y()))
        res = MatchPoint(ship, BBoxCenter(ship), pt)
        return res

    props = IrregularProperties(8/(2000*2000))
    part_ships = UniformPartition(props, KEEP_OUTSIDE)
    level2 = MapToFaces(face_to_ship, part_ships)

    # ———————— Final texture ————————
    c = ImportSVG("data/connector.svg")
    j = ImportSVG("data/joint.svg")

    # Mapper: generates connectors and soldered joints
    def make_cj(v):
        # Connectors at contacts between level1 and level2
        if len(IncidentEdges(v)) == 3 and HasLabel(
            IncidentFaces(v),"ship"):
            return MatchPoint(c, BBoxCenter(c), Location(v))
        # Generate joints at junction ends
        elif len(IncidentEdges(v)) == 1:
            return MatchPoint(j, BBoxCenter(j), Location(v))
        return Nothing()

    lvls_1_2 = Outside(level1,level2,CROP_ADD_BOUNDARY)
    cjs = MapToVertices(make_cj, lvls_1_2)
    final_tex = Outside(lvls_1_2, cjs, CROP_ADD_BOUNDARY)
    ExportSVG(final_tex, size)
```



APPENDIX

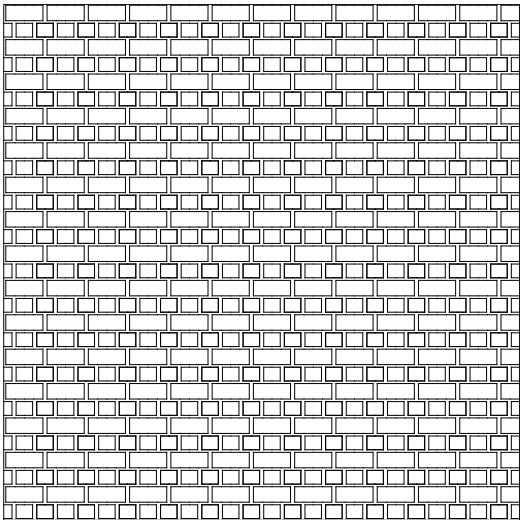
B

EXAMPLE SCRIPTS USING TILED PLANAR MAPS

In this appendix we show the scripts used to synthesize all the arrangements in Chapter 4.

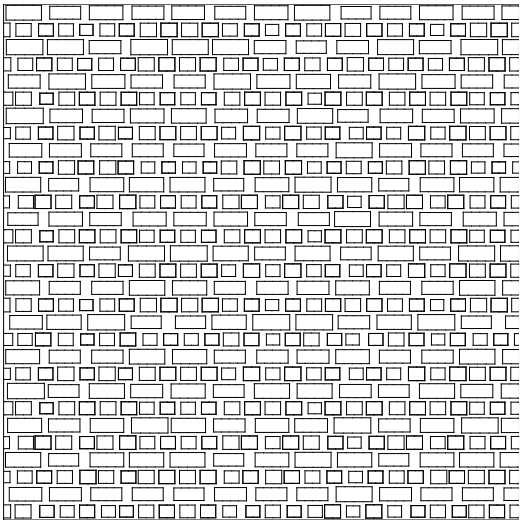
Brickwall 2 sec

```
def brickwall_tpm():
    size = 6000
    # Mapper: remove some vertical edges
    def filter_brickwall(edge):
        if HasLabel(edge, "h1") or HasLabel(
            edge, "h2") or HasLabel(edge, "h3
            ") or HasLabel(edge, "h4"):
            return ToCurve(edge)
        if HasLabel(IncidentEdges(
            TargetVertex(edge)), "h1") or
            HasLabel(IncidentEdges(
            TargetVertex(edge)), "h3"):
            return ToCurve(edge) if HasLabel(
                edge, "v1") else Nothing()
        else:
            return ToCurve(edge) if (HasLabel
                (edge, "v2") or HasLabel(edge
                , "v4")) else Nothing()
    # Mapper: scales down a face's contour
    def shrink_face(face):
        return Scale(Contour(face),0.9) if
            HasLabel(IncidentEdges(face), "v1
            ") else Scale(Contour(face),0.8)
    # Grid partition
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,120)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2","v3","v4")
    grid_tpm = TPMGridPartition(lines1,lines2
        ,KEEP_OUTSIDE)
    # Mapping and export
    tpm = MapToTPMedges(filter_brickwall,
        grid_tpm)
    tpm2 = MapToTPMfaces(shrink_face, tpm)
    ExportTPMtoSVG(tpm2, size)
```



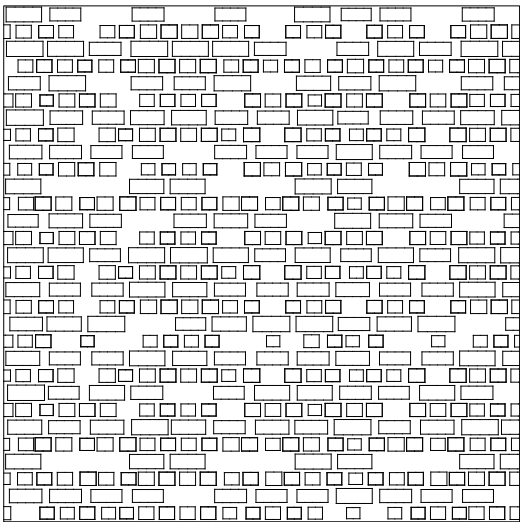
Noisy Brickwall 2 sec

```
def noisy_brickwall_tpm():
    size = 6000
    # Mapper: remove some vertical edges
    def filter_brickwall(edge):
        if HasLabel(edge, "h1") or HasLabel(
            edge, "h2") or HasLabel(edge, "h3
            ") or HasLabel(edge, "h4"):
            return ToCurve(edge)
        if HasLabel(IncidentEdges(
            TargetVertex(edge)), "h1") or
            HasLabel(IncidentEdges(
            TargetVertex(edge)), "h3"):
            return ToCurve(edge) if HasLabel(
                edge, "v1") else Nothing()
        else:
            return ToCurve(edge) if (HasLabel
                (edge, "v2") or HasLabel(edge
                , "v4")) else Nothing()
    # Mapper: scales down a face's contour
    def shrink_face(face):
        return Scale(Contour(face),0.9) if
            HasLabel(IncidentEdges(face), "v1
            ") else Scale(Contour(face),0.8)
    # Mapper: scales and translates randomly
        a face's contour
    def jitter_face(face):
        return Translate(Scale(Contour(face),
             Random(face, 0.8, 1.0, 1)),
            Point(Random(face, -24, 24, 0),
            0))
    # Grid partition
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,120)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2","v3","v4")
    grid_tpm = TPMGridPartition(lines1,lines2
        ,KEEP_OUTSIDE)
    # Mapping
    tpm = MapToTPMedges(filter_brickwall,
        grid_tpm)
    tpm2 = MapToTPMfaces(shrink_face, tpm)
    # Ghost mapping
    GhostMapToTPMfaces(jitter_face, tpm2)
    # Export
    ExportTPMtoSVG(tpm2, size)
```



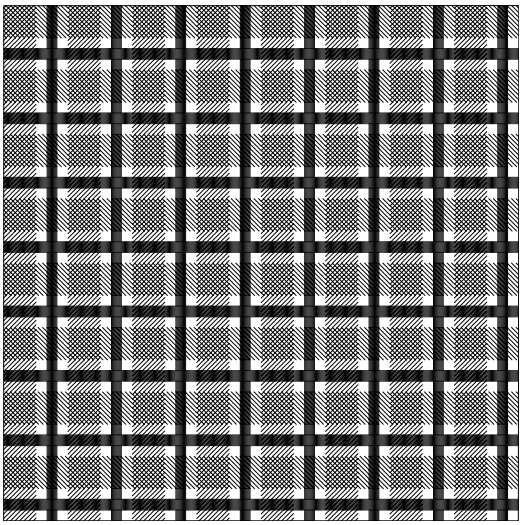
Noisy Brickwall 2 2 sec

```
def noisy_brickwall_tpm_2():
    size = 6000
    # Mapper: remove some vertical edges
    def filter_brickwall(edge):
        if HasLabel(edge, "h1") or HasLabel(
            edge, "h2") or HasLabel(edge, "h3
            ") or HasLabel(edge, "h4"):
            return ToCurve(edge)
        if HasLabel(IncidentEdges(
            TargetVertex(edge)), "h1") or
            HasLabel(IncidentEdges(
            TargetVertex(edge)), "h3"):
            return ToCurve(edge) if HasLabel(
                edge, "v1") else Nothing()
        else:
            return ToCurve(edge) if (HasLabel
                (edge, "v2") or HasLabel(edge
                , "v4")) else Nothing()
    # Mapper: scales down a face's contour
    def shrink_face(face):
        return Scale(Contour(face),0.9) if
            HasLabel(IncidentEdges(face), "v1
            ") else Scale(Contour(face),0.8)
    # Mapper: deletes randomly a face, or
        scales and translates it randomly
    def jitter_face(face):
        if Random(face, 0,1,2) < 0.1:
            return Nothing()
        return Translate(Scale(Contour(face),
             Random(face, 0.8, 1.0, 1)),
            Point(Random(face, -24, 24, 0),
            0))
    # Grid partition
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,120)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2","v3","v4")
    grid_tpm = TPMGridPartition(lines1,lines2
        ,KEEP_OUTSIDE)
    # Mapping
    tpm = MapToTPMedges(filter_brickwall,
        grid_tpm)
    tpm2 = MapToTPMfaces(shrink_face, tpm)
    # Ghost mapping
    GhostMapToTPMfaces(jitter_face, tpm2)
    # Export
    ExportTPMtoSVG(tpm2, size)
```



Fabric 4 sec

```
ef fabric(): # Reproduction of teaser_h
    size    = 6000
    # Stripe partition
    lines1 = StripesProperties(0.0,300)
    SetFaceTags(lines1,"h1","h2")
    stripes1 =  StripesPartition(lines1)
    # Stripe partition
    lines2 = StripesProperties(pi/2,300)
    SetFaceLabels(lines2,"v1","v2")
    stripes2 =  StripesPartition(lines2)
    # Grid partition with labels
    lines3 = StripesProperties
        (0.0,100,100,100,300)
    SetFaceLabels(lines3,"h1","h2","h3","h4")
    lines4 = StripesProperties(pi
        /2,100,100,100,300)
    SetFaceLabels(lines4,"v1","v2","v3","v4")
    grid = GridPartition(lines3,lines4,
        KEEP_OUTSIDE)
    # Create a stripe partition with
        specified angle and width
    def hatch(angle,width):
        lines = StripesProperties(angle,width
            )
        return StripesPartition(lines)
    # Mapper: create -pi/4 stripes in each
        face
    def hatch_map_stripes1(f):
        angle = -pi/4
        if HasLabel(f,"h1"):
            return hatch(angle,30)(f)
        else:
            return Nothing()
    # Mapper: create pi/4 stripes in each
        face
    def hatch_map_stripes2(f):
        angle = pi/4
        if HasLabel(f,"v1"):
            return hatch(angle,30)(f)
        else:
            return Nothing()
    # Mapper: create pi/2 stripes in specific
         faces
    def hatch_map_grid(f):
        angle = pi/2
        if HasLabel(f,"h3"):
            return hatch(angle,10)(f)
        elif HasLabel(f,"v3"):
            return hatch(angle,10)(f)
        else:
            return Nothing()
    # Mapping operators
    tex1 = MapToTPMfaces(hatch_map_stripes1,
        stripes1)
    tex2 = MapToTPMfaces(hatch_map_stripes2,
        stripes2)
    tex4 = MapToTPMfaces(hatch_map_grid,grid)
    # Merging operators
    tex3 = TPMUnion(tex1,tex2)
    tex5 = TPMUnion(tex3,tex4)
    # Export
    ExportTPMtoSVG(tex5,size)
```



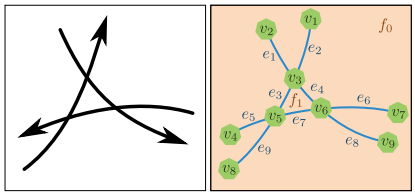
APPENDIX

C

USER STUDY TUTORIAL

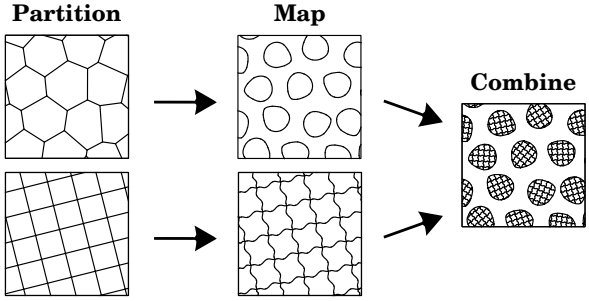
I. General Principle

In this tool textures will be represented as *planar maps*. A planar map is a set of curves whose intersections and enclosed faces are computed. Therefore, you can manipulate three kinds of *cells* in the planar map: *vertices*, *edges* and *faces*. Each of these cells knows all of its neighbors, also called *incident cells*.



Curves Planar map

All textures begin with a *partition*, then they are refined using *mappers*. When two textures are created this way, they can be *combined*:



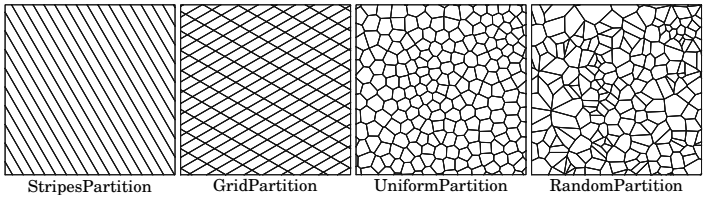
II. Partitions

Partitions act as construction lines which define the broad-scale organization of the texture. You can choose between four starting partitions:

— 2 regular ones (stripes, grid)

— 2 irregular ones (uniform, random)

Partitions are cut at the border of the domain using a border management option (see Additional document No 2). You will be able to manipulate the cells of partitions using mappers.

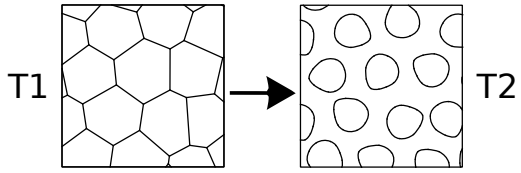


Practice: execute partitions scripts and modify their parameters

