Natural Computing manuscript No.
(will be inserted by the editor)

A parallel data-structure for modular programming of triangulated computing media.

Frédéric Gruau, gruau@lisn.fr

February 24, 2022

Abstract Our long-term project involves performing general-purpose computation on 2D amorphous computing media, which consist of arbitrary many, small, identical processing elements that are homogeneously spread in 2D Euclidian space, and that communicate locally in space. While the minimal assumptions on hardware provide the fascinating perspective of arbitrary large computing power, they also make programming notoriously difficult. Furthermore, our project involves simulating objects extended in 2D-space, called "blobs". Maintaining the connectedness of blobs while they move in space adds another layer of difficulty since it demands to process the topology of 2D space. This paper describes a new parallel data structure that can simplify the programming task, in this context.

In computer graphics, processing related to 2D topology is performed by using triangle meshes. We consider synchronous media whose underlying network is also a triangle mesh. Our data structure, derived from computer graphics, is anchored on that mesh so that its operations can be compiled on the medium. More precisely, our compiler produces a circuit of logic gates, which enables a high-performance simulation, in the case of crystalline media (Cellular Automata). We demonstrate the expressiveness of the data structure's operation by using an incremental and modular programming style. We program, first small, then larger building-block functions, and re-use them. Blobs are implemented and re-used to compute the Voronoï diagram.

What is the scope of the data-structure? This poses the question of whether there exists a universal set of primitives able to program any processing specified only in terms of 2D-geometry.

Laboratoire Interdisciplinaire des Sciences du Numérique, Université Paris Saclay, Orsay, France.

Keywords Computing medium; Cellular Automata; data structure; triangle mesh; blob; Voronoï diagram;

1 Introduction

Our long-term project involves performing general-purpose computation on computing media [10]. We consider 2D computing media consisting of billions of small identical Processing Elements (PE). The network linking PEs is defined by an embedding in 2D space, with two properties:

1. Euclidian sampling: PEs are homogeneously spread in 2D Euclidian space.
2. Locality: a PE can communicate only with its closest neighbors in space.

We restrict our scope to synchronous media. During one synchronous cycle, each PE makes a bounded amount of computation and exchanges a bounded amount of data with its neighbors. Synchronicity is usually considered difficult to obtain. However, the two hypotheses defining a computing medium make it feasible to also bound the distance, and therefore the communication time between any two neighbors, in which case synchronicity can be obtained just by letting each PE waits a fixed bounded time.

If no additional hypothesis is put on the network, the medium is called amorphous [1]. We consider also crystalline media, where the placement in 2D is done regularly according to a lattice structure such as the 2D grid or the hexagonal grid. Crystalline media have been extensively studied, and they are commonly called Cellular Automata [6] (CA). From a computational point of view, the striking feature of a computing medium is its arbitrary large size. Locality combined with homogeneity ensures that communication can always occur

in constant time, without violating the laws of physics, no matter how large the medium is. Furthermore, for the amorphous case, PEs are spread on a 2D surface without having to worry about the exact location, like smartdust [13]. The crystalline case is however also worthy of interest because it can be simulated more efficiently.

Arbitrary scalability is a fascinating property because it means the feasibility of parallel hardware with arbitrary large power. However, programming computing media is a difficult task. Furthermore, our project involves simulating *large* objects extended in space. This demands new primitives that are able to process topological information.

Our proposal is to restrict the range of topololgy on the network linking the PEs: we assume it is *triangulated*. For amorphous media, a premiliminary step of Delaunay triangulation must be conducted. Another option is to build a *combinatorial Delaunay graph* as a second layer based on the hop counts between PEs [25]. For crystalline media, the hexagonal CA is triangulated.

Thanks to triangulation, information can be grouped into fields located more precisely in space, and processed using operators derived from standard data structure used in image processing. The programming task becomes highly simplified: 1- Fields enable a modular programming style 2- Operators on fields can process topological information and handle large objects, and 3- circuits can be compiled and simulated with high-performance..

Fields. First, programming computing media is facilitated by using abstraction to directly program the ensemble of PEs. This is called *aggregate programming* in [5]. This implies rooting the programming layer on a parallel data-structure:

1. The elementary data are fields of values [3] over the whole medium. Each PE holds a particular local value corresponding to its location.
2. Primitive operations are elementary functions that compute new fields.

