

Map Style Formalization: Rendering Techniques Extension for Cartography - Supplementary Materials

S. Christophe¹, B. Duménieu¹, J. Turbet², C. Hoarau¹, N. Mellado³, J. Ory¹, H. Loi⁴,
A. Masse¹, B. Arbelot⁴, R. Vergne⁴, M. Brédif¹, T. Hurtut⁵, J. Thollot⁴, D. Vanderhaeghe³

¹ Université Paris-Est, IGN, SRIG, COGIT/MATIS, France ² IPBS

³ IRIT, Université de Toulouse, CNRS, INPT, UPS, UT1C, UT2J, France

⁴ Univ. Grenoble Alpes, CNRS, Inria ⁵ Polytechnique Montréal

This document presents two types of supplementary materials: a catalog of services enabling expressive map design, and a set of additional results.

1. Catalog

The purpose of this catalog is to illustrate the capabilities of our approach, according to the services we have implemented in the software Geoxygène [Geo16]. The generation of vector patterns is currently not fully integrated, and requires further processing outside of Geoxygène, as described in the dedicated section.

1.1. Curves stylization

We propose three services to stylize curves, described in Listings 1, 2 and 3. These three services are respectively tailored to painting simulation (see Figure 1), procedural variations of the stroke width (see Figure 3), and simple texture repetition (see Figure 2). They have been exclusively used to produce all the examples shown in the paper and in this document. These examples show the variability of the stylization formalism we propose in this paper.

Listing 1: Definition of the *StrokePainting* curve stylization service

```
<RenderingMethod>
    <Name>StrokePainting</Name>
    <GeneralMethodReference>BrushStroke</GeneralMethodReference>
    <ShaderList>
        <ShaderRef gltype="GL_FRAGMENT_SHADER">.../shaders/subshader1d.derain.frag.glsl</ShaderRef>
    </ShaderList>
    <Parameters>
        <Parameter>
            <Description>The portion of the stroke extremities where the brush
            sharpness smoothly increases/decreases.</Description>
            <Name>extremitiesLength</Name>
            <Type>float</Type>
            <Restrictions>
                <BoundsRestriction min="0.0" max="1000.0"/>
            </Restrictions>
        </Parameter>
    </Parameters>
</RenderingMethod>
```

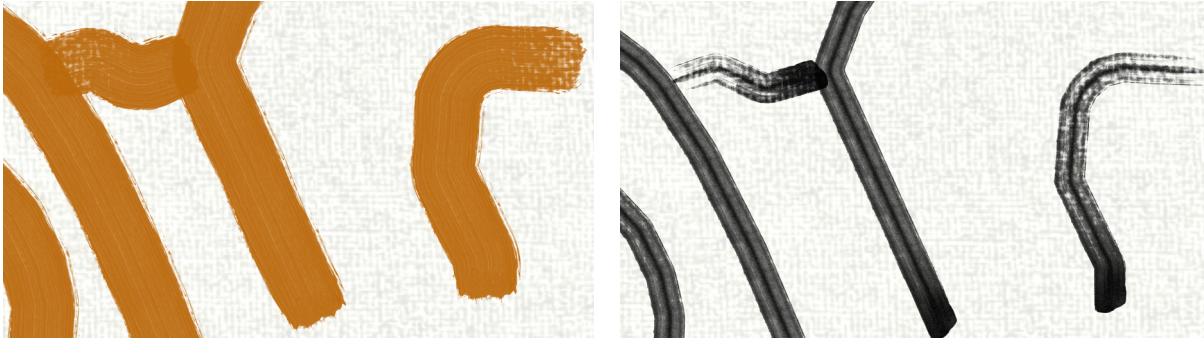


Figure 1: Top: Generation of stylized lines using the *StrokePainting* expressive style. Variations are produced by changing the stroke width and the applied texture. Bottom: SE description of example a).