```
def test_StripesPartition():
    lines1    = StripesProperties(- pi/3.0,110)
    part = StripesPartition(lines1)
    ExportSVG(part,2000)

def test_GridPartition():
    lines1    = StripesProperties(pi/6.0,110)
    lines2    = StripesProperties(pi/2.0+pi/3.0,200)
    part = GridPartition(lines1,lines2,CROP_ADD_BOUNDARY)
    ExportSVG(part,2000)

def test_UniformPartition():
    props = IrregularProperties(1 / 22000)
    part = UniformPartition(props, CROP_ADD_BOUNDARY)
    ExportSVG(part,2000)

def test_RandomPartition():
    props = IrregularProperties(1 / 22000)
    part = RandomPartition(props, CROP_ADD_BOUNDARY)
    ExportSVG(part,2000)
```

III. Mappers

Mappers are the operators that give you the most freedom in this tool. Actually you write yourself your mappers so that they do exactly what you want them to do. In practice, a mapper is a kernel function that draws a new element from one cell of the planar map. For instance, the following mapper is a function that draws a randomly-rotated blob shape on a given face:




```
def K ( face ) :
    blob = ImportSVG ( "data/blob.svg" )
    new_blob  = Rotate(blob,Random(face,0,2*pi,0))
    return  MatchPoint(new_blob,BBoxCenter(new_blob),Centroid(face))
T2 = MapToFaces ( K, T1 )
```

Once you wrote the mapper, you have to wrap it inside a mapping operator which applies your mapper on every cell of the planar map. Here, this mapping operator is “MapToFaces” because the mapper deals with faces. There are also a “MapToEdges” and a “MapToVertices” for the two other kinds of cells. This mapping operator ensures that your mapper has an homogeneous effect all over the texture, thus preserving a repetitive, predictable look.
You can write almost anything inside a mapper. At the end of this document there is an API of built-in functions you can use, but you can develop your own. All you have to do is observing the three following rules:

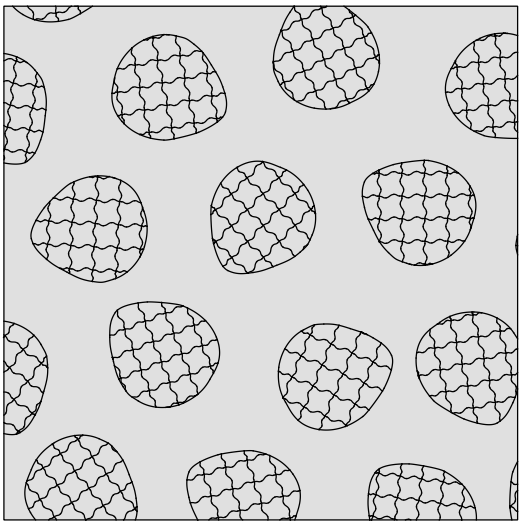
— Always stay in a **bounded neighborhood** around the current cell. For instance, you must not use neighborhood functions for writing a loop that travels along the texture until reaching the border. This would yield non-homogeneous effects and unwanted artifacts.
— **Do not modify global variables** in your mapper. If global variables change at each execution of your mapper, then the mapping operator will not be able to apply it with a homogeneous effect.
— **Do not use global, hard-coded coordinates**. For example if your mapper depends on the position (255, 42) then it will never have a homogeneous effect all over the texture.

In practice, mappers can be used for modifying both geometry and connections, for randomizing the texture, etc. See the examples for more info.

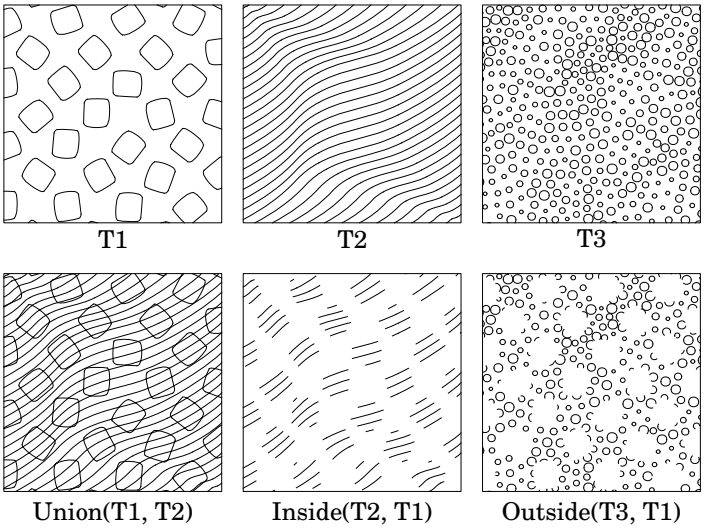
IV. Combining Textures

There are two ways of combining textures. The first one is to call the code of one of the textures in a mapper:

```
def test_Overview():
    size = 2000
    blob = Scale(ImportSVG("data/blob.svg")
        ,0.6)
    zig  = ImportSVG("data/zig.svg")

    # Mapper: place a blob in each 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: replace each 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: generate a texture in each face
    def map_texture_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_tex = GridPartition(lines1,lines2,
            CROP_ADD_BOUNDARY)

        # Mapping operator
        texture2 = MapToEdges(map_curve_to,
            init_tex)
        return texture2(face)

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

    # Mapping operators
    blob_tex  = MapToFaces(map_blob_to,
        init_tex)
    final_tex = MapToFaces(map_texture_to,
        blob_tex)

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

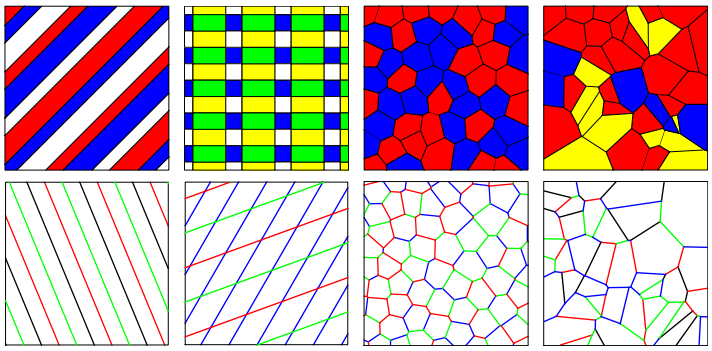


You can also combine two textures using either Union, Inside or Outside:



V. Labels

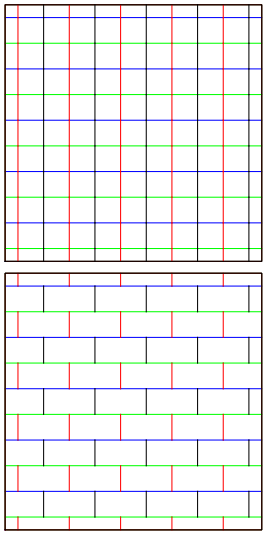
You can add additional information to the planar map's cells. In particular you can add *labels*, which can then be used in mappers for varying effects.
In StripesPartition labels are defined periodically: "Red, blue, red, blue, ..." for instance. It is the same in GridPartition, except with two dimensions. For UniformPartition and RandomPartition, the labels are added randomly. You can specify the chances of each label appearing.



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

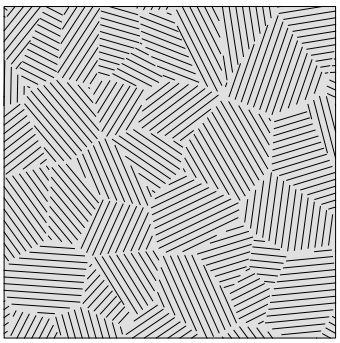
Labels can be used everytime you want to make your mappers' behavior variable. Here is an example with periodic refinement of the texture's topology:

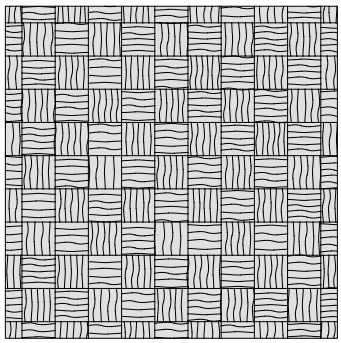
```
def brickwall():
    size    = 2000

    # Grid partition, including edge labels
    lines1 = StripesProperties(0.0,size/16)
    lines2 = StripesProperties(pi/2.0,size/16)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    # Mapper: remove edges
    def grid_to_wall(edge):
        if((HasLabel(edge,"v1") and HasLabel(
            IncidentEdges(TargetVertex(edge)),"
            h1")) or
           (HasLabel(edge,"v2") and HasLabel(
               IncidentEdges(TargetVertex(edge)
               ),"h2"))):
            return Nothing()
        return ToCurve(edge)

    # Mapping operator
    wall_tex    = MapToEdges(grid_to_wall,
        grid_tex)

    # Final texture
    ExportSVG(wall_tex,size)
```



VI. Sandbox

Let's plug the components in the sandbox script (Additional document No III) together so as to make the two following textures:





Additional document No I. API

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

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

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

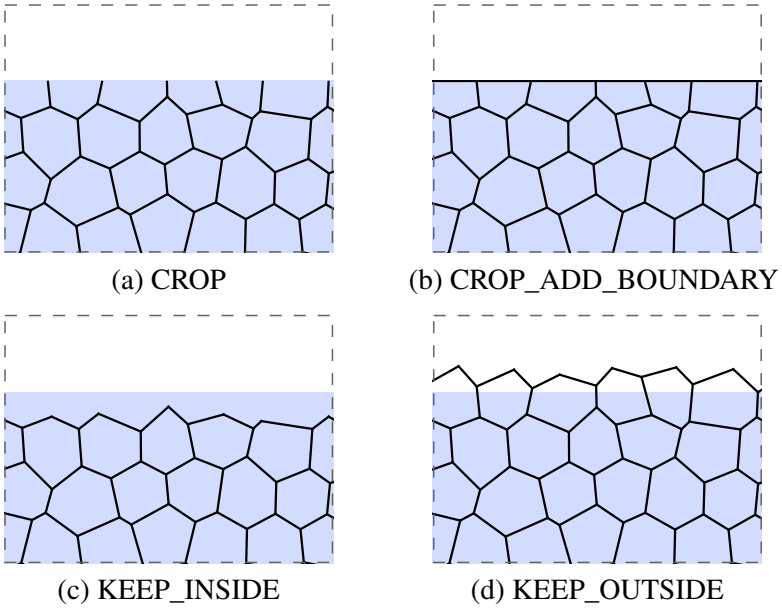
Merging operators

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

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

Additional document No II. Border Management.



(a) CROP (b) CROP_ADD_BOUNDARY

(c) KEEP_INSIDE (d) KEEP_OUTSIDE

Additional document No III-1. Sandbox script part 1

```
# Call your function here
def main():
    test1()

# Paste pieces of code in your function here
def test1():
    print("Hello!")

# Warning: partitions() will raise errors if called
# Grab pieces of code that you want and modify their parameters
def partitions():
    lines1    = StripesProperties(theta,width)
    lines2    = StripesProperties(theta+pi/2.0,width)
    grid_tex  = GridPartition(lines1,lines2,KEEP_OUTSIDE)

    props    = StripesProperties(theta,width)
    stripes  = StripesPartition(props)

    props   = IrregularProperties(density)
    part    = UniformPartition(props,KEEP_OUTSIDE)

    props1   = IrregularProperties(100/(size*size))
    tex1     = UniformPartition(props1,KEEP_OUTSIDE)

    props2   = IrregularProperties(1200/(size*size))
    tex2     = RandomPartition(props2,KEEP_OUTSIDE)

    props     = IrregularProperties(30/(size*size))
    init_tex  = RandomPartition(props,KEEP_OUTSIDE)

    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,KEEP_OUTSIDE)

# Warning: elements() will raise errors if called
# Grab pieces of code that you want and modify their parameters
def elements():
    circle = ImportSVG("data/circle.svg")
    line     = ImportSVG("data/line6.svg")
    square    = Scale(ImportSVG("data/square.svg"),0.5)
    wheel    = ImportSVG("data/wheel1.svg")
    stipple = ImportSVG("data/stipple1.svg")
```