Some of the fields are stored in the PE's memory. We call these "mutable" because their value can be updated. Each mutable field is associated with an updating function. A synchronous medium computes by iterating those updating functions at each time step. As an illustrating example, if mutable fields represent physical states, and the updating function models a discretized version of some physical law, the parallel execution of all the medium's PEs can simulate the physics throughout the sampled space. In general, simulating physics is what computing media are naturally proficient at, due to Euclidian sampling. Fluid dynamics are translated into simple CA local rules in [7]. Rauch modeled wave propagation on an amorphous medium [19], thus demonstrating that the crystalline property is not mandatory for doing physics.

The key advantage brought by aggregate programming is *modularity*. A program is designed by combining operators to produce increasingly complex field-functions. Thus, a tool-box of generic "spatial functions" is developed. For example, a useful tool computes a distance field from a boolean field representing a set of sources. In [8], such a distance field is used to move particles away or towards the sources, or parallel to the sources.

Large objects. Second, our project simulates large objects extended in space and moving in space. We call them "blobs" to convey the idea that they are "shapeless" since no constraints are placed a priori on the shape. Many approaches have been proposed for a programming layer of computing media [4]. However, very few works, have considered large objects. The only example we found is polymers in [21]. However, polymers are 1D objects; moreover, only a crystalline medium was considered in this study. In general, blobs are 2D-objects. The support of a blob is encoded by a boolean field representing the presence flag. Blobs move in space by modifying this flag on their border. However, they should not be disrupted; nor should they merge. Each PE must compute whether it can reset (resp. set) the presence flag, without causing disruption (resp. merging). This *blob-predicate* derives all the structures used in our project, such as implementing membranes or establishing a bond between two blobs, as proved in [12]. Blob is a building block of generic interest. In this article, we show how using blobs enables us to easily compute the Voronoï diagram.

Simulating gas or fluid dynamics involves only particles, each one hosted on a single PE. The real physics can be translated in a time-effective way, using only simple primitive exchanging information between pairs of adjacent PEs. In contrast, implementing blobs involves interaction between multiple PEs. and demands more elaborate primitives. Simulating real physics of solid-state objects would be too time-consuming. Instead, we resort to "artificial physics". We design elementary operations inspired by computer graphics. They are defined for *triangulated computing media* whose underlying network is a triangle mesh. They can process fields representing the topology of 2D-space, to match the topological nature of the blob-predicate. Since the primitive types and the elementary operations are both rooted on the triangle mesh, we call the resulting data

structure the *triangle data-structure* or more shortly the *triangle type*.

Performance. Third, our project involves complex update functions. In the third interactive simulation filmed in [9] each PE has 77 bits of state and 13878 binary logic gates. This should be contrasted with usual CA rules such as the game of life, with a single bit of state, and a transition computed with a dozen gates. For this unusual level of complexity, the direct naïve method of updating each PE one by one prevents interactive simulation. High-performance is obtained by first compiling a program using triangle-types into a circuit of logic gates, and then optimizing the circuit simulation.

Outline. We present the triangle-type, its elementary fields, and operations. We then develop a small library of macro-operations, for computation and communication. Next, we utilize this library to compute the blob-predicate, which then allows us to compute the Voronoï diagram. Finally, we explain how to compile into circuits, and obtain high-performance. We illustrate the general applicability, with a complementary example of triangle-type use.

2 The triangle type: a parallel data-structure based on triangle mesh

We need to process large objects represented on arbitrary many PEs of a computing medium. The intuition behind our data representation is to compute "*in 2D-space, about 2D-space*". A feature of a large object has a natural location in 2D-space. For example, an object's frontier lies between adjacent PEs, i.e. on the edges of the network. An edge links the two PEs that need to be consulted to compute whether there is a frontier or not. One of the two should be in, and the other out. The frontier should ideally be computed "on edges". We thus imagine that there are *virtual* PEs on the edges themselves, which will compute "in edges" whatever is naturally computed "about edges". Likewise, the edge is also the ideal place to store the frontier, because the frontier is likely to be used by the two linked PEs that are nearby. By assuming that the network linking PEs is a triangle mesh, we can define virtual PEs on many loci other than just vertices and edges, and generalize this principle of computing "*in 2D-space, about 2D-space*". We will use 9 types of loci.