Listing 2: Definition of the *CassiniRoadsPainting* curve stylization service

```

<RenderingMethod>
    <Name>CassiniRoadsPainting</Name>
    <GeneralMethodReference>BrushStroke</GeneralMethodReference>
    <ShaderList>
        <ShaderRef glType="GL_FRAGMENT_SHADER">../shaders/subshader1d.roads.frag.glsl</ShaderRef>
    </ShaderList>
    <Parameters>
        <Parameter required="true">
            <Name>extremitiesLength</Name>
            <Type>Float</Type>
        </Parameter>
    </Parameters>
</RenderingMethod>

```

```

<Stroke>
    <CssParameter name="stroke">#ffffff</CssParameter>
    <CssParameter name="stroke-opacity">1.0</CssParameter>
    <CssParameter name="stroke-width">100.0</CssParameter>
    <CssParameter name="stroke-linejoin">round</CssParameter>
    <CssParameter name="stroke-linecap">round</CssParameter>
    <ExpressiveStroke>
        <ExpressiveMethod>CassiniRoadsPainting</ExpressiveMethod>
        <ExpressiveParameter name="transitionSize">1.0</ExpressiveParameter>
        <ExpressiveParameter name="paperReferenceMapScale">100000.0</ExpressiveParameter>
        <ExpressiveParameter name="paperRoughness">2.0</ExpressiveParameter>
        <ExpressiveParameter name="brushRoughness">4.0</ExpressiveParameter>
        <ExpressiveParameter name="strokePressure">17.0</ExpressiveParameter>
        <ExpressiveParameter name="strokeSoftness">0.2</ExpressiveParameter>
        <ExpressiveParameter name="paperSizeInCm">4.0</ExpressiveParameter>
        <ExpressiveParameter name="extremitiesLength">210.375</ExpressiveParameter>
        <ExpressiveParameter name="paperTexture">
            <SimpleTexture XRepeat="false" YRepeat="false">
                <Displacement x="0.0" y="0.0"/>
                <ScaleFactor x="1.0" y="1.0"/>
                <Rotation Angle="0.0"/>
                <URI>../images/canvas-normalized.png</URI>
            </SimpleTexture>
        </ExpressiveParameter>
        <ExpressiveParameter name="brushTexture">
            <SimpleTexture XRepeat="false" YRepeat="false">
                <Displacement x="0.0" y="0.0"/>
                <ScaleFactor x="1.0" y="1.0"/>
                <Rotation Angle="0.0"/>
                <URI>../images/StJeanDeLuz/road.png</URI>
            </SimpleTexture>
        </ExpressiveParameter>
    </ExpressiveStroke>
</Stroke>

```

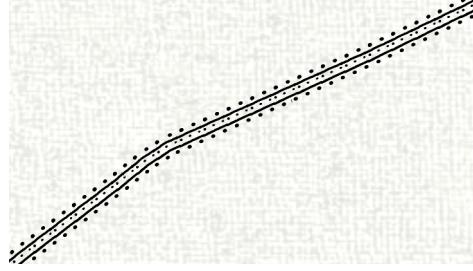


Figure 2: Generation of stylized lines using the *CassiniRoadsPainting* expressive style.

Listing 3: Definition of the *CassiniRiverPainting* curve stylization service

```

<RenderingMethod>
    <Name>CassiniRiverPainting</Name>
    <GeneralMethodReference>BrushStroke</GeneralMethodReference>
    <ShaderList>
        <ShaderRef gltype="GL_FRAGMENT_SHADER">../shaders/subshader1d.hydro.frag.glsl</ShaderRef>
    </ShaderList>
    <Parameters>
        <Parameter>
            <Description>The portion of the stroke extremities where the brush
                sharpness smoothly increases/decreases.</Description>
            <Name>extremitiesLength</Name>
            <Restrictions>
                <BoundsRestriction min="0.0" max="1000.0"/>
            </Restrictions>
            <Type>float</Type>
        </Parameter>
        <Parameter>
            <Name>hardness</Name>
            <Type>float</Type>
        </Parameter>
        <Parameter>
            <Name>thickness</Name>
            <Type>float</Type>
        </Parameter>
        <Parameter>
            <Name>noiseWavelength</Name>
            <Type>float</Type>
        </Parameter>
    </Parameters>
</RenderingMethod>

