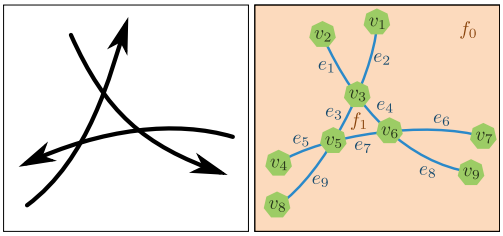
Programmable 2D Arrangements for Element Texture Design - Tutorial

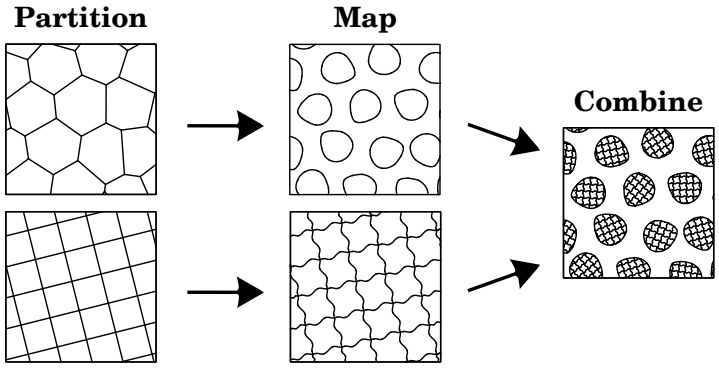
HUGO LOI
Inria-LJK (UGA, CNRS)
and
THOMAS HURTUT
École Polytechnique de Montréal
and
ROMAIN VERGNE and JOELLE THOLLOT
Inria-LJK (UGA, CNRS)

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*.



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

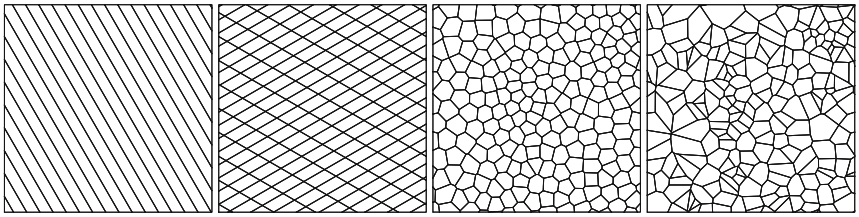


II. Partitions

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 Appendix 2). You will be able to manipulate the cells of partitions using mappers.