Triangle meshes. A triangle mesh is equivalently a maximal planar graph. Maximal means no edges can be added, hence all the faces are triangles. We consider

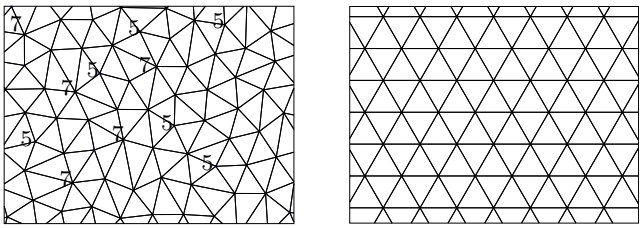


Fig. 1 Two examples of 2D triangle mesh (a) amorphous isotropic (b) christaline - hexagonal

two types of triangle meshes shown in fig. 1 representing respectively amorphous and crystalline computing media.

For amorphous media, a simple and often used PE distribution in 2D is the *"Poisson-disk"* sampling: the location of each new PE is chosen with a uniform probability, but the PE is discarded if there are already other PEs nearby, i.e. within a disk of a given radius. For our simulation, instead of discarding, we used the Furthest Point Optimization (FPO) algorithm [20] which produces more homogeneous and isotropic distribution. A Delaunay triangulation builds the triangle mesh shown in fig. 1 (a). The improved quality due to FPO causes the hop-count distance to become a good approximation of the geometric distance. This, in turn, enables us to compute spatial features with accuracy. For example, the hop-count discrete Voronoï Diagram (VD) computed in this paper approximates the real continuous VD.

For crystalline media, the usual architecture with CA is the 2D square grid. However, it is not a maximal planar graph, neither with the Von Neumann neighborhood (4 neighbors, north, south, east, west) which is not maximal nor with the Moore neighborhood (8 neighbors including the diagonals) which is not planar. The hexagonal grid in fig. 1 (b) is the canonical maximal planar graph. This paper is an extended version of [11], which considered only this lattice.

The DCEL data-structure. Triangle meshes are widely used in computer graphics due to their simplicity. They are usually represented by a Doubly-Connected Edge List (DCEL) data structure [23], due to the capability to find the neighbors of a vertex, face, or edge in constant time. As fig. 2 shows, an edge in the DCEL connects two vertices and is defined by a pair of half-edges, each one being a twin of the other. A face is then defined by a cycle of half-edges at the boundary of the face, in counter-clockwise order. The next and previous pointers implement another cycle of half-edges around vertices.

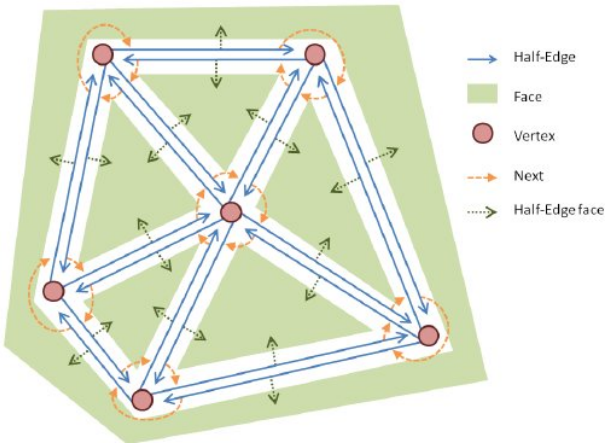


Fig. 2 DCEL data-structure. Edges are represented by pairs of parallel blue arrows, vertices by red circles, and faces by green polygons

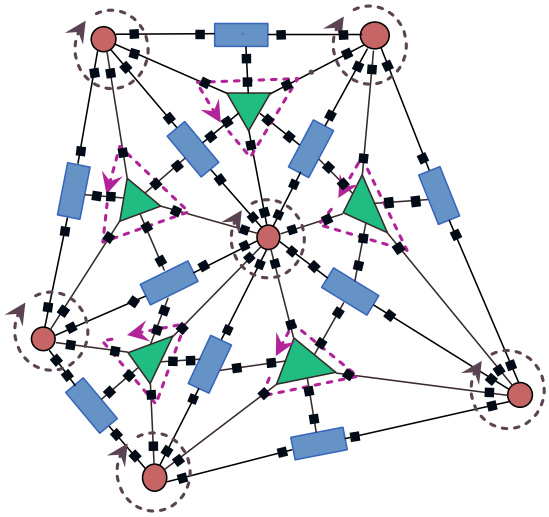


Fig. 3 The nine locus. s-locus are represented by circles for vertices, triangles for faces, and rectangles for edges; t-locus are little black squares. The clockwise (resp. counter-clockwise) dotted arrows highlight the interleaving between the fV and eV (resp. vF and eF) t-locus.

