Discrete Texture Design Using a Programmable Approach
Operator Set and Example Programs

Hugo Loi[1], Thomas Hurtut[2], Romain Vergne[1], Joëlle Thollot[1]
[1]Inria-LJK (U. Grenoble, CNRS) [2]LIPADE - U. Paris Descartes

1 Notations

In this document we present our operators and give the pseudo-code of the programs corresponding to the images in Figure 1. Our operators manipulate *scalars* (boolean ∈ $\mathbb{B}$, integer ∈ $\mathbb{N}$, real ∈ $\mathbb{R}$), *elements* (0D points, 1D curves, 2D regions), *sets* of scalars or elements ($s(\mathbb{R})$, $s(2\text{D})$ ...), and other operators. Note that these operators are *functors*: they can be manipulated such as other variables. See below the examples of declarations, initializations, partial application or composition of operators and allowed type overrides. We use the same notation for these examples, the specification of our operator set and the pseudo-code of our programs.

Declarations, initializations, sets and operator syntax	
r: 2D	Declaration of a region
a: $\mathbb{R} = 1.1$	Initialization of a real
pts: $s(0\text{D})$	Declaration of a set of points
nbs: $s(\mathbb{N}) = \{1, 2, 3\}$	Initialization of a set of integers
n: $\mathbb{N}$ = Size(nbs)	Size of a set
nbs ≪ n	Appending a new member to a set
predicate: (2D → $\mathbb{B}$)	Declaration of a region → boolean operator
pinning: (∅ → 0D)	Declaration of an operator returning a point with no argument
translate: ((2D, 0D) → 2D)	Declaration of a multiple-argument operator
shaping: (0D → 2D) = translate(r)	Partial application of the previous operator
checkpoint: (0D → $\mathbb{B}$) = predicate ∘ shaping	Operator composition
regions: $s(2\text{D})$ = shaping(pts)	Application of an operator to a set of inputs
inv: (2D → $\mathbb{B}$) = Not ∘ predicate	Boolean operators are handled as our other operators
if (And(n⩽5, **true**)) **then** n= 6	We use classic comparison, constants and control structures

Type overrides	
transform: (2D → 2D) = translate(pinning)	(∅ → Type) operators can be used instead of Type variables
throwing: (0D → 0D) = pinning	(∅ → Type2) operators can be used instead of (Type1 → Type2)
pinning = pts	s(Type) can be used instead of a (∅ → Type) operator (but not conversely)
pin_all: (Elt → 0D)	Declaration of an operator defined for each input type along element types
tran_all: (Elt → Elt)	With this notation, output type is the same as input

2 Operators

Construction operators	
RandomPinning: (2D → (∅ → 0D))	Pins random points in the given region
RegularPinning: ((2D, $\mathbb{N}$, $\mathbb{N}$) → $s(0\text{D})$)	Pins points on a grid covering the given region
RegularPinning1D: ((1D, $\mathbb{R}$) → $s(0\text{D})$)	Pins regularly points on a curve with a given density
Centroid: (2D → 0D)	Pins the centroid of a region
TangentFlowField: ($s(2\text{D})$ → (0D → 0D))	Computes points whose coordinates interpolate tangents of input regions
Contour: (2D → 1D)	Returns the contour of a region
Interpolation: ($s(0\text{D})$ → 1D)	Returns a curve interpolating the given set of points
VoronoiCells: ($s(2\text{D})$ → $s(2\text{D})$)	Returns the Voronoï cells of the given region set (see technical details)
ApplyThickness: ((1D, ($\mathbb{R}$ → $\mathbb{R}$)) → 2D)	Changes a curve into a region given a thickness function

Transformation operators	
Translation: ((Elt, 0D) → Elt)	Translates any element at the given location
Rotation: ((Elt, $\mathbb{R}$) → Elt)	Applies a rotation of the given angle to any element
Scale: ((Elt, $\mathbb{R}$) → Elt)	Applies a homothetic transformation to any element