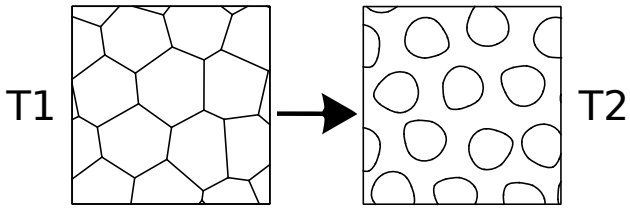


Practice: execute partitions scripts and modify their parameters

```
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.

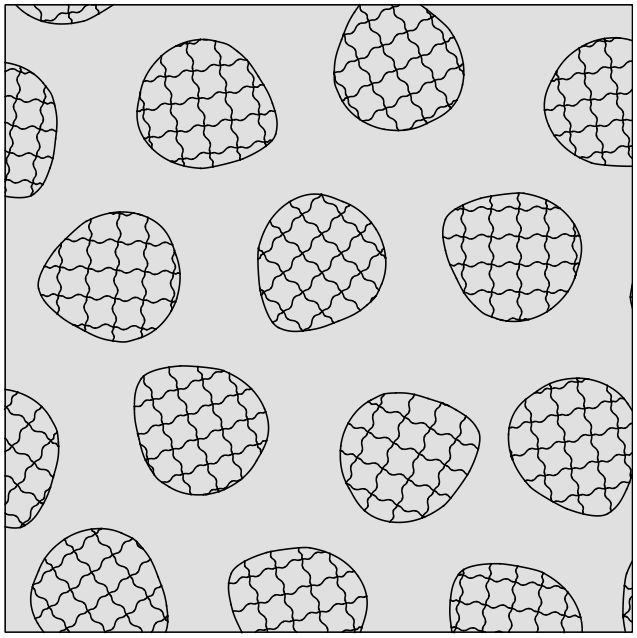
—**Do not use global, hard-coded coordinates**. For example if your mapper depends on the position (255, 42) then it will never have a homogeneous effect all over the texture.

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

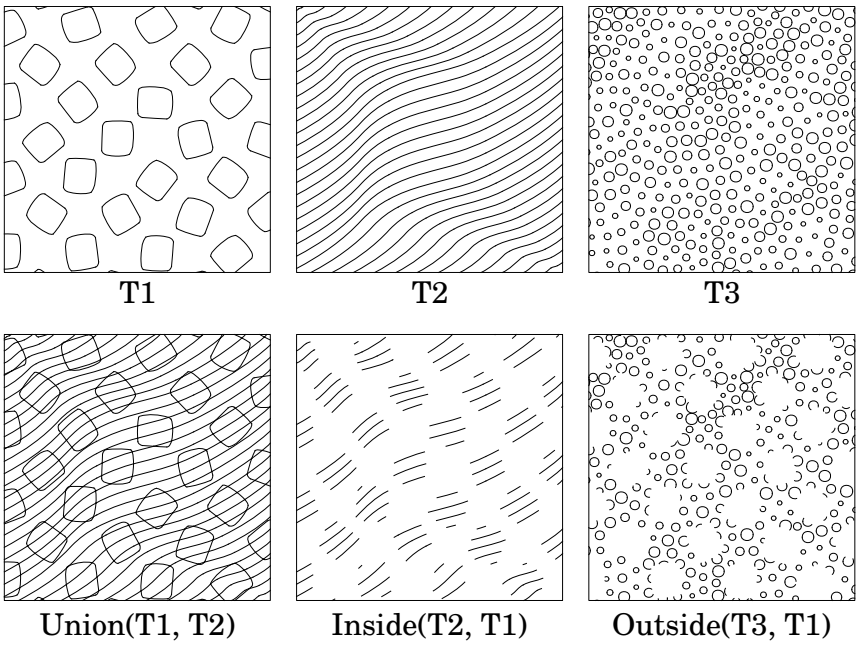
IV. Combining Textures

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)
```

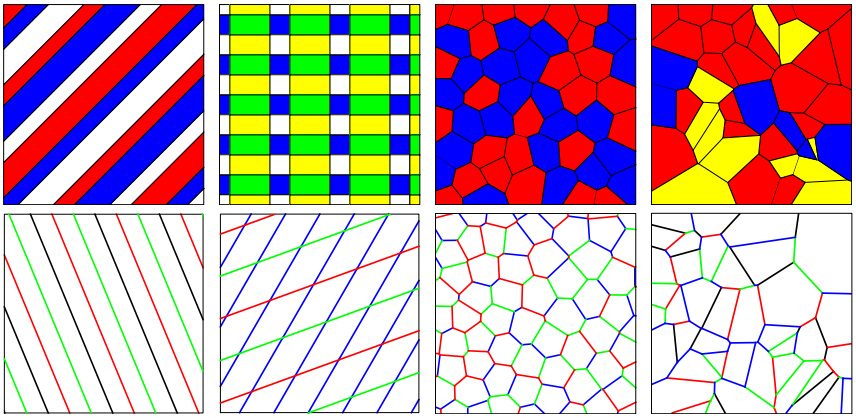


You can also combine two textures using either Union, Inside or Outside:



V. Labels

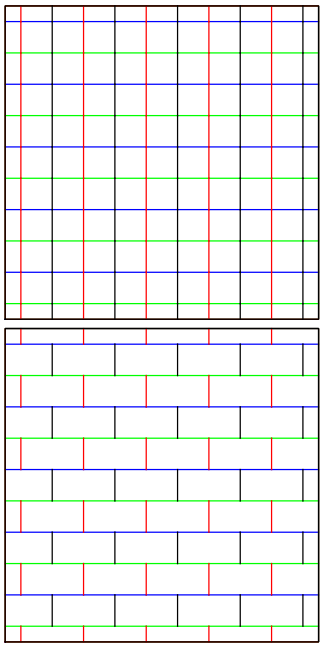
You can add additional information to the planar map's cells. In particular you can add *labels*, which can then be used in mappers for varying effects.
In StripesPartition labels are defined periodically: "Red, blue, red, blue, ..." for instance. It is the same in GridPartition, except with two dimensions. 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