The nine locus of triangle types. We consider synchronous computing media whose PE network is a triangle mesh. We sub-sample 9 specific sets of points on the triangle mesh, as shown in fig 3. These 9 loci will enable us to use 9 kinds of fields and operations between them, reproducing the same functionalities offered by the DCEL, but in a data-parallel framework.

First, we consider three simplicial loci (s-locus) which include vertices (V), edge's middles (E), and barycenters of faces (F). Those points share the following simplicial neighborhood relationship:

- An edge is adjacent to two vertices and two faces.
- A face is adjacent to three edges and three vertices.

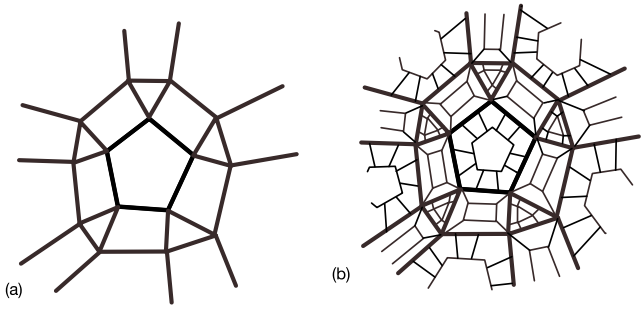


Fig. 4 Representing fields: (a) VEF tiles for simplicial fields alone. Edges are rectangle, faces are triangles, vertices are polygons (b) tiles for transfer fields are a subtiling.

The hop-count associated with this neighborhood defines a distance between the simplicial points called the *VEF-distance*. Two connected vertices are at VEF-distance 2 because an edge separates them.

Second, we introduce six transfer locus (t-locus) which implement communication between s-locus. In between each pair of adjacent points in (V,F), (resp. (V,E) , (E,F)), we insert a point of the t-locus "fV" and another point of the t-locus "vF" , (resp. "eV" and "vE", "eF" and "fE"). The two t-locus of each pair are called *companion*. The upper (resp. lower) case designates the nearest (resp. furthest) s-locus. The nearest s-locus is called the *father*. Two t-locus with identical fathers such as eV and fV are called *brothers*. The concept of brother and companion is extended from locus to the points making a locus. As highlighted by dotted arrows in fig. 3, the fV, and eV (resp eF and vF) points alternate as we go around the common vertex (resp. face) father. This interleaving between brother points transcribes the DCEL cycle of half-edges around a vertex (resp. face).

Density of locus. Only vertices correspond to real PEs, the other points are handled as if they were virtual PEs. Their density is defined as the number of virtual PEs per real PE. Let the vertex (resp edge, face) count be v, (resp. e, f). A maximal planar graph verifies $2e = 3f$, as can be derived by taking the sum over every face of the number of edges in each face which is 3. We also have $v - e + f = 2$, (Euler's formula) hence $f = 2v - 4, e = 3v - 6$. The density of V is 1, by definition. By neglecting the additive constant, we find that the density of the E (resp. F) s-locus is 3 (resp. 2). The density of all the t-loci is the same, since companion t-loci are facing each other, and brother t-loci are interleaved. By looking at the eF locus, we find that it is 6.

Fields. Fields are functions from virtual PEs of one of the 9 loci, towards a domain representing a scalar type. The type of a field combines the scalar type with the

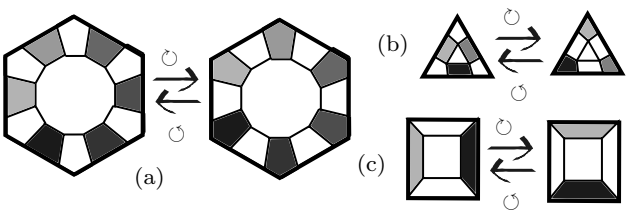


Fig. 5 Rotation between brother t-locus, applied on an integer field, values are encoded by gray tones. (a) between eV and fV (b) between eF and vF, (c) between vE and fE.

locus name and is called a *triangle-type*. For example a boolV is a function from the V locus towards $\{0, 1\}$. We also call it a "vertex boolean field". We use integer fields as well, with a small number of bits to minimize computation cost. Usually, 2 or 3 is sufficient. For example, a vertex field of integers on three bits has type int3V. Such a field is used in [14] to compute distance to a moving target. The density of a triangle-type is the number of bits needed to store a field, per real-PE. It is equal to the density of the locus, multiplied by the number of bits needed for the scalar. Most of the fields that we compute are simplicial fields. To represent only simplicial fields, we draw the Voronoï Diagram of the three VEF loci taken together, as shown in fig. 4 (a) and color the tiles according to the corresponding field values.