```

```

<Stroke>
  <CssParameter name="stroke">#000000</CssParameter>
  <CssParameter name="stroke-opacity">1</CssParameter>
  <CssParameter name="stroke-width">100.0</CssParameter>
  <ExpressiveStroke>
    <ExpressiveMethod>CassiniRiverPainting</ExpressiveMethod>
    <ExpressiveParameter name="extremitiesLength">500.0</ExpressiveParameter>
    <ExpressiveParameter name="brushStartWidth">330.0</ExpressiveParameter>
    <ExpressiveParameter name="brushEndWidth">330.0</ExpressiveParameter>
    <ExpressiveParameter name="transitionSize">5.0</ExpressiveParameter>
    <ExpressiveParameter name="brushRoughness">4.5</ExpressiveParameter>
    <ExpressiveParameter name="strokePressure">10.0</ExpressiveParameter>
    <ExpressiveParameter name="strokeSoftness">0.4</ExpressiveParameter>
    <ExpressiveParameter name="paperReferenceMapScale">30000.0</ExpressiveParameter>
    <ExpressiveParameter name="paperSizeInCm">4.0</ExpressiveParameter>
    <ExpressiveParameter name="paperRoughness">2.0</ExpressiveParameter>
    <ExpressiveParameter name="hardness">0.8</ExpressiveParameter>
    <ExpressiveParameter name="thickness">0.9</ExpressiveParameter>
    <ExpressiveParameter name="noiseWavelength">80.0</ExpressiveParameter>
    <ExpressiveParameter name="paperTexture">
      <SimpleTexture XRepeat="false" YRepeat="false">
        <Displacement x="0.0" y="0.0"/>
        <ScaleFactor x="1.0" y="1.0"/>
        <Rotation Angle="0.0"/>
        <URI>../images/canvas-normalized.png</URI>
      </SimpleTexture>
    </ExpressiveParameter>
    <ExpressiveParameter name="brushTexture">
      <SimpleTexture XRepeat="false" YRepeat="false">
        <Displacement x="0.0" y="0.0"/>
        <ScaleFactor x="1.0" y="1.0"/>
        <Rotation Angle="0.0"/>
        <URI>../images/brush2-335-311.png</URI>
      </SimpleTexture>
    </ExpressiveParameter>
  </ExpressiveStroke>
</Stroke>

```



Figure 3: Top: Generation of stylized lines using the CassiniRiverPainting expressive style. Bottom: SE description.

1.2. Raster texture synthesis for region filling

We propose a single service to stylize regions, described in Listing 4. This service takes as input texture patches, and distribute them into the stylized region.

Listing 4: Definition of the *TextureFill* region stylization service

```

<RenderingMethod>
  <Name>TextureFill</Name>
  <GeneralMethodReference>WorldToScreenSpace</GeneralMethodReference>
  <ShaderList>
    <ShaderRef gltype="GL_VERTEX_SHADER">identity.vert.gls1</ShaderRef>
    <ShaderRef gltype="GL_FRAGMENT_SHADER">polygon.texture.frag.gls1</ShaderRef>
  </ShaderList>
  <Parameters>
    <Parameter>
      <Name>fill-texture</Name>
      <Type>Texture</Type>
      <UniformRef>textureColor1</UniformRef>
    </Parameter>
  </Parameters>
</RenderingMethod>

```

The way patches are distributed is controlled by a scalar field, which is computed inside the region : patches are distributed w.r.t. the value of the field, and oriented w.r.t. its gradient. To do so, each individual texture patch is defined by a file URL, and a 1D probability function, describing the probability of a patch to be pasted in the region according to the scalar field value at its location (see Figures 4 and 6).