Additional document No III-2. Sandbox script part 2

```
# Warning: mappers() will raise errors if called
# Grab pieces of code that you want and modify their parameters
def mappers():
    def face_to_square(face):
        return Scale(Rotate(MatchFace(square,face),Random(face,0.0,2.0*pi,1)),0.5)

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)

    def face_to_circle(face):
        src_p = BBoxCenter(circle)
        dst_p = Centroid(face)
        return Scale(MatchPoint(circle,src_p,dst_p),Random(face,0.05,0.15,0))

    def face_to_wheel(face):
        w = Scale(Rotate(wheel,Random(face,0,2*pi,0)),Random(face,0.8,1,1))
        return MatchPoint(w,BBoxCenter(w),Centroid(face))

    def face_to_stipples(face):
        s = Scale(Rotate(stipple,Random(face,0,2*pi,0)),Random(face,0.9,1,1))
        return MatchPoint(s,BBoxCenter(s),Centroid(face))

    def scale_map(face):
        return Scale(Contour(face),0.95)

    def hatch_map(face):
        angle = Random(face,0,2*pi,1)
        lines = StripesProperties(angle,40)
        return StripesPartition(lines)(face)

    def border_map(edge):
        if IsBoundary(edge):
            return Nothing()
        return ToCurve(edge)

    def face_to_stripes(face):
        width = BBoxWidth(face)/Random(face,4,6,0)
        theta = 0
        if((HasLabel(face,"h1") and HasLabel(face,"v1")) or
            (HasLabel(face,"h2") and HasLabel(face,"v2"))):
            theta = pi/2
        lines = StripesProperties(theta,width)
        return StripesPartition(lines)(face)
```

APPENDIX

D

USER STUDY RESULTS

Target textures

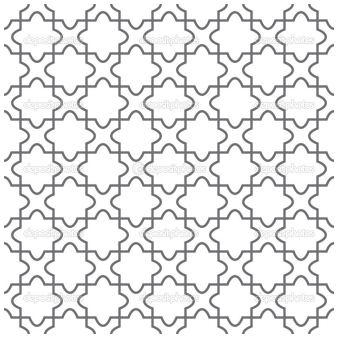


FIGURE D.1 – *"puzzle"*

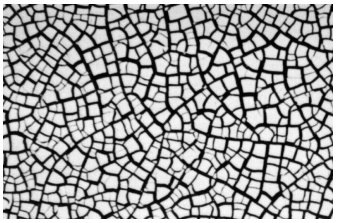


FIGURE D.2 – *"cracks"*

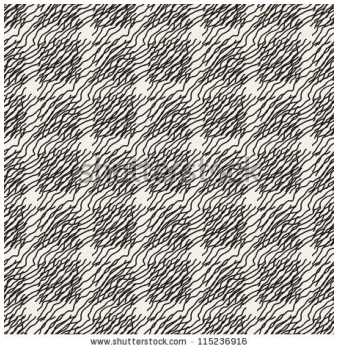
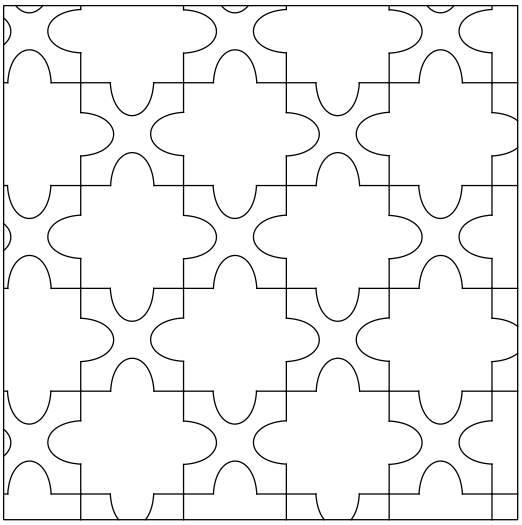


FIGURE D.3 – *"waves"*

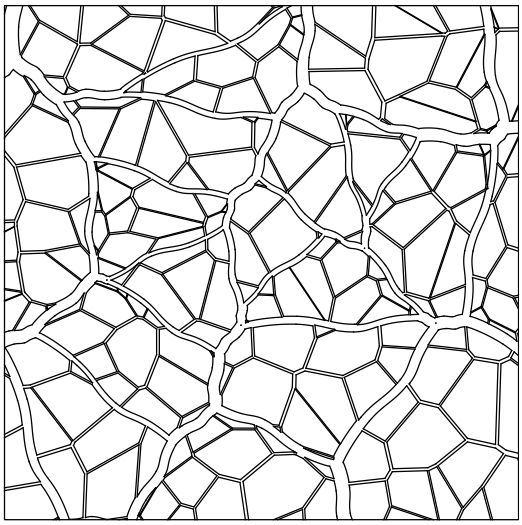
User 1 - puzzle. 21 lines, two operators (green), three executions. User satisfaction: 10/10

```
def test2():
    line = ImportSVG("data/estelle/bitonio.svg")

    def edge_to_bitonio(edge):
        dst_c = PointLabeled(line,"start")
        src_c = PointLabeled(line,"end")
        if (HasLabel(LeftFace(edge), "v1")
            and HasLabel(LeftFace(edge), "h1")):
            src_c = PointLabeled(line,"start")
            dst_c = PointLabeled(line,"end")
        elif (HasLabel(LeftFace(edge),"v2")
            and HasLabel(LeftFace(edge),"h2")):
            src_c = PointLabeled(line,"start")
            dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)

    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,KEEP_OUTSIDE)
    part = MapToEdges(edge_to_bitonio,grid_tex)

    ExportSVG(part, 1000)
```



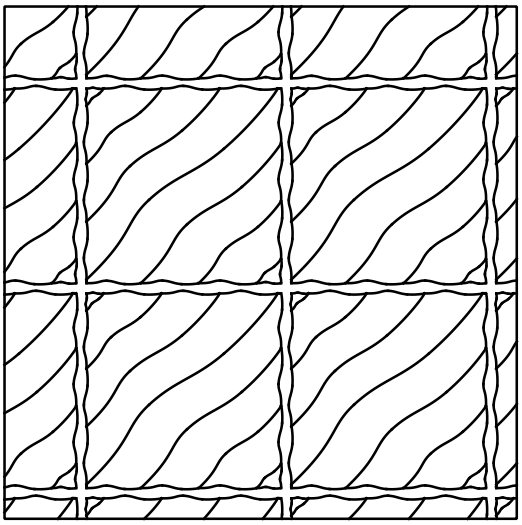
User 1 - cracks. 20 lines, six operators (green), three executions. User satisfaction: 10/10

```
def test3():
    props     = IrregularProperties(1/100000)
    init_tex  = RandomPartition(props,
        KEEP_OUTSIDE)

    def hatch_map(face):
        props     = IrregularProperties(1/10000)
        return RandomPartition(props,
            CROP_ADD_BOUNDARY)(face)

    line      = ImportSVG("data/line6.svg")
    def line_to_curve(edge):
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    def scale_map(face):
        return Scale(Contour(face),0.95)

    texture = MapToEdges(line_to_curve,
        init_tex)
    texture = MapToFaces(scale_map, texture)
    texture = MapToFaces(hatch_map, texture)
    texture = MapToFaces(scale_map, texture)

    ExportSVG(texture, 1000)
```



User 1 - waves. 21 lines, five operators (green), three executions. User satisfaction: 8/10

```
def test4():
    lines1    = StripesProperties(0,200)
    lines2    = StripesProperties(pi/2.0,200)
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    line     = ImportSVG("data/line6.svg")
    def line_to_curve(edge):
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    def scale_map(face):
        return Scale(Contour(face),0.95)

    def hatch_map(face):
        props    = StripesProperties(pi/4.0,40)
        zorglub = StripesPartition(props)
        bloub = MapToEdges(line_to_curve,
            zorglub)
        return bloub(face)

    texture = MapToFaces(scale_map, grid_tex)
    texture = MapToFaces(hatch_map, texture)

    ExportSVG(texture, 500)
```

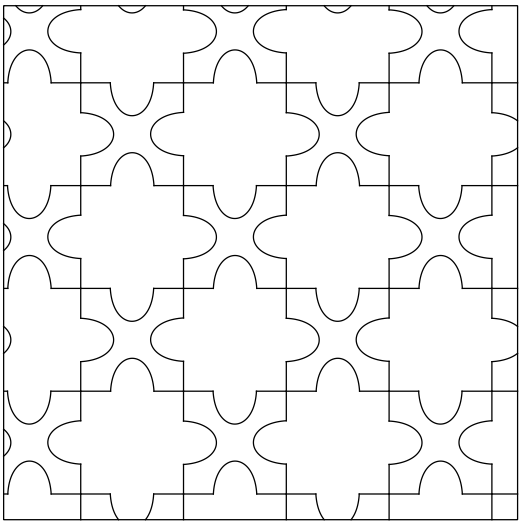


Interview of User 1

How easy was it to decide what you would do in order to reach the target designs? Extremely easy. No difficulty. The decomposition process looks very natural because it corresponds to the way our brain decomposes images into separate structures.
How easy was it to realize your plans by scripting in our tool? Not so hard. With a little bit of practice it becomes easy, and this learning appears to take little time. I would say labels took me the most time to handle in practice.
How often did you loose the understanding of what your script was doing? Never. The scripts are short and the syntax is clear. No problem.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? It works quite well. It fits our natural way to split structures into simpler parts. I liked this set of operators.
What are your thoughts about what you liked or disliked while experiencing our tool? The system itself is not far from being extremely comfortable. I would say the synthesis speed is the thing to be improved, because some trial-and-errors are slown down when the texture becomes complex.

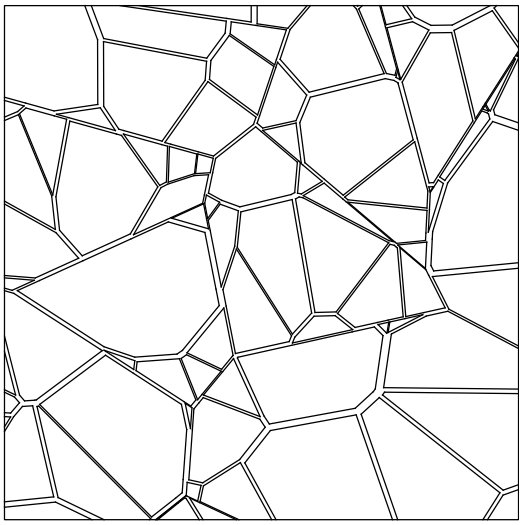
User 2 - puzzle. 26 lines, two operators (green), three executions. User satisfaction: 10/10

```
def test3():
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    line = ImportSVG("data/estelle/bitonio.
        svg")

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()

        face = LeftFace(edge)

        if((HasLabel(face,"h1") and HasLabel(
            face,"v1")) or
            (HasLabel(face,"h2") and HasLabel
                (face,"v2"))):
            src_c = PointLabeled(line,"start"
                )
            dst_c = PointLabeled(line,"end")
            src_v = Location(SourceVertex(
                edge))
            dst_v = Location(TargetVertex(
                edge))
            return MatchPoints(line,src_c,
                dst_c,src_v,dst_v)
        else:
            src_c = PointLabeled(line,"end")
            dst_c = PointLabeled(line,"start"
                )
            src_v = Location(SourceVertex(
                edge))
            dst_v = Location(TargetVertex(
                edge))
            return MatchPoints(line,src_c,
                dst_c,src_v,dst_v)

    texture = MapToEdges(line_to_curve,
        grid_tex)
    ExportSVG (texture, 1000)
```



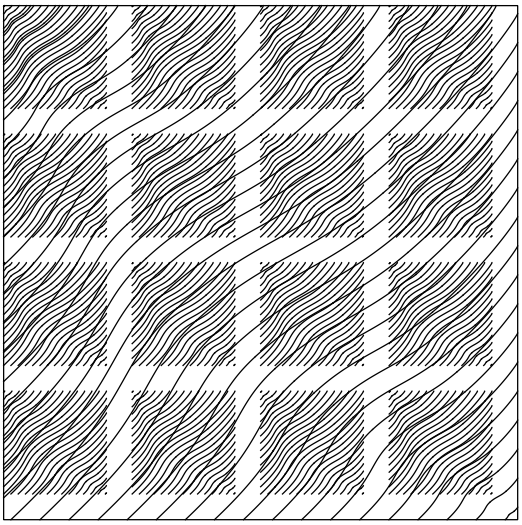
User 2 - cracks. 12 lines, four operators (green), two executions. User satisfaction: 6/10

```
def test4():
    props  = IrregularProperties(10/4000000)
    part   = RandomPartition(props,
        KEEP_OUTSIDE)

    def subdivideFace(face):
        angle = Random(face,0,2*pi,1)
        props2  = IrregularProperties
            (100/4000000)
        return RandomPartition(props2,
            CROP_ADD_BOUNDARY)(face)

    def scale_map(face):
        return Scale(Contour(face),0.95)

    texture = MapToFaces(subdivideFace, part)
    texture2 = MapToFaces(scale_map, texture)
    ExportSVG (texture2, 1000)
```



User 2 - waves. 29 lines, seven operators (green), three executions. User satisfaction: 7/10

```
def test5():
    lines1 = StripesProperties(0,50,200)
    lines2 = StripesProperties(pi/2,50,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    def vagues(face):
        if((HasLabel(face,"h1") and HasLabel(
            face,"v1"))):

            angle = pi/4
            lines = StripesProperties(angle
                ,10)
            return StripesPartition(lines)(
                face)
        else:
            return Nothing()

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()

        line    = ImportSVG("data/line6.svg")
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    texture = MapToFaces(vagues, grid_tex)
    T1 = MapToEdges(line_to_curve, texture)

    props   = StripesProperties(pi/4,40)
    stripes = StripesPartition(props)
    T2 = MapToEdges(line_to_curve, stripes)
    textureFinale = Union(T1,T2)

    ExportSVG (textureFinale, 1000)
```

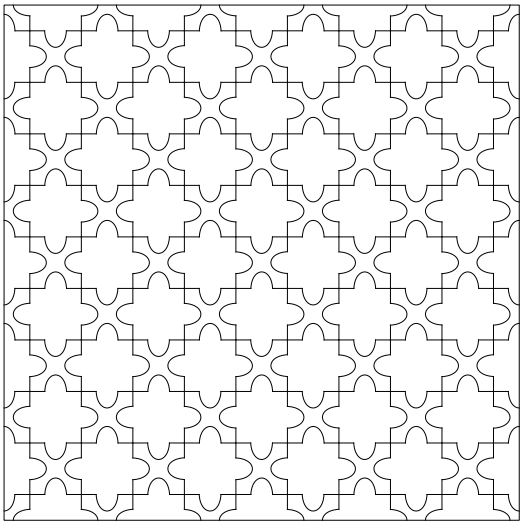


Interview of User 2

How easy was it to decide what you would do in order to reach the target designs? It was quite simple. The nodal formulation is common to other usual modeling tools. Only the labeling design was a bit harsh.
How easy was it to realize your plans by scripting in our tool? It was simple, and it becomes quicker and quicker when you start re-using bricks you designed before.
How often did you loose the understanding of what your script was doing? Never.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? I thought it was correct and logical. I liked the way of thinking textures as simple bases which are refined and then combined. It reminded me Maya's system for designing procedural textures.
What are your thoughts about what you liked or disliked while experiencing our tool? I liked the operator syntax embedded in Python, which is quite practical as a scripting language. I think the main limitation is the computational cost of the textures. I would like them to update in real-time when I achieve one piece of code. I liked the way modeling is thought in the system. You have an idea, you can easily make it real. The needed reasoning looks obvious. I also liked the quality of random textures that can be done, such as in the cracks example. Such textures are really hard to make with current commercial software.