Sometimes, when we decompose a field function operator by operator, and intermediate steps are represented by transfer fields. We use the sub-tilling shown in fig. 4 (b): a central sub-tile represent an s-locus, while the peripheral sub-tile represent t-locus.

Primitive operations. Primitive operations on fields can move, duplicate or combine bits of data, using the locus neighborhood derived from the triangle mesh (fig. 3). We distinguish three kinds of operations:

1. One-to-one communication denoted with arrows, from one t-locus:
   - to its companion t-locus: *transfer* noted $\uparrow$
   - to its brother t-locus: *rotation* noted $\circlearrowleft$ or $\circlearrowright$.
2. Multi-cast from one s-locus to one of its son's t-locus, noted $*^v$, $*^e$, or $*^f$.
3. Reduction from one t-locus to its father s-locus, such as $/\wedge$, $/\vee$, $/\oplus$, $/+$,

Rotation is possible because two brother s-locus are interleaved. There exist two rotations shown in fig 5, clockwise, and the reciprocal counter-clockwise. When the father is a vertex (resp. a face) the rotation corresponds to the traversal of half-edge cycles in the DCEL, around a vertex (resp. face).

A multi-cast specifies an upper script indicating the one of the two possible son towards which information

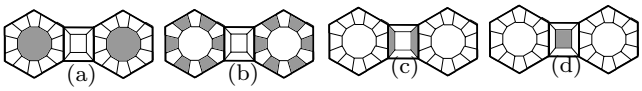


Fig. 6 Decomposition of $x \mapsto \forall^E(x)$. (a) Initial boolV. (b) Multicast to a booleV (c) Transfer to a boolvE (d) Reduction to a boolE

is sent. For example $*^e$ multi-casts bits from each vertex to its adjacent eV points, or from each face to its three adjacent eF points. When a function like $x \mapsto *^e x$ as a domain including distinct locus (here V and F), it is said to be *overloaded*. We often use overloading to minimize the set of introduced symbols.

A reduction applies a commutative, associative operation on the neighbors such as the logical conjunction or the addition. This operation is specified after a slash. For example, since the logical binary AND is noted $\wedge$, the conjunction reduction is noted $/\wedge$.

New fields can also be computed using "non-spatial" operations, i.e. operations taking place within the PEs of a given locus such as $x \mapsto \neg x$. They apply to fields of any locus and produce a new field on the same locus.

3 A tool-box of macro-operators

First, we program functions that implement macro operator, i.e. small building-blocks of generic interest that we re-use often. We will consider macro-operators for computation and communications.

3.1 Macro-operators for computation.

A given S-locus cannot distinguish between its adjacent T-loci, therefore computation needs to be specified using reductions.

Simplicial reductions. A useful sequence of operations consists of a multi-cast, followed by a transfer and finally a reduction. Bits travel from one s-locus to another, by transiting through the two intermediate companions t-locus. It is called a *simplicial reduction* since the source and the target are both simplicial fields. For example, bits can move from V to E by passing through eV, and then vE, as shown in fig 6. We use the following notation, when the reduction is a conjunction $\wedge$, a disjunction $\vee$ or the exclusive or $\oplus$:

$$\forall^E = /^\wedge \circ \uparrow \circ *^e; \quad \exists^E = /^\vee \circ \uparrow \circ *^e; \quad \delta^E = /^\oplus \circ \uparrow \circ *^e \quad (1)$$

The upper script E indicates the target locus. We can also similarly define $\forall^V$, $\forall^F$,$\exists^V$, $\exists^F$,δ^V, δ^F. Because of overloading of $x \mapsto *^e x$, the mapping $x \mapsto \forall^E(x)$ (resp. $\exists^E, \delta^E$) is also overloaded. It can be applied to a boolV or a boolF.