The scalar field generation can be adjusted by two main parameters, as shown in Figure 4:

- MaxCoastlineLength: By default the scalar field is computed as the distance field from the polygon boundaries. In order to cope with polygon edges describing the border of the map, and not the border of the polygon (e.g. top left corner in Figure 4), the user can filter long edges using this parameter. Filtering is disabled when $\text{MaxCoastlineLength}=0$.
- Rotation: This parameter is used only if $\text{MaxCoastlineLength}<0$. In that case, the scalar field is a simple linear gradient, oriented according to the input direction angle. As a consequence, textures patches are distributed linearly, i.e. independently of the region geometry and by following an uniform direction, as shown in Figure 6, sea and ground regions.

```

<Fill>
  <ExpressiveFill>
    <ExpressiveMethod>TextureFill</ExpressiveMethod>
    <ExpressiveParameter name="fill-texture">
      <TileDistributionTexture XRepeat="false" YRepeat="false">
        <Displacement x="0.0" y="0.0"/>
        <ScaleFactor x="0.5" y="0.5"/>
        <Rotation Angle="0.0"/>
        <MaxCoastlineLength>3000.0</MaxCoastlineLength>
      <Tile>
        <URI>.../images/StJeanDeLuz/seapatch1.png</URI>
        <ScaleFactor>0.5</ScaleFactor>
        <MinDistance>0.0</MinDistance>
        <MaxDistance>2200.0</MaxDistance>
        <InRangeProbability>1.0</InRangeProbability>
        <OutOfRangeProbability>0.0</OutOfRangeProbability>
      </Tile>
      <tile> ... </Tile>
      <Resolution>600.0</Resolution>
      <Blending>GRAPHCUT</Blending>
      <DistributionManagement>KEEP_OUTSIDE</DistributionManagement>
      <BlurSize>1</BlurSize>
    </TileDistributionTexture>
  </ExpressiveParameter>
</ExpressiveFill>
<CssParameter name="fill">#ffffff</CssParameter>
<CssParameter name="fill-opacity">1.0</CssParameter>
</Fill>

```

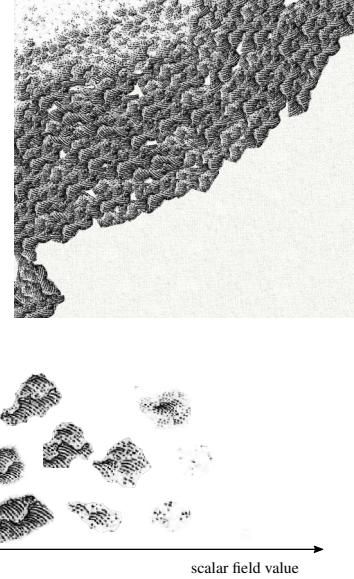


Figure 4: Region stylized using the *TextureFill* expressive style. 10 input textures patches are distributed inside the region according to the distance field, generated from the coast edges.

1.3. Generation of vector patterns

As stated in the paper, vector patterns are generated using an external controllable process. The scripts used in the paper are given at the end of this document.

2. Results

We present, in the section, various expressive rendering of cartographic data. Figure 5 shows a map with a Japanese print style: This stylization is obtained by applying expressive methods to linear and surface features (roads and hydrological network, and the sea area), along with overlayed paper textures and color gradients. The forest (in green) consists in two layers of cartographic data. The first layer contains the surface geometries of the forest areas, and is rendered with a raster texture generated from patches of sketched trees. The second layer was manually created upstream from the map design process by slightly simplify and move the forest areas, it is rendered with a solid green color fill. A canvas texture is overlaid on the entire map to obtain the visible variations.

We also present in figure 6 a more abstract map that imitates the pointillistic style of Paul Signac [Sig13]. Three cartographic layers are stylized with raster patches extracted from a painting of the artist.