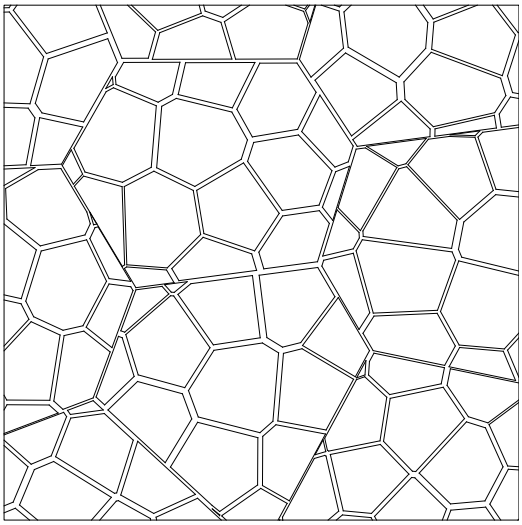
User 3 - puzzle. 20 lines, two operators (green), three executions. User satisfaction: 8/10

```
def test3():
    line = ImportSVG("data/estelle/bitonio.svg")

    def edge_to_bla(edge):
        if (HasLabel(LeftFace(edge), "v1") and HasLabel(LeftFace(edge), "h1")) or (HasLabel(LeftFace(edge), "v2") and HasLabel(LeftFace(edge), "h2")):
            src_c = PointLabeled(line,"start")
            dst_c = PointLabeled(line,"end")
        else :
            dst_c = PointLabeled(line,"start")
            src_c = PointLabeled(line,"end")

        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)

    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,KEEP_OUTSIDE)
    part = MapToEdges(edge_to_bla, grid_tex)

    ExportSVG(part, 2000)
```



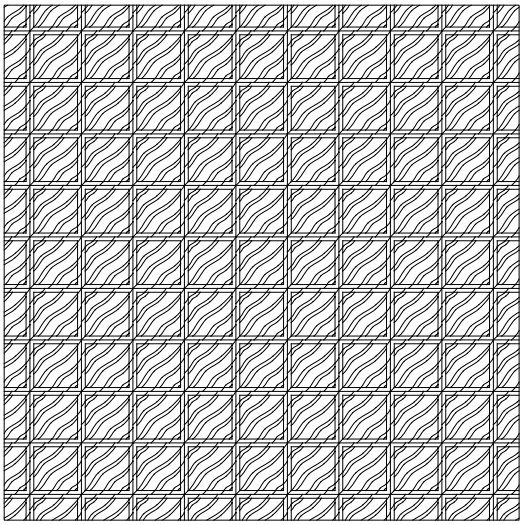
User 3 - cracks. 12 lines, four operators (green), two executions. User satisfaction: 5/10

```
def test4():
    def shrink_face(face):
        return Scale(Contour(face), Random(
            face, 0.90, 0.97, 1))

    def uniform_to_uniform(face):
        props2  = IrregularProperties
            (50/4000000)
        part2   = UniformPartition(props2,
            CROP_ADD_BOUNDARY)
        part2 = MapToFaces(shrink_face, part2)
        return part2(face)

    props  = IrregularProperties(5/4000000)
    part   = UniformPartition(props,
        KEEP_OUTSIDE)
    part = MapToFaces(uniform_to_uniform, part
        )

    ExportSVG(part, 2000)
```



User 3 - waves. 30 lines, nine operators (green), three executions. User satisfaction: 10/10

```
def test5():
    line     = ImportSVG("data/line6.svg")
    def line_to_curve(edge):
        if IsBoundary(edge):
            return ToCurve(edge)

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    def shrink_faces(face):
        return Scale(Contour(face), 0.85)

    def wavify_face(face):
        props   = StripesProperties(pi/4,50)
        stripes = StripesPartition(props)
        stripes = MapToEdges(line_to_curve,
            stripes)
        return stripes(face)

    props   = StripesProperties(pi/4,50)
    stripes = StripesPartition(props)
    lines1   = StripesProperties(0,200)
    lines2   = StripesProperties(pi/2.0,200)
    stripes = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)
    stripes = MapToFaces(wavify_face, stripes)

    lines1   = StripesProperties(0,200)
    lines2   = StripesProperties(pi/2.0,200)
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    grid_tex = MapToFaces(shrink_faces,
        grid_tex)
    grid_tex = MapToFaces(wavify_face,
        grid_tex)

    final = Union(grid_tex, stripes)

    ExportSVG(final, 2000)
```

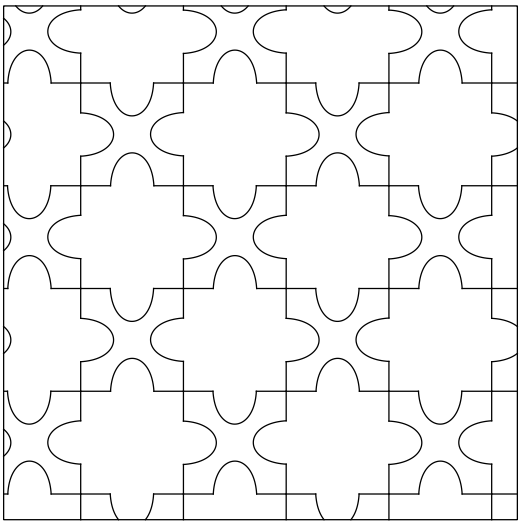


Interview of User 3

How easy was it to decide what you would do in order to reach the target designs? Quite easy. You always succeed in extracting some structure, some hierarchy in the target images. I think it would remain easy as long as there is some clear arrangement to see.
How easy was it to realize your plans by scripting in our tool? It was simple and enjoyable. I liked to have base bricks provided, that helped a lot constructing my own examples.
How often did you loose the understanding of what your script was doing? Never. I had some difficulties with the first combination involving a partition inside a mapper. After that I had no trouble.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? It works very well. I find the idea sound.
What are your thoughts about what you liked or disliked while experiencing our tool? I liked a lot that some ready-to-use bricks are provided on top of the set of operators. It helps a lot finding your way during the first moments. I found the computation a bit slow, but it is not blocking the creative process. Let say, it is not slow enough to be frustrating. One other cool point is that since you are scripting, you can use whatever text editor or IDE you want. It looks neglictible, but actually the usual sources of frustration with user interfaces are the basic shortcuts and the undo/redo feature. All of these are no problem here.

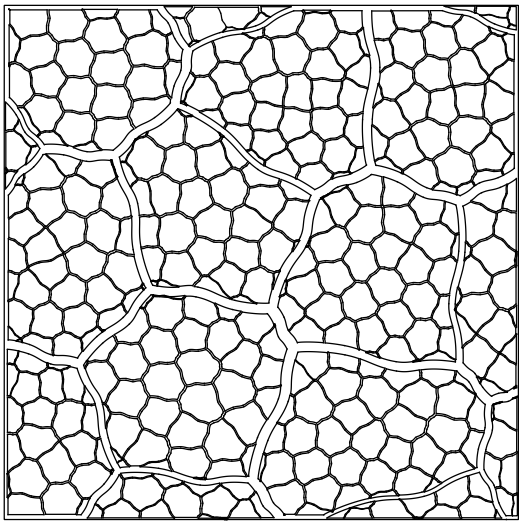
User 4 - puzzle (10min only). 20 lines, two operators (green), five executions. User satisfaction: 10/10

```
def test3():
    size = 2000
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    bitonio = ImportSVG("data/estelle/bitonio
        .svg")

    def line_to_curve(edge):

        src_c = PointLabeled(bitonio,"start")
        dst_c = PointLabeled(bitonio,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))

        face = LeftFace(edge)

        if((HasLabel(face,"h1") and HasLabel(
            face,"v1")) or
            (HasLabel(face,"h2") and HasLabel
                (face,"v2"))):
            return MatchPoints(bitonio,src_c,
                dst_c,src_v,dst_v)
        else:
            return MatchPoints(bitonio,dst_c,
                src_c,src_v,dst_v)

    T1 = MapToEdges(line_to_curve, grid_tex)
    ExportSVG(T1, 1000)
```



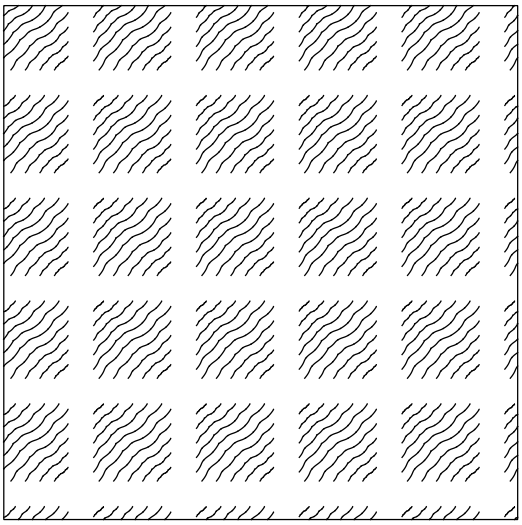
User 4 - cracks. 25 lines, seven operators (green), three executions. User satisfaction: 8/10

```
def test4():
    size = 2000
    props1  = IrregularProperties(30/(size*size))
    tex1    = UniformPartition(props1, CROP_ADD_BOUNDARY)

    def scale_map(face):
        return Scale(Contour(face),0.95)

    def part_map(face):
        s = 2000
        props  = IrregularProperties(1000/(s*s))
        return UniformPartition(props, CROP_ADD_BOUNDARY)(face)

    line    = ImportSVG("data/line6.svg")

    def line_to_curve(edge):
        if IsBoundary(edge):
            return ToCurve(edge)

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)

    T1 = MapToEdges(line_to_curve, tex1)
    T2 = MapToFaces(scale_map, T1)
    T3 = MapToFaces(part_map, T2)
    T4 = MapToEdges(line_to_curve, T3)
    T5 = MapToFaces(scale_map, T4)
    ExportSVG(T5, 1000)
```



User 4 - waves. 28 lines, six operators (green), four executions. User satisfaction: 6/10

```
def test5():
    size = 2000
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    def scale_map(face):
        return Scale(Contour(face),0.75)

    def hatch_map(face):
        angle = pi/4.
        size = Random(face,0,5)
        lines = StripesProperties(angle,20)
        return StripesPartition(lines)(face)

    line    = ImportSVG("data/line6.svg")

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    T1 = MapToFaces(scale_map, grid_tex)
    T2 = MapToFaces(hatch_map, T1)
    T3 = MapToEdges(line_to_curve, T2)
    T4 = MapToEdges(line_to_curve, T3)
    ExportSVG(T4, 1000)
```

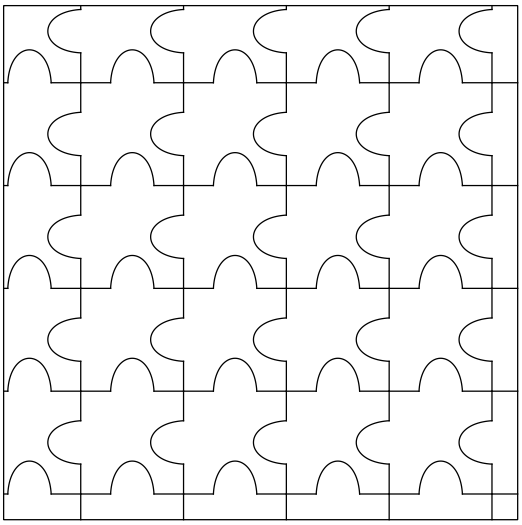


Interview of User 4

How easy was it to decide what you would do in order to reach the target designs? Quite easy. You only have to look at how to decompose the image, and then how to do each part with mappers. These are a lot like shaders, which is very practical.
How easy was it to realize your plans by scripting in our tool? It is quite easy to put the ideas into practice - I already coded with Python. I had a few difficulties only with the parameters of the Random operator and the density in RandomPartition and UniformPartition. Besides, it is possible to design the textures iteratively, which is super practical when you are editing code. Also the mapper/mapping operator design is very compact, which made my life a lot simpler.
How often did you loose the understanding of what your script was doing? Never.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? The concept is really sound. It allows to create complex results very easily, step by step. In particular I found that the way of editing topology in mappers is very powerful.
What are your thoughts about what you liked or disliked while experiencing our tool? Among all things I appreciated, I think the mappers were my favourite part. Maybe the thing I disliked was the computation time which was a bit heavy.