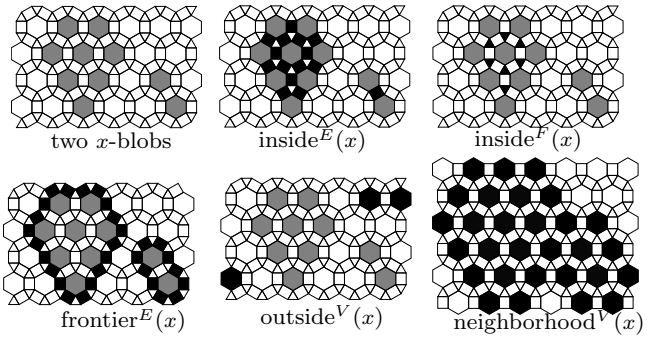


Fig. 7 Boolean Fields (in black) encoding simple features of the same two x-blobs (in gray). A boolV (resp. boolE, boolF)is represented as a set of hexagons (resp. rectangles, triangles).

Computing simple blob features with simplicial reductions. We consider spatially extended objects called *blobs* whose support spans a set of vertices encoded using a bool$V$. A single bool$V$ x can represent several distinct x-blobs. Blobs are separated by considering connected components for vertex-adjacency. Using one or two simplicial reductions, we can compute fields representing simple and useful blob features illustrated in fig. 7:

- Function $x \mapsto \text{frontier}^E(x) = \delta^E(x)$ (resp. $\text{inside}^E(x) = \forall^E(x)$, $\text{outside}^E(x) = \forall^E(\neg x)$) is the set of edges adjacent to both an empty and filled (resp. to only filled, to only empty) vertices.
  Function $x \mapsto \text{inside}^F(x) = \forall^F(x)$ is the set of faces adjacent to only filled vertices.
- Function $x \mapsto \text{inside}^V(x) = \forall^V(\text{inside}^E(x))$ (resp. $\text{outside}^V(x) = \forall^V(\text{outside}^E(x))$ is the set of filled (resp. empty) vertices surrounded by vertices in the same blob (resp. hole).
- Function $x \mapsto \text{neighborhood}^V(x) = \exists^V(\exists^E(x))$ computes the neighborhood.

Transfer reductions. By applying a reduction on the fields produced using the clockwise and anti-clockwise rotations, we obtain a second set of six reductions for a given reduction operator. It maps one t-locus to its brother. For example, with $\wedge$, the function $x \mapsto= (\circlearrowright x) \wedge (\circlearrowleft x)$ defines six new reductions denoted by $\exists$ with a subscript specifying the target locus: $\exists^{eV}$, $\exists^{vE}$, $\exists^{eF}$, $\exists^{fE}$, $\exists^{fV}$, $\exists^{vF}$.

3.2 Macro-operators for communication.

Central symmetry. Let the fan-out of a simplicial point be the fan-out of its multi-cast. It is always three (resp two) for a face (resp. an edge). For the hexagonal medium, the fan-out of vertices is equal to 6. but in the general amorphous case, it can range from 5 to 7 as shown in fig. 1. When a t-locus verifies that the fan-out f of its father s-locus is uniform, there are always $2 * f$ interleaved brothers. Thus, a central symmetry noted $\leftrightarrow$ can be defined by composing f elementary rotations, as illustrated in fig. 8. It is idempotent $\leftrightarrow \circ \leftrightarrow$=Id. It maps an eF field to a vF field and vice-versa, whereas it maps a vE (resp. fE) field to a vE (resp. fE) field.

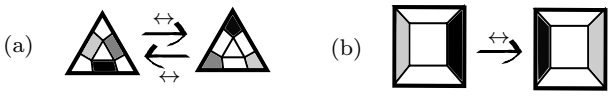


Fig. 8 Central symmetry applied on an integer field represented with gray tones. (a) between eF and vF (b) from vE to itself, or fE to itself.

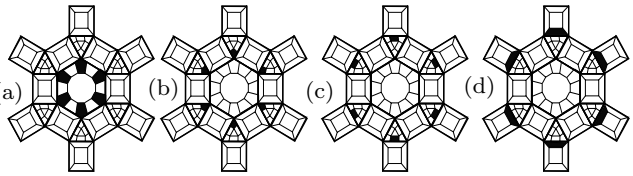


Fig. 9 Apex communication.(a) fV field (b) transfer to a vF field (c) central symmetry to a eF field(d)transfer to a fE field.