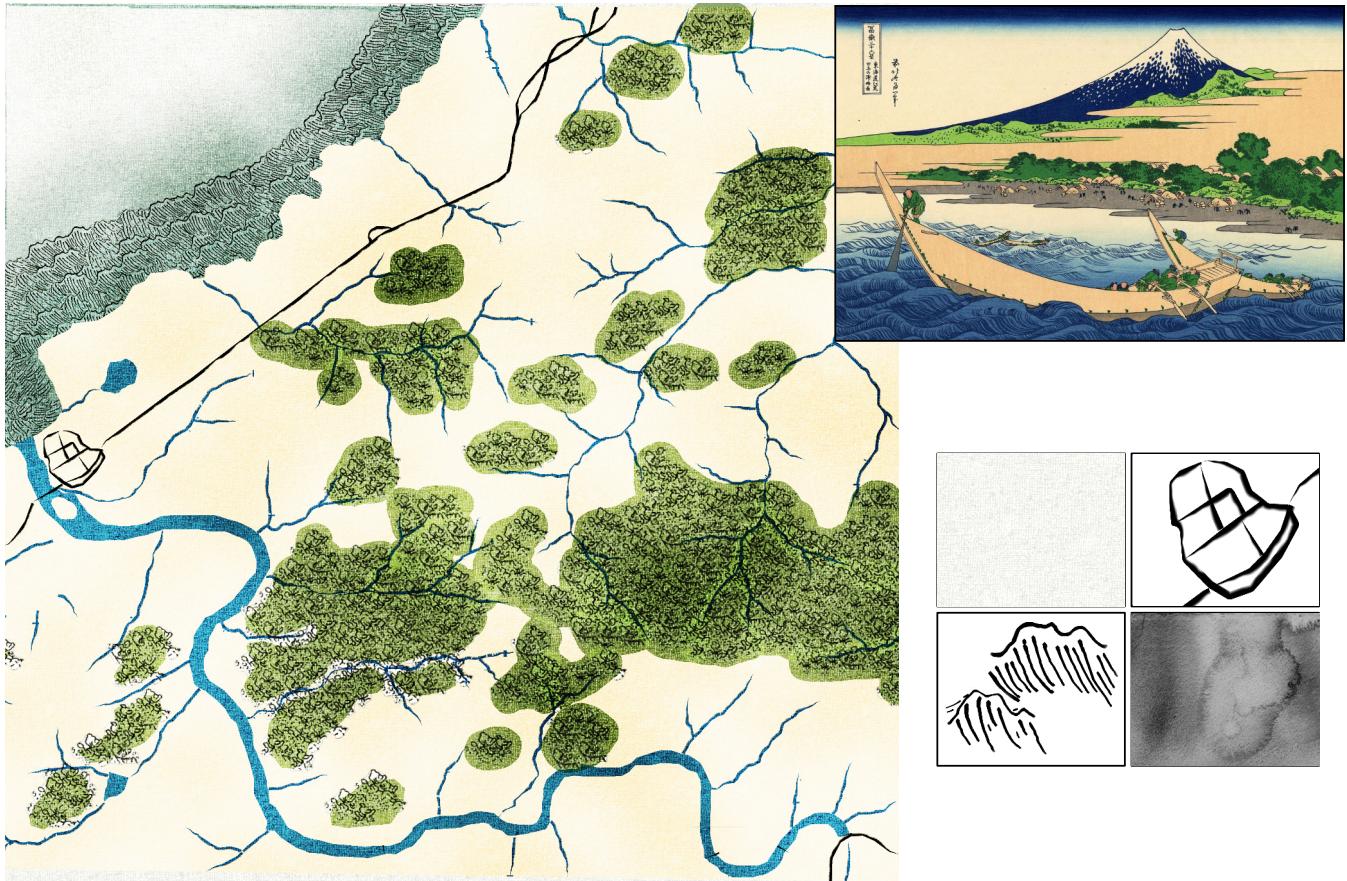


Figure 5: An example of a map stylized as a Japanese print. This result is obtained with a combination of layer blending and application of expressive methods for the sea, the linear roads and rivers, and the forest. This map takes inspiration from Hokusai's paintings [Hok30].