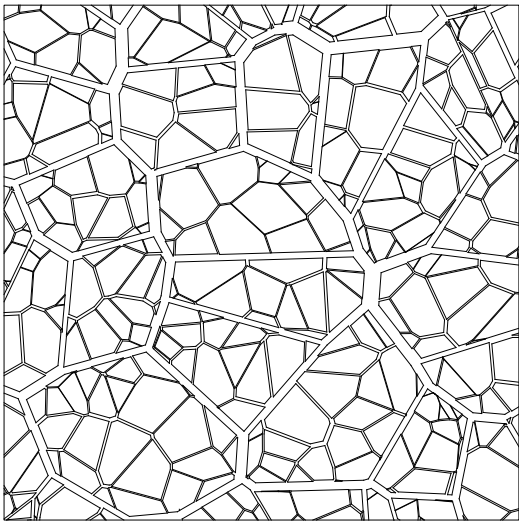
User 5 - puzzle. 23 lines, two operators (green), four executions. User satisfaction: 6/10

```
def test3():
  lines1 = StripesProperties(0,200)
  lines2 = StripesProperties(pi/2,200)
  SetEdgeLabels(lines1,"h1","h2")
  SetEdgeLabels(lines2,"v1","v2")
  grid_tex = GridPartition(lines1,lines2,
      KEEP_OUTSIDE)
  line = ImportSVG("data/estelle/bitonio.svg"
      )

  # element to edge
  def line_to_curve(edge):
    if IsBoundary(edge):
      return Nothing()

    src_c = PointLabeled(line,"start")
    dst_c = PointLabeled(line,"end")
    src_v = Location(SourceVertex(edge))
    dst_v = Location(TargetVertex(edge))

    if (HasLabel(IncidentEdges(TargetVertex(
        edge)), "h2")) or (HasLabel(
        IncidentEdges(TargetVertex(edge)), "
        v2")) and (HasLabel(edge, "v2") or
        HasLabel(edge, "h1")):
      return MatchPoints(line,src_c,dst_c,
          src_v,dst_v)

    elif (HasLabel(IncidentEdges(TargetVertex
        (edge)), "h1")) or (HasLabel(
        IncidentEdges(TargetVertex(edge)), "
        v1")) and (HasLabel(edge, "v1") or
        HasLabel(edge, "h2")):
      return MatchPoints(line,src_c,dst_c,
          src_v,dst_v)

    else:
      return MatchPoints(line, src_c, dst_c,
          dst_v, src_v)

  T = MapToEdges(line_to_curve, grid_tex)
  ExportSVG(T, 1000)
```



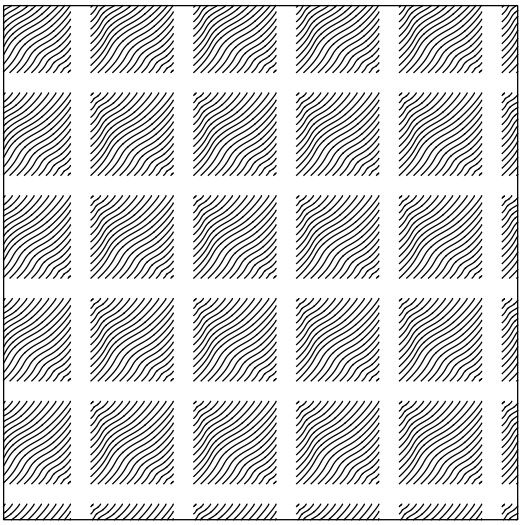
User 5 - cracks. 14 lines, five operators (green), two executions. User satisfaction: 8/10

```
def test4():
  size = 2000
  props     = IrregularProperties(30/(size*
      size))
  init_tex = RandomPartition(props,
      KEEP_OUTSIDE)

  def face_to_stripes(face):
    props     = IrregularProperties(150/(size*
        size))
    init_tex = RandomPartition(props,
        CROP_ADD_BOUNDARY)
    return init_tex(face)

  def scale_map(face):
    return Scale(Contour(face),Random(face,
        0.9, 0.99,0))

  T = MapToFaces(scale_map,init_tex)
  T2 = MapToFaces(face_to_stripes,T)
  T3 = MapToFaces(scale_map, T2)

  Add(T3(StartDomain(2000)))
```



User 5 - waves. 24 lines, six operators (green), three executions. User satisfaction: 9/10

```
def test5():
  lines1 = StripesProperties(0,200)
  lines2 = StripesProperties(pi/2,200)
  grid_tex = GridPartition(lines1,lines2,
      KEEP_OUTSIDE)

  props   = StripesProperties(pi/4,10)
  stripes = StripesPartition(props)
  line    = ImportSVG("data/line6.svg")

  def scale_map(face):
    return Scale(Contour(face),0.8)

  def face_to_stripes(face):
    lines = StripesProperties(pi/4,10)
    tex_stripes = MapToEdges(line_to_curve,
        StripesPartition(lines))
    return tex_stripes(face)

  def line_to_curve(edge):
    if IsBoundary(edge):
      return Nothing()

    src_c = PointLabeled(line,"start")
    dst_c = PointLabeled(line,"end")
    src_v = Location(SourceVertex(edge))
    dst_v = Location(TargetVertex(edge))
    return MatchPoints(line,src_c,dst_c,src_v
        ,dst_v)

  T2 = MapToFaces(scale_map, grid_tex)
  T3 = MapToFaces(face_to_stripes, T2)

  ExportSVG(T3, 1000)
```

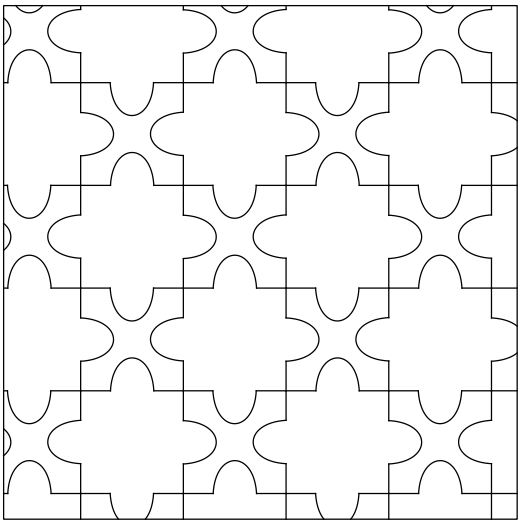


Interview of User 5

How easy was it to decide what you would do in order to reach the target designs? Easy, natural. As long as there is a structure to observe, it takes just an instant. It became a little harder for me for random patterns (cracks).
How easy was it to realize your plans by scripting in our tool? Surprisingly easy. In particular, I got exactly what I expected for the waves although I coded everything in a single strike.
How often did you loose the understanding of what your script was doing? Once with labels in the puzzle. Then never again.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? At least it was appropriate for the given examples, which is already something given the broad variety of targets. The principle is easy to understand and easy to visualize. It is easy to imagine the resulting texture of a given composition of operators.
What are your thoughts about what you liked or disliked while experiencing our tool? It was a nice experience. I liked that the operators fit well in a node-based strategy, which could be used for making an even more convenient interface.

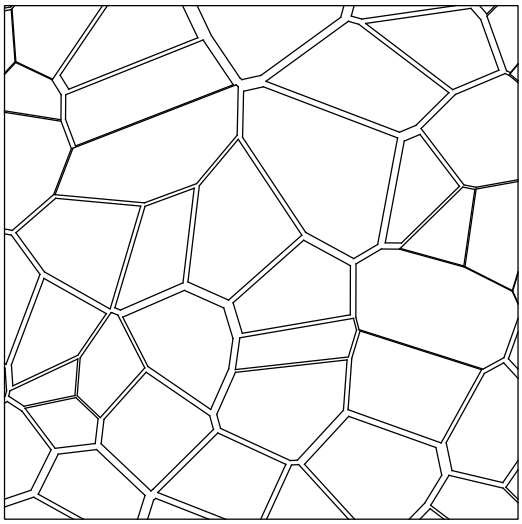
User 6 - puzzle. 20 lines, two operators (green), three executions. User satisfaction: 10/10

```
def test4():

  lines1 = StripesProperties(0,200)
  lines2 = StripesProperties(pi/2,200)
  SetEdgeLabels(lines1,"h1","h2")
  SetEdgeLabels(lines2,"v1","v2")
  grid_tex = GridPartition(lines1,lines2,
      KEEP_OUTSIDE)

  bitonio = ImportSVG("data/estelle/bitonio.
      svg")

  def line_to_curve(edge):
    if IsBoundary(edge):
      return Nothing()

    src_c = PointLabeled(bitonio,"start")
    dst_c = PointLabeled(bitonio,"end")
    src_v = Location(SourceVertex(edge))
    dst_v = Location(TargetVertex(edge))
    if((HasLabel(edge,"v1") and HasLabel(
        IncidentEdges(TargetVertex(edge)), "
        h1")) or (HasLabel(edge,"h1") and
        HasLabel(IncidentEdges(SourceVertex(
        edge)), "v1")) or (HasLabel(edge,"v2"
        ) and HasLabel(IncidentEdges(
        TargetVertex(edge)), "h2")) or (
        HasLabel(edge,"h2") and HasLabel(
        IncidentEdges(SourceVertex(edge)), "
        v2"))):
      dst_v = Location(SourceVertex(edge))
      src_v = Location(TargetVertex(edge))
    return MatchPoints(bitonio,src_c,dst_c,
        src_v,dst_v)

  T = MapToEdges(line_to_curve, grid_tex)

  ExportSVG(T, 1000)
```



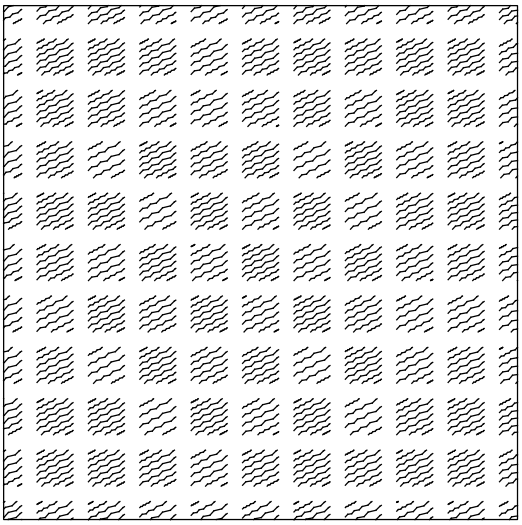
User 6 - cracks. Nine lines, two operators (green), two executions. User satisfaction: 8/10

```
def test5():
  size = 1000
  jensaisrien = 0.999999
  props     = IrregularProperties(30/(size*
      size))
  init_tex = RandomPartition(props,
      KEEP_OUTSIDE)

  def scale_map(face):
    return Scale(Contour(face),Random(face,
        0.9, jensaisrien, 1))

  T = MapToFaces(scale_map, init_tex)
  ExportSVG(T, 1000)
```



User 6 - waves. 25 lines, five operators (green), four executions. User satisfaction: 7/10

```
def test6():
  theta = 0
  width = 100
  lines1    = StripesProperties(theta,width)
  lines2    = StripesProperties(theta+pi/2.0,
      width)
  grid_tex = GridPartition(lines1,lines2,
      KEEP_OUTSIDE)

  def scale_map(face):
    return Scale(Contour(face),0.7)

  T = MapToFaces(scale_map,grid_tex)

  def face_to_stripes(face):
    width2 = BBoxWidth(face)/Random(face
        ,4,6,0)
    lines = StripesProperties(pi/6.0,width2)
    return StripesPartition(lines)(face)
  T2 = MapToFaces(face_to_stripes, T)

  line     = ImportSVG("data/line4.svg")

  def line_to_curve(edge):
    if IsBoundary(edge):
      return Nothing()

    src_c = PointLabeled(line,"start")
    dst_c = PointLabeled(line,"end")
    src_v = Location(SourceVertex(edge))
    dst_v = Location(TargetVertex(edge))
    return MatchPoints(line,src_c,dst_c,src_v
        ,dst_v)

  T3 = MapToEdges(line_to_curve, T2)

  ExportSVG(T3, 1000)
```

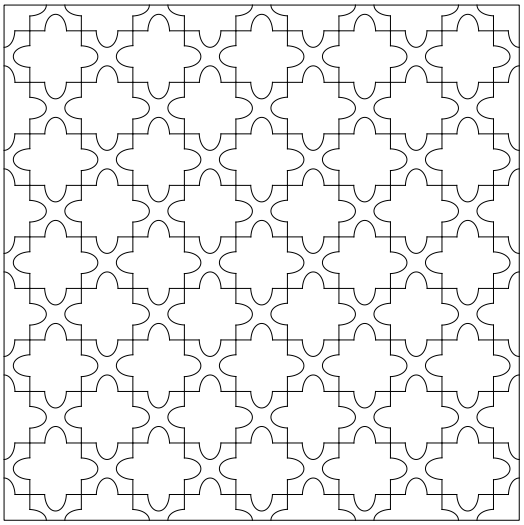


Interview of User 6

How easy was it to decide what you would do in order to reach the target designs? Easy. As a structure-finding task, it was easy.
How easy was it to realize your plans by scripting in our tool? It was very easy as well even though I never coded using Python before.
How often did you loose the understanding of what your script was doing? Never.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? It looked natural and sound.
What are your thoughts about what you liked or disliked while experiencing our tool? I liked that the model is intuitive.