Apex neighbors. The central symmetry on faces is used to implement a composite communication called *apex*. On a triangle mesh, each edge has two distant vertices called *apex-vertices*, lying at distance 2, adjacent to the two faces next to the edge. Conversely, each vertex has also distant edges lying at distance 2, also called *apex-edges*. The function $x \mapsto \text{apex}(x) = \uparrow \circ \leftrightarrow \circ \uparrow (x)$ implements a one-to-one composite communication from a vertex tile to its apex-edge's tile, from the fV to the fE locus. The effect is illustrated in fig. 9. Data moves from the vertex to the face tiles, within the face tiles (central symmetry), and finally from the face to the edge tiles. Because of the overloading of "$\uparrow$", the apex function also implements the reciprocal transformation from an edge to its apex vertices and is idempotent: apex$\circ$apex=Id.

4 Computing the blob-predicate

We will now use our tool-box of macro-operators to program a more elaborate field functions, also of generic interest, which is the central example of this paper. It represents the *blob-predicate* which identifies points where blobs can move.

4.1 Analysis of the blob-predicate

Global meeting points. Let x be a boolV, we recall that x-blobs (resp. x-holes) are connected components of

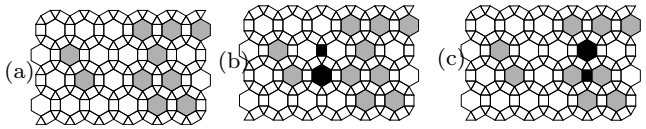


Fig. 10 Meeting-points in black, (a) a boolV x representing 2 blobs in gray, x=0 beyond the border. (b) a merge-vertex and a merge-edge (c) a div-vertex and a div-edge.

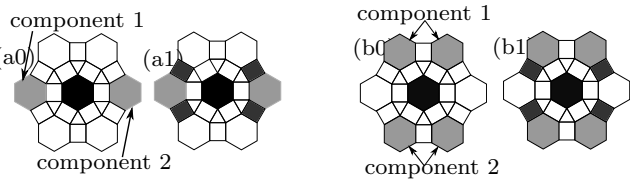


Fig. 11 Locally induced blobs (in gray) for (a) the merge-vertex, and (b) the dividing-vertex of fig. 10 (b,c). The added black rectangles in (a1,b1) are the apex edges in frontier$^E(x)$.

filled (resp. empty) vertices. Let the *vertex frontier* be the set of vertices adjacent to the E-frontier. We have: $x \mapsto \text{frontier}^V(x) = \exists^V(\text{frontier}^E(x))$. It is decomposed into an inside frontier $\text{frontier}^V_{\text{in}}(x) = x \wedge \text{frontier}^V(x)$, and an outside frontier: $\text{frontier}^V_{\text{out}}(x) = \neg x \wedge \text{frontier}^V(x)$. The inside and outside frontier can also be defined for holes, which is simply the negative field $\neg(x)$. The inside frontier of blobs is the outside frontier of holes and vice versa. Blobs move on the medium by emptying (resp. filling) a given vertex on the inside (resp. outside) frontier. However, such a move can cause a division (resp. a merge) of supports. To prevent division or merge we must identify which filling and which emptying is OK. We need to distinguish two cases, depending on whether the gap between the two blobs (resp. holes) that could potentially merge is one vertex or two vertices wide. Divide (resp. merge) can happen:

1. When emptying (resp. filling) a single vertex.
2. When simultaneously emptying (resp. filling) two adjacent vertices.

The first case is addressed by computing two boolV functions $x \mapsto \text{div}^V(x)$ and $x \mapsto \text{merge}^V(x)$. In the second case, what we need to identify is edges whose two vertices on each side are not allowed to simultaneously empty (resp. fill). We call the corresponding functions $x \mapsto \text{div}^E(x)$ and $x \mapsto \text{merge}^E(x)$. They compute a boolE from a boolV. Those four functions are illustrated in fig. 10. Merge-points can be characterized as follows:

Definition 1 Let x be a bool$V$ representing blobs. $\text{merge}^V(x)$ is true for empty vertices adjacent to two distinct $x-$blobs. $\text{merge}^E(x)$ is true for an edge e , if (i) e is adjacent to the out-frontier^V of two x-blobs (ii) $\text{merge}^V(x)$ is false for the vertices adjacent to e.