References

- [Geo16] Geoxygene. <http://oxygene-project.sourceforge.net/>, 2016. 1
- [Hok30] HOKUSAI K.: Sea coast at tago, near ejiri. part of the series thirty-six views of mount fuji, no. 36. c., 1830. woodblock color print, 25.2 x36 cm. 6
- [Sig13] SIGNAC P.: The green towers, la rochelle, 1913. Oil on canvas, 81.3 x 100.3 cm. 5, 7

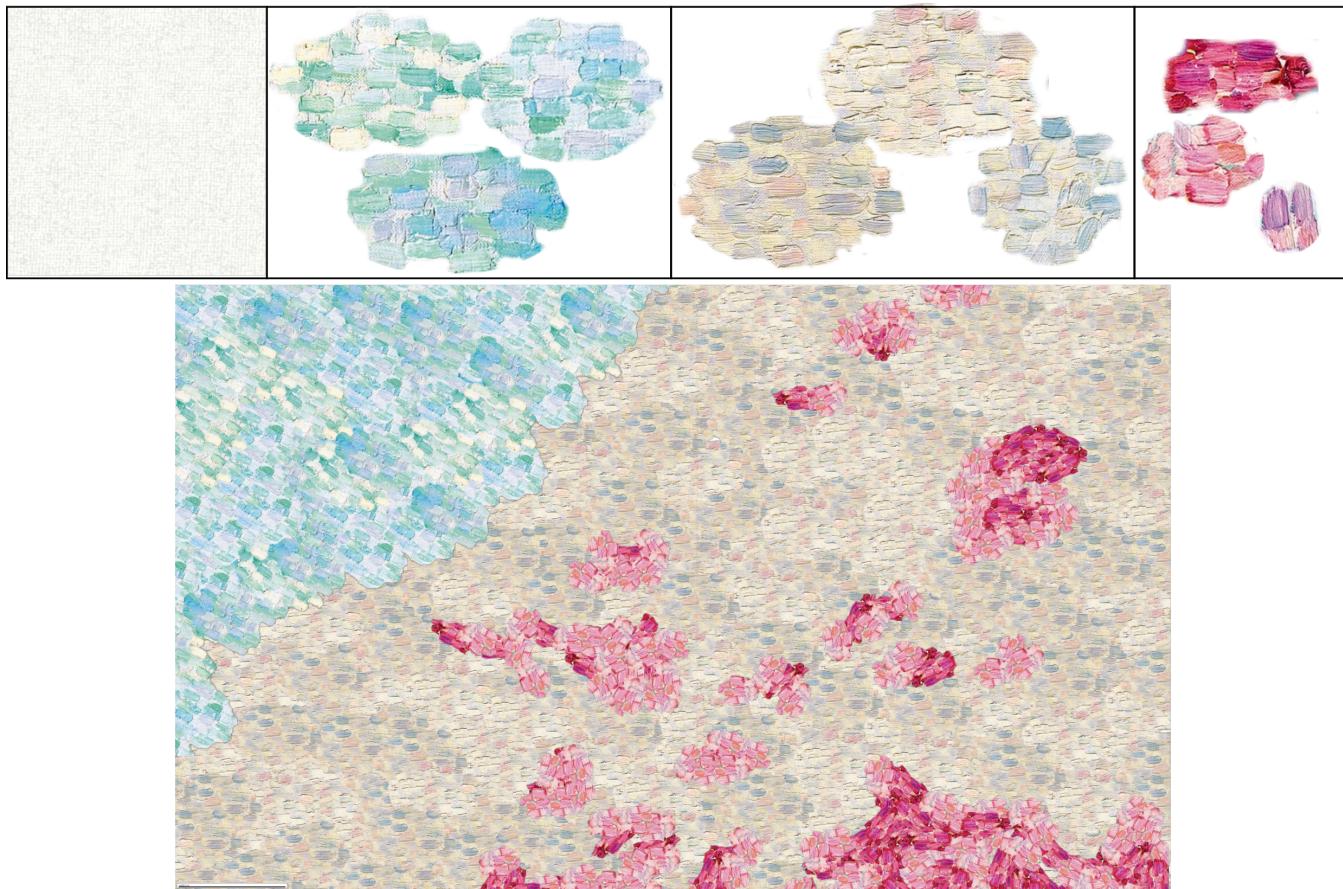


Figure 6: An example of a pointillism-like style. Only the ground, the sea and the forest is represented and is stylized with raster textures created from patches extracted from a painting by Paul Signac [Sig13].