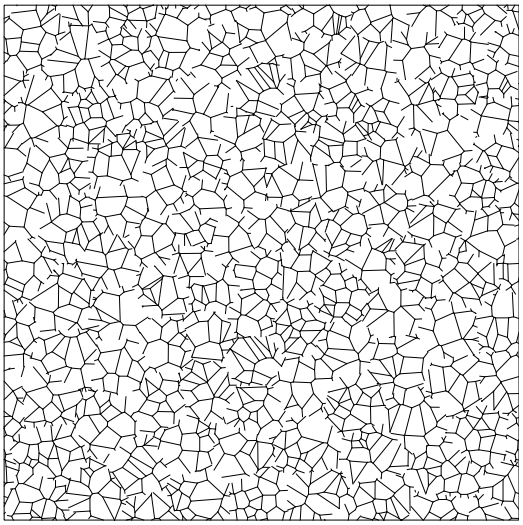
User 7 - puzzle. 25 lines, two operators (green), two executions. User satisfaction: 10/10

```
def test3():
    theta = 0
    width = 200
    lines1    = StripesProperties(theta,width)
    lines2    = StripesProperties(theta+pi
        /2.0,width)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    line = ImportSVG("data/estelle/bitonio.
        svg")

    def face_to_stripes2(edge):
        face_left = LeftFace(edge)

        sens = True

        if((HasLabel(face_left,"h1") and
            HasLabel(face_left,"v1")) or
           (HasLabel(face_left,"h2") and
               HasLabel(face_left,"v2"))):
            sens = False

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        if sens:
            return MatchPoints(line,src_c,
                dst_c,src_v,dst_v)
        else:
            return MatchPoints(line,src_c,
                dst_c,dst_v,src_v)

    tex2 = MapToEdges(face_to_stripes2,
        grid_tex)
    ExportSVG(tex2,2000)
```



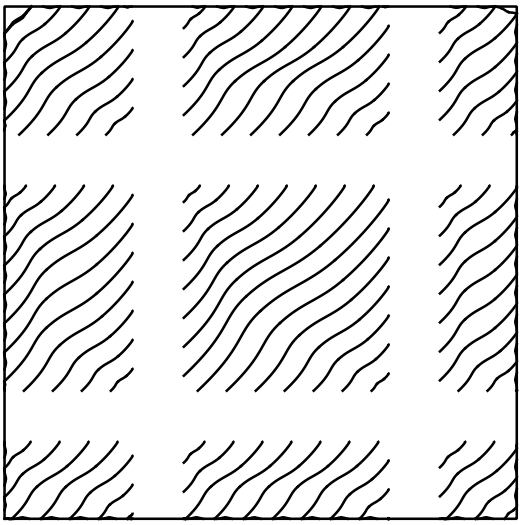
User 7 - cracks. Ten lines, three operators (green), two executions. User satisfaction: 7/10

```
def test4():
    size = 2000
    props1 = IrregularProperties(100/(size*
        size))
    tex1 = UniformPartition(props1,
        KEEP_OUTSIDE)

    def hatch_map(face):
        angle = Random(face,0,2*pi,1)
        props2 = IrregularProperties(1000/(
            size*size))
        return RandomPartition(props2,CROP)(
            face)

    tex3 = MapToFaces(hatch_map,tex1)
    ExportSVG(tex3,2000)
```



User 7 - waves. 25 lines, five operators (green), four executions. User satisfaction: 6/10

```
def test5():
    line     = ImportSVG("data/line6.svg")
    theta = 0
    width = 200

    lines1    = StripesProperties(theta,width,
        width/4)
    lines2    = StripesProperties(theta+pi
        /2.0,width, width/4)
    SetFaceLabels(lines1,"h1","h2")
    SetFaceLabels(lines2,"v1","v2")
    grid_tex1 = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    def clean(face):
        if((HasLabel(face,"h1") or HasLabel(
            face,"v1"))):
            return Nothing()
        return Contour(face)

    grid_tex1_clean = MapToFaces(clean,
        grid_tex1)

    props    = StripesProperties(pi/4,width
        /10)
    stripes = StripesPartition(props)

    grid_tex3 = Inside(stripes,
        grid_tex1_clean,CROP)

    def line_to_curve(edge):

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    tex3 = MapToEdges(line_to_curve,grid_tex3
        )
    ExportSVG(tex3,500)
```

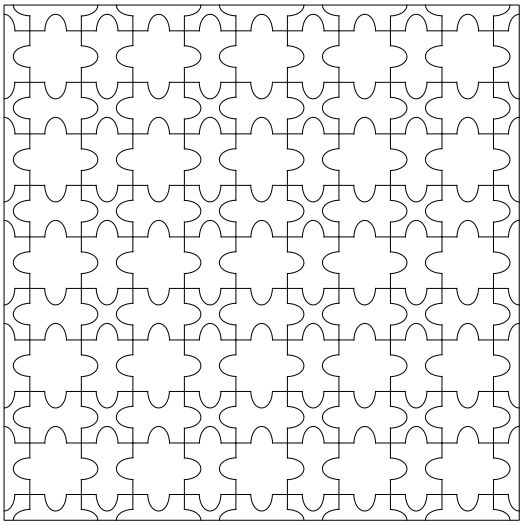


Interview of User 7

How easy was it to decide what you would do in order to reach the target designs? It was quite easy to set up a plan to reach the target designs. Sometimes you even see several ways to achieve the goal. I liked this because I find it always more safe to think before doing rather than dive into a dead end.
How easy was it to realize your plans by scripting in our tool? Quite easy. Maybe the only difficulty I had was the label filtering.
How often did you loose the understanding of what your script was doing? Never.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? It looks very natural. Actually I cannot think of any other way to model textures now that I learnt this model. I especially liked the intuitive way of stacking texture levels so as to obtain "multi-level" patterns.
What are your thoughts about what you liked or disliked while experiencing our tool? The tool was enjoyable. In particular there are few lines of code to manage even for complex textures. The computation time is a bit long, but nothing that cannot be optimized I believe. I think this approach has some big potential to be combined with a node-based GUI.

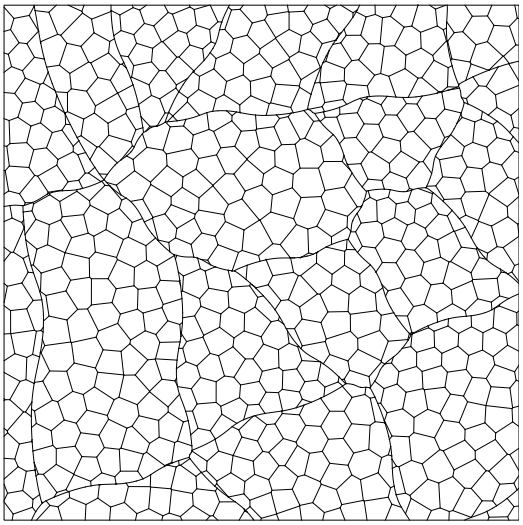
User 8 - puzzle. 21 lines, two operators (green), two executions. User satisfaction: 9/10

```
def test3():
    line = ImportSVG("data/estelle/bitonio.svg")
    lines1 = StripesProperties(0,200)
    lines2 = StripesProperties(pi/2,200)
    SetEdgeLabels(lines1,"h1","h2")
    SetEdgeLabels(lines2,"v1","v2")
    grid_tex = GridPartition(lines1,lines2,KEEP_OUTSIDE)

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        if(HasLabel(edge,"h2")):
            return MatchPoints(line,src_c,dst_c,dst_v,src_v)
        if(HasLabel(edge,"v2")):
            return MatchPoints(line,src_c,dst_c,dst_v,src_v)
        return MatchPoints(line,src_c,dst_c,src_v,dst_v)

    texE = MapToEdges(line_to_curve, grid_tex)
    ExportSVG(texE, 2000)
```



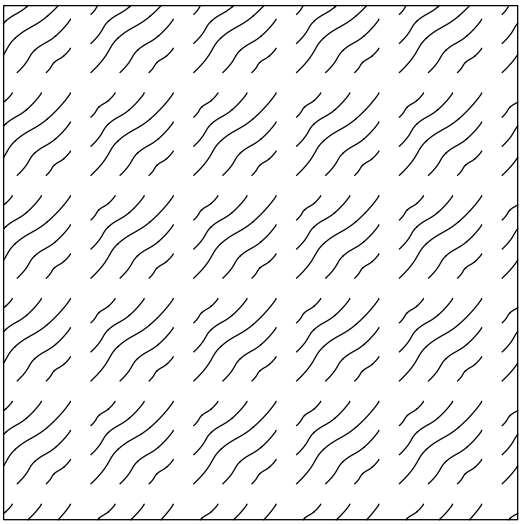
User 8 - cracks. 21 lines, four operators (green), three executions. User satisfaction: 10/10

```
def test4():
    size = 2000
    props2  = IrregularProperties(10/(size*
        size))
    tex2 = RandomPartition(props2,
        KEEP_OUTSIDE)
    line = ImportSVG("data/line6.svg")

    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()
        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    texE = MapToEdges(line_to_curve, tex2)

    def fill_map(face):
        density=500/(size*size)
        irr_props = IrregularProperties(
            density)
        unif_part = UniformPartition(
            irr_props,CROP_ADD_BOUNDARY)
        return unif_part(face)

    texF = MapToFaces(fill_map, texE)

    ExportSVG(texF, 2000)
```



User 8 - waves. 26 lines, five operators (green), five executions. User satisfaction: 9/10

```
def test5():
    theta = 0.0
    width = 200

    lines1    = StripesProperties(theta,width)
    lines2    = StripesProperties(theta+pi
        /2.0,width)

    grid_tex0 = GridPartition(lines1,lines2,
        KEEP_OUTSIDE)

    def scale_map(face):
        return Scale(Contour(face),0.8)

    grid_tex = MapToFaces(scale_map,
        grid_tex0)

    line = ImportSVG("data/line6.svg")
    def line_to_curve(edge):
        if IsBoundary(edge):
            return Nothing()

        src_c = PointLabeled(line,"start")
        dst_c = PointLabeled(line,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(line,src_c,dst_c,
            src_v,dst_v)

    def hatch_map(face):
        angle = pi/4
        lines = StripesProperties(angle,40)
        part = StripesPartition(lines)
        myTex = MapToEdges(line_to_curve,
            part)
        return myTex(face)

    hatch_tex = MapToFaces(hatch_map,
        grid_tex)

    ExportSVG(hatch_tex, 1000)
```



Interview of User 8

How easy was it to decide what you would do in order to reach the target designs? Very easy. I guess it is partly because I already know some node-based systems such as Blender's Cycles rendering engine.
How easy was it to realize your plans by scripting in our tool? Relatively easy. Trial-and-error is a good option which is allowed by the tool and that made my life easier for the most complex cases. It would be even easier if the computation time was a bit shorter.
How often did you loose the understanding of what your script was doing? Never. It takes a bit of time to learn each feature of the model, but once you have done it one time, it is very easy.
How did you feel about the general principle of designing textures with our partitions+mappers+combinations? You have to handle the learning curve, but then you have a clear and strong base for referring yourself to.
What are your thoughts about what you liked or disliked while experiencing our tool? I am very happy of the textures I was able to create during this first trial of the tool. I did not expect to reach such complex results, that was a good surprise.

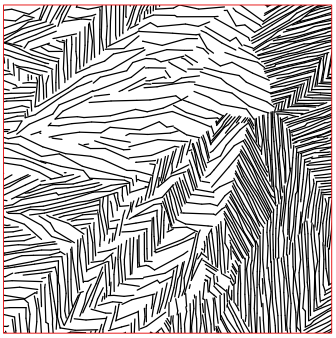
APPENDIX

E

PYTHON SCRIPTS FOR CARTOGRAPHIC DESIGN

In this appendix we show the scripts used to synthesize all the arrangements in Chapter 5.