Scalar operators	
CountUpTo: ($\mathbb{N}$ → (∅ → $\mathbb{B}$))	Predicate returning **true** n times, and then **false**
Overlap: (($s(2\text{D})$, Elt) → $\mathbb{B}$)	Computes an overlap test between any element and a set of regions
RandomR: (($\mathbb{R}$,$\mathbb{R}$) → (∅ → $\mathbb{R}$))	Random number generator given a real range
Distance: ((Elt, Elt) → $\mathbb{R}$)	Distance between two elements
MinimalDistance: (($s(2\text{D})$, Elt, $\mathbb{R}$) → $\mathbb{B}$)	Computes a minimal distance test between any element and a set of regions

Input / Output	
Image: ($\varnothing \to$ 2D)	The boundary of the image
BuiltinHatch: ($\varnothing \to$ 2D)	Built-in small hatch-shaped region
BuiltinRectangle: ($\varnothing \to$ 2D)	Built-in large rectangle-shaped region
BuiltInCircle: ($\varnothing \to$ 2D)	Built-in small circle-shaped region
User(Type) or User($A \to B$)	User-specified Type variable or ($A \to B$) operator
Display: (s(2D) $\to \varnothing$)	Display operator for tests (see technical details)

3 Programs for Example Images

3.1 Classic Distribution Algorithms

Greedy Distribution Algorithm

greedy_distribution: (
    (out: s(2D), loop_condition: ($\varnothing \to \mathbb{B}$), pinning: ($\varnothing \to$ 0D),
    shaping: (0D $\to$ 2D), checking: (2D $\to \mathbb{B}$)) $\to \varnothing$)

out = {}
while (loop_condition) **do** {
    p: 0D = pinning
    r: 2D = shaping(p)
    if (checking(r)) **then** out $\ll$ r }

Region-Based Relaxation Algorithm

relaxation: (
    (in: s(2D), out: s(2D), loop_condition: ($\varnothing \to \mathbb{B}$),
    reshaping: (s(2D) $\to s$(2D)), repinning: (2D $\to$ 0D) $\to \varnothing$)

out = in
while (loop_condition) **do** {
    s: s(2D) = reshaping(out)
    pts: s(0D) = repinning(s)
    shaping: (0D $\to$ 2D) = Translation(s)
    out = {}
    for (p **in** pts) **do** out $\ll$ shaping(p) }

3.2 Figure 1a -1d

Figure 1a – Classic anisotropic dart throwing

hatches: s(2D)
greedy_distribution(
    hatches, CountUpTo(1000), RandomPinning(image),
    Translation(BuiltinHatch), Not $\circ$ Overlap (hatches))
Display(hatches)

Figure 1d – User-drawn region and variable shaping

shapes: s(2D)
shaping: (0D $\to$ 2D) = Translation (
    Rotation(RandomR$(0, 2\pi)$) $\circ$
    Scale(User($\varnothing \to \mathbb{R}$)) $\circ$
    User(2D))
greedy_distribution(
    shapes, CountUpTo(4000), RandomPinning(image),
    shaping, Not $\circ$ Overlap(shapes))
Display(shapes)

Figure 1b – Constrained dart throwing

rectangles: s(2D)
greedy_distribution(
    rectangles, CountUpTo(25), RegularPinning(image, 5, 5),
    Translation(BuiltinRectangle), Not $\circ$ Overlap(rectangles))
hatches: s(2D)
greedy_distribution(
    hatches, CountUpTo(1000), RandomPinning(image),
    Translation(BuiltinHatch),
    And (Not $\circ$ Overlap(hatches), Overlap(rectangles)))
Display(hatches)

Figure 1c – Composition of 1b with following instructions

hatches_orth: s(2D)
greedy_distribution(
    hatches_orth, CountUpTo(3000), RandomPinning(image),
    Translation (Rotation($\frac{\pi}{2}$) $\circ$ BuiltinHatch),
    And (Not $\circ$ Overlap(hatches), Not $\circ$ Overlap(hacthes_orth)))
Display(hatches_orth)