Roughness effect

```

1 # Controlled mapper: fine scale hatching
2 def hatch_face(density_field, orient_field):
3     hatches_spacing = linear_stripes_cycle(density_field)(face)
4     angle = orient_field(Centroid(face))
5     hatchlines = StripesProperties(angle, hatches_spacing)
6     SetEdgeLabels(hatchlines, "hatches")
7     hatchlines_part = StripesPartition(hatchlines)
8 # Noise mappers
9     randomization = 0.5
10    def jitter(vertex):
11        hatch_direction = Point(cos(angle), sin(angle))
12        ortho_direction = Point(-sin(angle), cos(angle))
13        random_x = Random(vertex, -hatches_spacing * randomization / 2,
14                           hatches_spacing * randomization / 2, 0)
15        random_y = Random(vertex, -hatches_length * randomization / 2,
16                           hatches_length * randomization / 2, 1)
17        location_randomization = ortho_direction * random_x +
18            hatch_direction * random_y
19        return Location(vertex) + location_randomization
20    perturbate_edge = lambda edge: MatchPoints(ToCurve(edge), SourceVertex(
21        edge), TargetVertex(edge), jitter(SourceVertex(edge)), jitter(
22        TargetVertex(edge)))
23    def is_part_of_comb(edge):
24        if HasLabel(edge, "real_hatches"):
25            return False
26        source_is_bot = False
27        target_is_bot = False
28        for e in IncidentEdges(SourceVertex(edge)):
29            if HasLabel(e, "real_hatches") and SourceVertex(e) ==
30                TargetVertex(e):
31                source_is_bot = True
32                break
33        for e in IncidentEdges(TargetVertex(edge)):
34            if HasLabel(e, "real_hatches") and TargetVertex(e) ==
35                TargetVertex(e):
36                target_is_bot = True
37                break
38        return source_is_bot and target_is_bot
39    def map_to_comb(edge):
40        return perturbate_edge(edge) if is_part_of_comb(edge) else Nothing()
41 # Mapping operator
42    randhatches = MapToEdges(map_to_comb, hatchlines)
43    return randhatches(face)
44 # Controlled arrangement
45    def roughness_texture(density_field, orient_field):
46        espacement_min = 4.0
47        espacement_max = 8.0
48        hatches_length = 15.0 #20.0
49        def out_arrangement(face):
50            # Coarse-scale grid partition
51            angle = orient_field(Centroid(face))
52            grid_spacing = 2 * hatchspacing_max
53            linesA = StripesProperties(angle, grid_spacing)
54            SetEdgeLabels(linesA, "grid")
55            linesB = StripesProperties(angle + pi / 2.0, hatches_length)
56            SetEdgeLabels(linesB, "grid")
57            grid = GridPartition(linesA, linesB, KEEP_OUTSIDE)
58            # Noise mappers
59            randomization = 1.0
60            def jitter_grid(vertex):
61                hatch_direction = Point(cos(angle), sin(angle))
62                ortho_direction = Point(-sin(angle), cos(angle))
63                random_x = Random(vertex, -grid_spacing * randomization / 2,
64                                   grid_spacing * randomization / 2, 0)
65                random_y = Random(vertex, -hatches_length * randomization / 2,
66                                   hatches_length * randomization / 2, 1)
67                location_randomization = ortho_direction * random_x +
68                    hatch_direction * random_y
69                return Location(vertex) + location_randomization
70            perturbate_grid_edge = lambda edge: MatchPoints(ToCurve(edge),
71                SourceVertex(edge), TargetVertex(edge), jitter_grid(
72                    SourceVertex(edge)), jitter_grid(TargetVertex(edge))))
73            randomize_grid = lambda edge: perturbate_edge(edge) if HasLabel(
74                edge, "hatches") else Nothing()
75            # Mapping to coarse noisy grid
76            randgrid = MapToEdges(randomize_grid, grid)
77            # Mapping to fine hatching
78            return MapToFaces(hatch_face(density_field, orient_field),
79                             randgrid)(face)
80        return out_arrangement
81 # Export
82 ExportSVG(roughness_texture(densities, orientations), p)