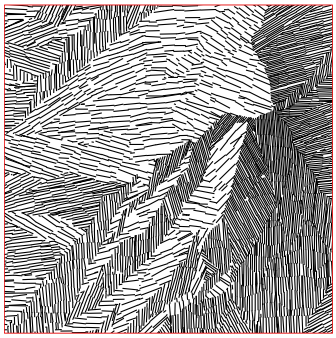
Noisy hatching



```
# Common to all scripts: Loading partition of mountain
p = ImportSHP("data/mountain/partition.shp")
# Data loading for control fields
mnt = LoaderGisRasters("data/DEM_Chamonix_5M_aiguille.tif")
orientations = LoadFaceWiseOrientations(p, "data/mountain/
    orient_L93.shp")
densities = LoadFaceWiseDensities(p, "data/mountain/
    densities_L93.shp")
# Finds the highest elevation in a face's adjacent vertices,
     normalized between 0 and 1
def highest_height(face):
    elevation = mnt.min_elevation()
    for v in IncidentVertices(face):
        x = Location(v).exact_point.approx_x()
        y = Location(v).exact_point.approx_y()
        new_alt = mnt(x,y)
        elevation = new_alt if new_alt > elevation else
            elevation
        elevation = (elevation - mnt.min_elevation()) / (mnt
            .max_elevation() - mnt.min_elevation())
    return elevation
# Finds the hatching spacing in function of the target
    density
def stripes_cycle(density_field):
    def out_f(face):
        density = density_field(Centroid(face))
        alt = highest_height(face) * highest_height(face)
        coeff_hatchspacing = 0.5 - 0.5 * alt if density <
            0.5 else 0.5 + 0.5 * alt
        return hatchspacing_min + (hatchspacing_max -
            hatchspacing_min) * coeff_hatchspacing
    return out_f

# Controlled arrangement
def noisy_hatching(density_field, orient_field):
    hatchspacing_min = 3.5
    hatchspacing_max = 15.0
    hatches_length = 30.0
    def out_arrangement(face):
        # Grid partition
        angle = orient_field(Centroid(face))
        hatches_spacing = stripes_cycle(density_field)(face)
        linesA = StripesProperties(angle, hatches_spacing)
        SetEdgeLabels(linesA, "hatches")
        linesB = StripesProperties(angle+pi/2.0,
            hatches_length)
        grid = GridPartition(linesA, linesB, KEEP_OUTSIDE)
        # Noise mappers
        randomization = 1.0
        def jitter(vertex):
            hatch_direction = Point(cos(angle), sin(angle))
            ortho_direction = Point(-sin(angle), cos(angle))
            random_x = Random(vertex, -hatches_spacing*
                randomization/2, hatches_spacing*
                randomization/2, 0)
            random_y = Random(vertex, -hatches_length*
                randomization/2, hatches_length*
                randomization/2, 1)
            location_randomization = ortho_direction *
                random_x + hatch_direction * random_y
            return Location(vertex) + location_randomization
        perturbate_edge = lambda edge: MatchPoints(ToCurve(
            edge), SourceVertex(edge), TargetVertex(edge),
            jitter(SourceVertex(edge)), jitter(TargetVertex
            (edge)))
        randomize_hatches = lambda edge: perturbate_edge(
            edge) if HasLabel(edge, "hatches") else Nothing
            ()
        randhatches = MapToEdges(randomize_hatches, grid)
        return randhatches(face)
    return out_arrangement

# Export
ExportSVG(noisy_hatching(densities,orientations), p)
```

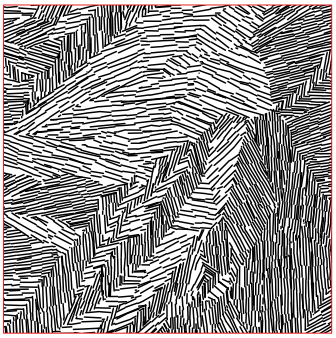
Multiscale density variations

```
# Controlled mapper: fine scale hatching
def hatch_face(density_field, orient_field):
    hatches_spacing = stripes_cycle(density_field)(face)
    angle = orient_field(Centroid(face))
    hatchlines = StripesProperties(angle, hatches_spacing)
    SetEdgeLabels(hatchlines, "hatches")
    hatchlines_part = StripesPartition(hatchlines)
    # Noise mappers
    randomization = 0.5
    def jitter(vertex):
        hatch_direction = Point(cos(angle), sin(angle))
        ortho_direction = Point(-sin(angle), cos(angle))
        random_x = Random(vertex, -hatches_spacing*
            randomization/2, hatches_spacing*randomization
            /2, 0)
        random_y = Random(vertex, -hatches_length*
            randomization/2, hatches_length*randomization
            /2, 1)
        location_randomization = ortho_direction * random_x
            + hatch_direction * random_y
        return Location(vertex) + location_randomization
    perturbate_edge = lambda edge: MatchPoints(ToCurve(edge)
        , SourceVertex(edge), TargetVertex(edge), jitter(
        SourceVertex(edge)), jitter(TargetVertex(edge)))
    randomize_hatches = lambda edge: perturbate_edge(edge)
        if HasLabel(edge, "hatches") else Nothing()
    # Mapping operator
    randhatches = MapToEdges(randomize_hatches, hatchlines)
    return randhatches(face)

# Controlled arrangement
def multiscale_density_variations(density_field,
    orient_field):
    hatchspacing_min = 2.0
    hatchspacing_max = 10.0
    hatches_length = 20.0
    def out_arrangement(face):
        # Coarse-scale grid partition
        angle = orient_field(Centroid(face))
        grid_spacing = 2 * hatchspacing_max
        linesA = StripesProperties(angle, grid_spacing)
        SetEdgeLabels(linesA, "grid")
        linesB = StripesProperties(angle+pi/2.0,
            hatches_length)
        SetEdgeLabels(linesB, "grid")
        grid = GridPartition(linesA, linesB, KEEP_OUTSIDE)
        # Noise mappers
        randomization = 1.0
        def jitter_grid(vertex):
            hatch_direction = Point(cos(angle), sin(angle))
            ortho_direction = Point(-sin(angle), cos(angle))
            random_x = Random(vertex, -grid_spacing*
                randomization/2, grid_spacing*randomization
                /2, 0)
            random_y = Random(vertex, -hatches_length*
                randomization/2, hatches_length*
                randomization/2, 1)
            location_randomization = ortho_direction *
                random_x + hatch_direction * random_y
            return Location(vertex) + location_randomization
        perturbate_grid_edge = lambda edge: MatchPoints(
            ToCurve(edge), SourceVertex(edge), TargetVertex
            (edge), jitter_grid(SourceVertex(edge)),
            jitter_grid(TargetVertex(edge)))
        randomize_grid = lambda edge: perturbate_edge(edge)
            if HasLabel(edge, "hatches") else Nothing()
        # Mapping to coarse noisy grid
        randgrid = MapToEdges(randomize_grid, grid)
        # Mapping to fine hatching
        return MapToFaces(hatch_face(density_field,
            orient_field), randgrid)(face)
    return out_arrangement
# Export
ExportSVG(multiscale_density_variations(densities,
    orientations), p)
```



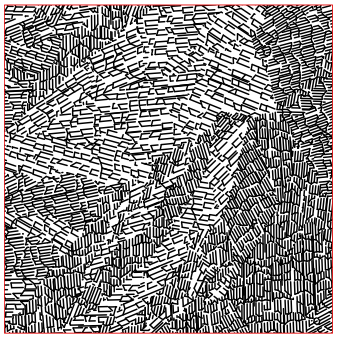
Balanced density variations

```
# Finds the hatching spacing in function of the target density
def linear_stripes_cycle(density_field):
    def out_f(face):
        density = density_field(Centroid(face))
        alt = highest_height(face) # Do not square anymore
        coeff_hatchspacing = 0.5 - 0.5 * alt if density < 0.5 else 0.5 + 0.5 * alt
        return hatchspacing_min + (hatchspacing_max - hatchspacing_min) * coeff_hatchspacing
    return out_f
# Controlled mapper: fine scale hatching
def hatch_face(density_field, orient_field):
    hatches_spacing = linear_stripes_cycle(density_field)(face)
    angle = orient_field(Centroid(face))
    hatchlines = StripesProperties(angle, hatches_spacing)
    SetEdgeLabels(hatchlines, "hatches")
    hatchlines_part = StripesPartition(hatchlines)
    # Noise mappers
    randomization = 0.5
    def jitter(vertex):
        hatch_direction = Point(cos(angle), sin(angle))
        ortho_direction = Point(-sin(angle), cos(angle))
        random_x=Random(vertex,-hatches_spacing*randomization/2,hatches_spacing*randomization/2,0)
        random_y=Random(vertex,-hatches_length*randomization/2,hatches_length*randomization/2,1)
        location_randomization = ortho_direction * random_x + hatch_direction * random_y
        return Location(vertex) + location_randomization
    perturbate_edge = lambda edge: MatchPoints(ToCurve(edge), SourceVertex(edge), TargetVertex(edge), jitter(SourceVertex(edge)), jitter(TargetVertex(edge)))
    randomize_hatches = lambda edge: Scale(perturbate_edge(edge), 0.95) if HasLabel(edge, "hatches") else Nothing()
    # Mapping operator
    randhatches = MapToEdges(randomize_hatches, hatchlines)
    return randhatches(face)
# Controlled arrangement
def balanced_density(density_field, orient_field):
    hatchspacing_min = 2.0
    hatchspacing_max = 10.0
    hatches_length = 20.0
    def out_arrangement(face):
        # Coarse-scale grid: same code as previous script
        # ...
        # ... Mapping to coarse noisy grid
        randgrid = MapToEdges(randomize_grid, grid)
        # Mapping to fine hatching
        return MapToFaces(hatch_face(density_field, orient_field), randgrid)(face)
    return out_arrangement
# Export
ExportSVG(balanced_density(densities,orientations), p)
```



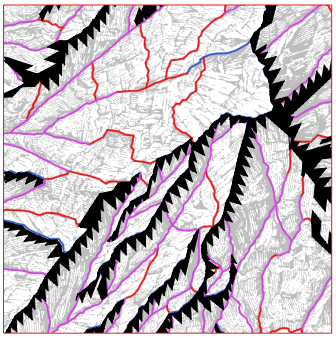
Roughness effect

```
# Controlled mapper: fine scale hatching
def hatch_face(density_field, orient_field):
    hatches_spacing = linear_stripes_cycle(density_field)(
        face)
    angle = orient_field(Centroid(face))
    hatchlines = StripesProperties(angle, hatches_spacing)
    SetEdgeLabels(hatchlines, "hatches")
    hatchlines_part = StripesPartition(hatchlines)
    # Noise mappers
    randomization = 0.5
    def jitter(vertex):
        hatch_direction = Point(cos(angle), sin(angle))
        ortho_direction = Point(-sin(angle), cos(angle))
        random_x=Random(vertex,-hatches_spacing*
            randomization/2,hatches_spacing*randomization
            /2,0)
        random_y=Random(vertex,-hatches_length*randomization
            /2,hatches_length*randomization/2,1)
        location_randomization = ortho_direction * random_x
            + hatch_direction * random_y
        return Location(vertex) + location_randomization
    perturbate_edge = lambda edge: MatchPoints(ToCurve(edge)
        , SourceVertex(edge), TargetVertex(edge), jitter(
        SourceVertex(edge)), jitter(TargetVertex(edge)))
    def is_part_of_comb(edge):
        if HasLabel(edge, "real_hatches"):
            return False
        source_is_bot = False
        target_is_bot = False
        for e in IncidentEdges(SourceVertex(edge)):
            if HasLabel(e, "real_hatches") and SourceVertex(
                edge) == TargetVertex(e):
                source_is_bot = True
                break
        for e in IncidentEdges(TargetVertex(edge)):
            if HasLabel(e, "real_hatches") and TargetVertex(
                edge) == TargetVertex(e):
                target_is_bot = True
                break
        return source_is_bot and target_is_bot
    def map_to_comb(edge):
        return perturbate_edge(edge) if is_part_of_comb(edge
            ) else Nothing()
    # Mapping operator
    randhatches = MapToEdges(map_to_comb, hatchlines)
    return randhatches(face)
# Controlled arrangement
def roughness_texture(density_field, orient_field):
    espacement_min = 4.0
    espacement_max = 8.0
    hatches_length = 15.0 #20.0
    def out_arrangement(face):
        # Coarse-scale grid: same code as previous script
        # ...
        # ... Mapping to coarse noisy grid
        randgrid = MapToEdges(randomize_grid, grid)
        # Mapping to fine hatching
        return MapToFaces(hatch_face(density_field,
            orient_field), randgrid)(face)
    return out_arrangement
# Export
ExportSVG(roughness_texture(densities,orientations), p)
```

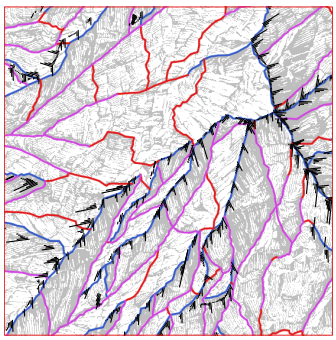


Simple ridge hatching (partition added for clarity)

```
# Controlled arrangement
def simple_ridge_hatching(density_field, orient_field):
    hatchspacing_min = 2.0
    hatchspacing_max = 10.0
    hatches_length = 20.0
    coulis_length = 15
    def out_arrangement(face):
        # Coarse-scale grid partition
        angle = orient_field(Centroid(face))
        grid_spacing = 2 * hatchspacing_max
        linesA = StripesProperties(angle, grid_spacing)
        SetEdgeLabels(linesA, "grid")
        # Grid with small angle so as to make triangle
            shapes
        linesB = StripesProperties(angle+pi/4.0,
            coulis_length)
        SetEdgeLabels(linesB, "grid")
        grid = GridPartition(linesA, linesB,
            CROP_ADD_BOUNDARY)
        # Mapper: keep only faces adjacent to ridges
        def keep_ridge(miniface):
            border_found = False
            for e in IncidentEdges(miniface):
                if HasLabel(e, "crete"):
                    border_found = True
            if not border_found:
                return Nothing()
            return Scale(Contour(miniface), 0.98)
        # Mapping to fine hatching
        return MapToFaces(keep_ridge(density_field,
            orient_field), grid)(face)
    return out_arrangement
# Export
ExportSVG(simple_ridge_hatching(densities,orientations), p)
```