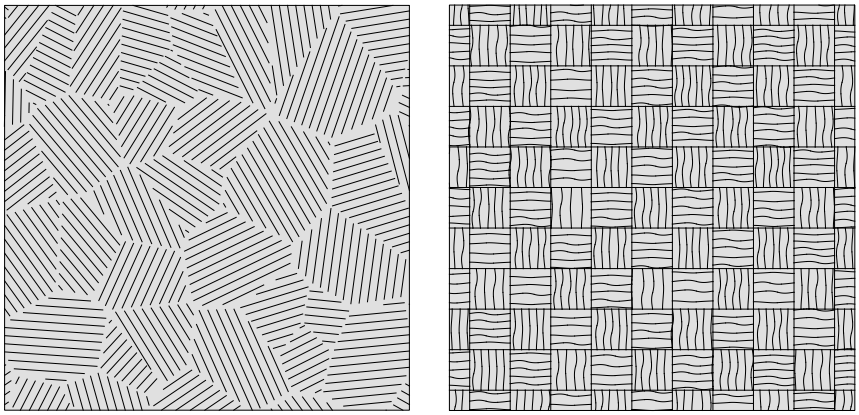
Labels can be used everytime you want to make your mappers' behavior variable. Here is an example with periodic refinement of the texture's topology:

```
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 (Appendix III) together so as to make the two following textures:



Appendix 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

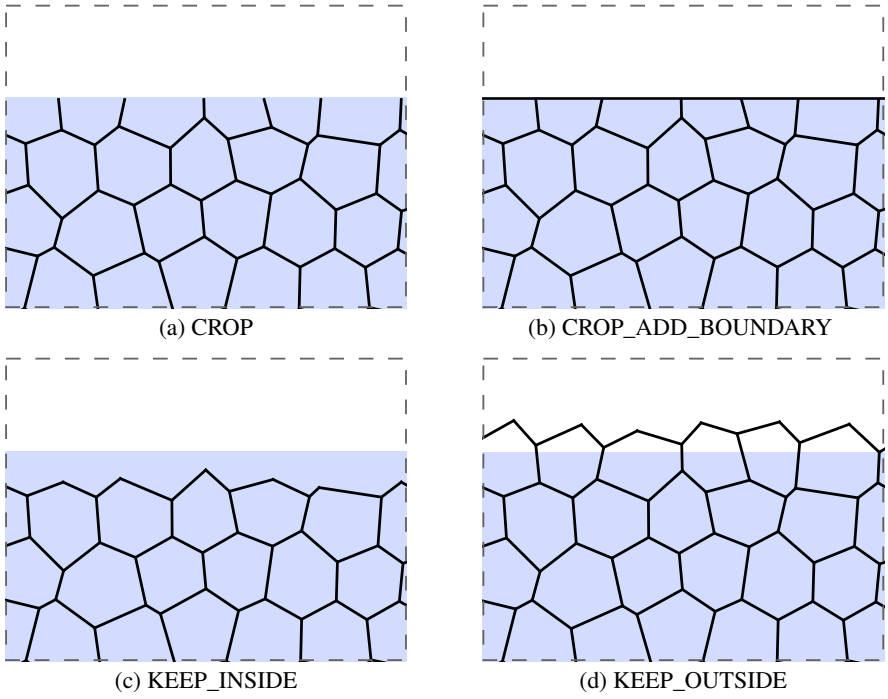
Merging operators

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

Appendix II. Border Management.



(a) CROP (b) CROP_ADD_BOUNDARY

(c) KEEP_INSIDE (d) KEEP_OUTSIDE

Appendix 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")
```

Appendix 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)
```