```



Finer ridge hatching

```

1 # Controlled arrangement
2 def finer_ridge_hatching(density_field, orient_field):
3     peakshape1 = ImportLineSVG("data/montagne/pls1.svg")
4     tailshape1 = ImportLineSVG("data/montagne/ts1.svg")
5     espacement_pics = 20.0
6     longueur_pics = 20.0
7     longueur_queues = 20.0
8     def out_arrangement(face):
9         # Stripes supporting the hatching
10        angle = orient_field(Centroid(face))
11        linesA = StripesProperties(angle, espacement_pics)
12        SetEdgeLabels(linesA, "peak_line")
13        stripes = StripesPartition(linesA)
14        # Determine if this edge receives hatching
15        def must_provide_coulis(e):
16            other_face = LeftFace(e) if face == RightFace(e) else
17                RightFace(e)
18            d1 = density_field(Centroid(face))
19            d2 = density_field(Centroid(other_face))
20            return d1 < 0.5 and d1 < d2
21        # Mapper: keep only faces adjacent to ridges
22        def keep_ridge(miniface):
23            border_found = False
24            for e in IncidentEdges(miniface):
25                if HasLabel(e, "crete"):
26                    border_found = True
27                if not border_found:
28                    return Nothing()
29            return Scale(Contour(miniface), 0.98)
30        def generate_fill_shape(vertex):
31            if not HasLabel(IncidentEdges(vertex), "crete"):
32                return Nothing()
33            peak_source = PointLabeled(peakshape1, "start")
34            peak_target = PointLabeled(peakshape1, "end")
35            tail_source = PointLabeled(tailshape1, "start")
36            tail_target = PointLabeled(tailshape1, "end")
37            normalized_orient=Point(cos(angle),sin(angle))
38            peak = MatchPoints(peakshape1,peak_source,peak_target,Location
39                (vertex),Location(vertex)+normalized_orient*
40                longueur_pics)
41        # Find tail orientation
42        ridge_edge = UNDEFINED_EDGE
43        for e in IncidentEdges(vertex):
44            if HasLabel(e, "crete") and must_provide_coulis(e):
45                ridge_edge = e
46            other_ridge_vertex = SourceVertex(ridge_edge) if vertex ==
47                TargetVertex(ridge_edge) else TargetVertex(ridge_edge)
48            tail_orient = Location(vertex) - Location(other_ridge_vertex)
49            clockwise_orth = Point(normalized_orient.y(), -
50                normalized_orient.x())
51            if tail_orient.dot(clockwise_orth) < 0:
52                tail_orient = tail_orient * (-1.0)
53            tail_orient = tail_orient / tail_orient.length()
54            tail = MatchPoints(tailshape1, tail_source, tail_target,
55                Location(vertex), Location(vertex) + tail_orient *
56                longueur_queues)
57            res = Append(peak, tail)
58        def hatch_vertex(vertex):
59            tofill = generate_fill_shape(vertex)
60            hatches_spacing = 2.0
61            angle = orient_field(Location(vertex))
62            # StripesPartition
63            hatchlines = StripesProperties(angle, hatches_spacing)
64            hatchlines_part = StripesPartition(hatchlines)
65            # Mapping to fine hatching
66            return MapToVertices(hatch_vertex(density_field, orient_field),
67                stripes)(face)
68        return out_arrangement
69    # Export
70    ExportSVG(finer_ridge_hatching(densities, orientations), p)

```