Finer ridge hatching (partition added for clarity)



```
# Controlled arrangement
def finer_ridge_hatching(density_field, orient_field):
    peakshape1 = ImportLineSVG("data/montagne/ps1.svg")
    tailshape1 = ImportLineSVG("data/montagne/ts1.svg")
    espacement_pics = 20.0
    longueur_pics = 20.0
    longueur_queues = 20.0
    def out_arrangement(face):
        # Stripes supporting the hatching
        angle = orient_field(Centroid(face))
        linesA = StripesProperties(angle, espacement_pics)
        SetEdgeLabels(linesA, "peak_line")
        stripes = StripesPartition(linesA)
        # Determine if this edge recieves hatching
        def must_provide_coulis(e):
            other_face = LeftFace(e) if face == RightFace(e)
                 else RightFace(e)
            d1 = density_field(Centroid(face))
            d2 = density_field(Centroid(other_face))
            return d1 < 0.5 and d1 < d2
        # Mapper: keep only faces adjacent to ridges
        def keep_ridge(miniface):
            border_found = False
            for e in IncidentEdges(miniface):
                if HasLabel(e, "crete"):
                    border_found = True
            if not border_found:
                return Nothing()
            return Scale(Contour(miniface), 0.98)
        def generate_fill_shape(vertex):
            if not HasLabel(IncidentEdges(vertex), "crete"):
                return Nothing()
            peak_source = PointLabeled(peakshape1, "start")
            peak_target = PointLabeled(peakshape1, "end")
            tail_source = PointLabeled(tailshape1, "start")
            tail_target = PointLabeled(tailshape1, "end")
            normalized_orient=Point(cos(angle),sin(angle))
            peak = MatchPoints(peakshape1,peak_source,
                peak_target,Location(vertex),Location(
                vertex)+normalized_orient*longueur_pics)
            # Find tail orientation
            ridge_edge = UNDEFINED_EDGE
            for e in IncidentEdges(vertex):
                if HasLabel(e, "crete") and
                     must_provide_coulis(e):
                    ridge_edge = e
            other_ridge_vertex = SourceVertex(ridge_edge) if
                 vertex == TargetVertex(ridge_edge) else
                TargetVertex(ridge_edge)
            tail_orient = Location(vertex) - Location(
                other_ridge_vertex)
            clockwise_orth = Point(normalized_orient.y(), -
                normalized_orient.x())
            if tail_orient.dot(clockwise_orth) < 0:
                tail_orient = tail_orient * (-1.0)
            tail_orient = tail_orient / tail_orient.length()
            tail = MatchPoints(tailshape1, tail_source,
                tail_target, Location(vertex), Location(
                vertex) + tail_orient * longueur_queues)
            res = Append(peak, tail)
        def hatch_vertex(vertex):
            tofill = generate_fill_shape(vertex)
            hatches_spacing = 2.0
            angle = orient_field(Location(vertex))
            # StripesPartition
            hatchlines = StripesProperties(angle,
                hatches_spacing)
            hatchlines_part = StripesPartition(hatchlines)
        # Mapping to fine hatching
        return MapToVertices(hatch_vertex(density_field,
            orient_field), stripes)(face)
    return out_arrangement
# Export
ExportSVG(finer_ridge_hatching(densities,orientations), p)
```

REFERENCES

[AdPWS10] ALVES DOS PASSOS V., WALTER M., SOUSA M. C.: Sample-based synthesis of illustrative patterns. In *Proceedings of the 2010 18th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2010), PACIFIC GRAPHICS '10, IEEE Computer Society, pp. 109–116.

[AKA13] ALMERAJ Z., KAPLAN C. S., ASENTE P.: Towards effective evaluation of geometric texture synthesis algorithms. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2013), NPAR '13, ACM, pp. 5–14.

[Ali73] ALINHAC M.: Présentation et Etude Critique des Meilleures Cartes Françaises et Etrangères. In *Colloque sur la cartographie des régions montagneuses* (May 1973).

[ASP07] ASENTE P., SCHUSTER M., PETTIT T.: Dynamic planar map illustration. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*, 3 (July 2007).

[AW90] ABRAM G. D., WHITTED T.: Building block shaders. *Computer Graphics (Proceedings of SIGGRAPH '90) 24*, 4 (Sept. 1990), 283–288.

[BA06] BAXTER W. V., ANJYO K. I.: Latent doodle space. *Computer Graphics Forum (Proceedings of Eurographics 2006) 25*, 3 (2006), 477–485.

[BBT*06] BARLA P., BRESLAV S., THOLLOT J., SILLION F., MARKOSIAN L.: Stroke pattern analysis and synthesis. *Computer Graphics Forum (Proceedings of Eurographics 2006) 25*, 3 (Sept. 2006), 663–671.

[BG89] BAUDELAIRE P., GANGNET M.: Planar maps: An interaction paradigm for graphic design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1989), CHI '89, ACM, pp. 313–318.

[BST09] BRATKOVA M., SHIRLEY P., THOMPSON W. B.: Artistic rendering of mountainous terrain. *ACM Trans. Graph. 28*, 4 (Sept. 2009), 102:1–102:17.

[CK14] CAMPBELL N. D. F., KAUTZ J.: Learning a manifold of fonts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014) 33*, 4 (July 2014), 91:1–91:11.

[Coo84] COOK R. L.: Shade trees. *Computer Graphics (Proceedings of SIGGRAPH '84) 18*, 3 (Jan. 1984), 223–231.

[Dem01] DEMERS O.: *Digital Texturing and Painting*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[DiV13] DIVERDI S.: A brush stroke synthesis toolbox. In *Image and Video-Based Artistic Stylisation*, vol. 42. Springer, 2013, pp. 23–44.

[DRVDP14] DALSTEIN B., RONFARD R., VAN DE PANNE M.: Vector Graphics Complexes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014) 33*, 4 (July 2014), 133:1–133:12.

[Duf66] DUFOUR A.: *Cours de cartographie, Cartographie générale, Titre I*. Tech. rep., IGN, Ecole Nationale des Sciences Géographiques, Paris, France, 1966.

[EMP*02] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*, 3 ed. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

[EWHS08] EISEMANN E., WINNEMÖLLER H., HART J. C., SALESIN D.: Stylized vector art from 3d models with region support. *Computer Graphics Forum (proceedings of the Eurographics Symposium on Rendering) 27*, 4 (2008), 1199–1207.

[FHW12] FOGEL E., HALPERIN D., WEIN R.: *CGAL arrangements and their applications*. Geometry and Computing. Springer, 2012.

[GH15] GEISTHÖVEL R., HURNI L.: Automatic rock depiction via relief shading. In *Proceedings of the 27th International Cartographic Conference, Rio de Janeiro, Brazil* (2015).

[GLLD12] GALERNE B., LAGAE A., LEFEBVRE S., DRETTAKIS G.: Gabor noise by example. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012) 31*, 4 (July 2012), 73:1–73:9.

[GTDS10] GRABLI S., TURQUIN E., DURAND F., SILLION F. X.: Programmable rendering of line drawing from 3d scenes. *ACM Transactions on Graphics 29*, 2 (Apr. 2010), 18:1–18:20.

[Gui05] GUILHOT N.: *Histoire d'une parenthèse cartographique : Les Alpes du Nord dans la cartographie topographique française aux 19e et 20e siècles*. PhD thesis, 2005.

[Gup97] GUPTILL A. L.: *Rendering in Pen and Ink: The Classic Book On Pen and Ink Techniques for Artists, Illustrators, Architects, and Designers (Practical Art Books)*. Watson-Guptill, Aug. 1997.

[Hen02] HENDERSON P.: Functional geometry. *Higher Order and Symbolic Computation 15*, 4 (Dec. 2002), 349–365.

[Her02] HERTZMANN A.: Fast paint texture. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering* (New York, NY, USA, 2002), NPAR '02, ACM, pp. 91–ff.

[HHD03] HILLER S., HELLWIG H., DEUSSEN O.: Beyond Stippling – Methods for Distributing Objects on the Plane. *Computer Graphics Forum (Proceedings of Eurographics 2003) 22*, 3 (Sept. 2003), 515–522.

[HL11] HURTUT T., LECORDIX F.: Cartography of mountain rocky areas, a statistical modeling and drawing of element arrangements. In *Proceedings of the 25th International Cartographic Conference, Paris, France* (2011).

[HL12] HURTUT T., LANDES P.-E.: Synthesizing structured doodle hybrids. In *SIGGRAPH Asia 2012 Posters* (New York, NY, USA, 2012), SA '12, ACM, pp. 43:1–43:1.

[HLT*09] HURTUT T., LANDES P.-E., THOLLOT J., GOUSSEAU Y., DROUILLHET R., COEURJOLLY J.-F.: Appearance-guided synthesis of element arrangements by example. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering* (2009), NPAR '09, ACM, pp. 51–60.

[IMIM08] IJIRI T., MĚCH R., IGARASHI T., MILLER G.: An example-based procedural system for element arrangement. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*, 2 (2008), 429–436.

[Jen04] JENNY B.: Bringing traditional panorama projections from the painter's canvas to the digital realm. In *Proceedings of the 4th ICA Mountain Cartography Workshop, Vall de Núria. Monografies tècniques, Institut Cartogràfic de Catalunya, Barcelona* (2004), pp. 151–157.

[JGG*14] JENNY B., GILGEN J., GEISTHÖVEL R., MARSTON B. E., HURNI L.: Design Principles for Swiss-style Rock Drawing. *The Cartographic Journal 51*, 4 (2014), 360–371.

[JL97] JOBARD B., LEFER W.: Creating evenly-spaced streamlines of arbitrary density. pp. 43–56.

[Jul81] JULESZ B.: Textons, the elements of texture perception, and their interactions. *Nature 290*, 5802 (1981), 91–97.

[Kap10] KAPLAN C. S.: Curve evolution schemes for parquet deformations. In *Proceedings of Bridges 2010: Mathematics, Music, Art, Architecture, Culture* (2010), Tessellations Publishing, pp. 95–102.

[KNBH12] KALOGERAKIS E., NOWROUZEZAHRAI D., BRESLAV S., HERTZMANN A.: Learning hatching for pen-and-ink illustration of surfaces. *ACM Trans. on Graph. 31*, 1 (2012), 1–17.

[LD05] LAGAE A., DUTRÉ P.: A procedural object distribution function. *ACM Transactions on Graphics 24*, 4 (2005), 1442–1461.

[LG08] L. GONDOL A. LE BRIS F. L.: Cartography of High Mountain Areas - Testing of a New Digital Cliff Drawing Method. In *Proceedings of the 6th ICA Mountain Cartography Workshop, Mountain Mapping and Visualisation, Lenk, Switzerland* (2008).

[LGH13] LANDES P.-E., GALERNE B., HURTUT T.: A shape-aware model for discrete texture synthesis. *Computer Graphics Forum (proceedings of the Eurographics Symposium on Rendering) 32*, 4 (2013), 67–76.

[LH15] L. HURNI A. TSORLINI L. P. R. S. R. G.: Re-mapping the cliffs of mount everest: Deriving a synoptic map from large-scale mountain map data. In *Proceedings of the 27th International Cartographic Conference, Rio de Janeiro, Brazil* (2015).

[Lin68] LINDENMAYER A.: Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology 18* (1968).

[LWSF10] LI H., WEI L.-Y., SANDER P. V., FU C.-W.: Anisotropic blue noise sampling. In *ACM SIGGRAPH Asia 2010 papers* (2010), ACM, pp. 167:1–167:12.

[MM12] MĚCH R., MILLER G.: The *Deco* framework for interactive procedural modeling. *Journal of Computer Graphics Techniques (JCGT) 1*, 1 (Dec 2012), 43–99.

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25*, 3 (July 2006), 614–623.

[MWT11] MA C., WEI L.-Y., TONG X.: Discrete element textures. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011) 30*, 4 (July 2011), 62:1–62:10.

[OBW*08] ORZAN A., BOUSSEAU A., WINNEMÖLLER H., BARLA P., THOLLOT J., SALESIN D.: Diffusion curves: a vector representation for smooth-shaded images. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*, 3 (Aug. 2008), 92:1–92:8.

[PL96] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[SGSG71] STINY G., GIPS J., STINY G., GIPS J.: Shape grammars and the generative specification of painting and sculpture. In *Proceedings of the Workshop on generalisation and multiple representation, Leicester* (1971).

[SHS02] SECORD A., HEIDRICH W., STREIT L.: Fast primitive distribution for illustration. In *Rendering Techniques 2002 (Eurographics Symposium on Rendering)* (2002), Debevec P., Gibson S., (Eds.), Springer-Verlag Wien New York, pp. 215–226.

[SSBG10] SCHMID J., SUMNER R. W., BOWLES H., GROSS M.: Programmable motion effects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010) 29*, 4 (July 2010), 57:1–57:9.

[TG80] TREISMAN A. M., GELADE G.: A feature-integration theory of attention. *Cognitive Psychology 12* (1980), 97–136.

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22*, 3 (July 2003), 669–677.

[WZS98] WONG M. T., ZONGKER D. E., SALESIN D. H.: Computer-generated floral ornament. In *Proceedings of SIGGRAPH '98* (New York, NY, USA, 1998), ACM, pp. 423–434.

[XCW14] XING J., CHEN H.-T., WEI L.-Y.: Autocomplete painting repetitions. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 172:1–172:11.

[YBY*13] YEH Y.-T., BREEDEN K., YANG L., FISHER M., HANRAHAN P.: Synthesis of tiled patterns using factor graphs. *ACM Transactions on Graphics 32*, 1 (Feb. 2013), 3:1–3:13.