3.3 Figure 1e -1g

Circle Distribution – A routine used in Figure 1e -1g

circle_distribution: ($\varnothing \to s$(2D))

circles: s(2D)
greedy_distribution (
    circles, CountUpTo(User($\mathbb{N}$)), RandomPinning(image),
    Translation(BuiltInCircle), Not $\circ$ Overlap(circles))
relaxed_circles: s(2D)
relaxation (
    circles, relaxed_circles, CountUpTo(User($\mathbb{N}$)),
    VoronoiCells, Centroid)
return relaxed_circles

Stream Lines – A routine used in Figure 1f -1g

stream_line: (
    (start: 0D, flow_field: (0D $\to$ 0D), check: (0D $\to \mathbb{B}$),
    length: $\mathbb{R}$) $\to$ 2D)

pts: s(0D)
curr_pt: 0D = start
curr_length: $\mathbb{R} = 0.0$
condition: ($\varnothing \to \mathbb{B}$) = And(curr_length$\leqslant$length, check(curr_pt))
next: (0D $\to$ 0D) = Translate(flow_field)
while (condition) **do** {
    pts $\ll$ curr_pt
    next_pt: 0D = next(curr_pt)
    curr_length = curr_length + Distance(curr_pt, next_pt)
    curr_pt = next_pt }
return ApplyThickness(User($\mathbb{R} \to \mathbb{R}$)) $\circ$ Interpolation(pts)

```
Figure 1e — Texturing with transformation of Voronoï cells
  shapes: s(2D) = VoronoiCells ∘ circle_distribution
  shapes = Scale(shapes, User(∅ → ℝ))
  Display(shapes)
```

```
Figure 1f — Circle distribution and stream lines
  relaxed_circles: s(2D) = circle_distribution
  Display(relaxed_circles)
  stream_lines: s(2D)
  shaping: (0D → 2D) = stream_line(
      TangentFlowField(relaxed_circles),
      And(
          Not ∘ Overlap(circles),
          Not ∘ Overlap(stream_lines)),
      User(ℝ))
  greedy_distribution(
      stream_lines, CountUpTo(2000), RandomPinning(image),
      shaping, And(
          Not ∘ Overlap(circles),
          Not ∘ Overlap(stream_lines)) )
  Display(stream_lines)
```

```
Figure 1g — Interlocked 1D distribution of stream lines
  relaxed_circles: s(2D) = circle_distribution
  stream_lines: s(2D)
  pinning: s(0D) =
      RegularPinning1D(Contour(relaxed_circles), User(ℝ))
  shaping: (0D → 2D) = stream_line(
      Rotation(π/6) ∘ TangentFlowField(relaxed_circles),
      Not ∘ Overlap(stream_lines), User(ℝ))
  greedy_distribution(
      stream_lines, CountUpTo(Size(pinning)),
      pinning, shaping,
      Not ∘ MinimalDistance(stream_lines, User(∅ → ℝ)) )
  Display(stream_lines)

  scnd_strlines: s(2D)
  pinning = RegularPinning1D(Contour(stream_lines), User(ℝ))
  shaping = stream_line(
      TangentFlowField(relaxed_circles),
      Not ∘ Overlap(stream_lines), User(ℝ))
  greedy_distribution(
      stream_lines, CountUpTo(Size(pinning)),
      pinning, shaping,
      Not ∘ MinimalDistance(scnd_strlines, User(∅ → ℝ)) )
  Display(scnd_strlines)
```

4 Technical details

- We show our operator set and example programs in pseudo-code with a notation specified at the beginning of the document. In practice, our operator set is implemented as a C++ library and each operator is a separate functor class. Thus, the programs are C++ functions and each line of pseudo-code in this document is implemented with a line of C++ source code.
- 1D and 2D types depend on the underlying implementation. In the current version, we compute curves as non self-intersecting polylines and regions as non self-intersecting polygons without holes.
- We implement the computation of Voronoï cells from any region set with the method from [Hoff et al. 2000] and polygon fitting.
- Our system produces discrete textures that can be saved as an SVG file or rendered directly with a very simple style (Display operator). Stylizing such discrete textures is out of the scope of our contribution and is a very interesting avenue for future works.

References

HOFF, III, K. E., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 2000. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 16th annual Symposium on Computational Geometry, Clear Water Bay, Kowloon, Hong Kong*, ACM, New York, NY, USA, SCG '00, 375–376.