Condition (ii) means that simultaneous filling on both sides is a requirement. More precisely, if filling just one side of the edge causes a merge, then that edge is not a merge-edge. Divide-points can be obtained by replacing "blob" with "hole" since dividing a blob is equivalent to merging two holes. We regroup the four functions into two functions computing *meeting-points*, distinguishing only vertice-meetings from edge-meetings:

$$\begin{aligned} \text{meet}^V(x) &= \text{merge}^V(x) \vee div^V(x); \\ \text{meet}^E(x) &= \text{merge}^E(x) \vee div^E(x) \end{aligned} \tag{2}$$

Local meeting-points. What we just defined is global meeting-points, but those cannot be computed in bounded time, because blobs can be arbitrarily big, and with a non-convex shape. To obtain a definition of local meeting-points, we rewrite definition 1; we replace "blob" with "locally-induced blobs" which are obtained by intersecting blobs with a ball of a small radius, centered on the potential studied meeting-point. Local meeting-points are not necessarily global. Locally, one can see two filled components that do not touch. But they may meet if we look much further. What really matters though, is that global meeting-points are always local meeting-points, so detecting local meeting-points is an overkill, but works out, for our purpose of preserving blobs.

4.2 The meeting-vertex function, $x \mapsto \text{meet}^V(x)$.

In order to compute the local version of $x \mapsto \text{meet}^V(x)$ for a vertex v, we therefore consider the radius-2 ball centered on v, shown in fig. 11. Because we used a triangle-mesh, the vertices on the perimeter of this ball form a cycle and the locally-induced blobs are 1D segments of consecutive filled vertices. Let $x \mapsto n(x)$ be their count; In fig. 11 (a) and (b), we have $n(x) = 2$. Each segment is delimited by two edges painted in black, there are four of them in fig. 11 (a1) and (b1). Those edges are part of frontier$^E(x)$, and are also apex-edges of v. Finally, $n(x)$ is equal to the number of apex-edges belonging to the frontier, divided by two.

$$n(x) = (/^+(\text{apex}(\text{frontier}^E(x))))/2. \tag{3}$$

If $n(x) = 2$, filling v merges (resp. emptying v divides into) the two induced blobs. If $n(x) = 3$ the same reasoning applies with three induced blobs. Finally, if $n(x) \leq 1$, no division nor merge happens, hence:

$$\text{meet}^V(x) = n(x) \geq 2 \tag{4}$$

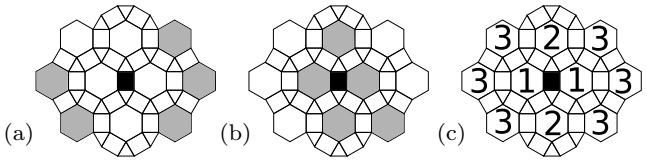


Fig. 12 Locally induced blobs (in gray) for (a) the merge-edge, (b) the dividing-edge of fig. 10. In (c) the radius 3 ball centered on an edge contains vertices at distances 1,2 and 3.

4.3 The meeting-edge function $x \mapsto \text{meet}^E(x)$.

For the local version of $x \mapsto \text{meet}^E(x)$, we consider the radius-3 ball centered on a given edge e, shown fig. 12. It includes three kinds of vertices: two immediate neighbors at distance 1, two apex neighbors at distance 2, and 6 more remote neighbors at distance 3. The immediate and apex neighbors form a rhombus. We will need a function $x \mapsto \forall^\diamond(x)$ which takes a boolV x and computes a boolE true for an edge if x is true within the rhombus centered on that edge. It can be computed as:

$$x \mapsto \forall^\diamond(x) = \forall^E(\forall^F(x)) \tag{5}$$

Note that because of overloading of $x \mapsto \forall^F(x)$, this formula also applies to a boolE to detect if all the four edges within the rhombus are true. The definition 1 implies that e is a local merge point, if and only if there are two locally induced blobs b_1 and b_2 verifying:

- (i) b_1 and b_2 occupy remote vertices outside each side of the rhombus.
- (ii) The rhombus itself is totally empty.

Both (i) and (ii) follow from point (ii) of definition 1 which stresses that filling just one immediate vertex neighbor of e should not merge b_1 and b_2. Point (i) is checked if and only if both immediate vertex neighbors of e belong to $\text{Frontier}^V(x)$. This is computed as $\forall^E(\text{Frontier}^V(x))$. Putting together (i) with (ii) we obtain:

$$\text{merge}^E(x) = \forall^E(\text{frontier}^V(x)) \wedge \forall^\diamond(\neg x) \tag{6}$$

A div-edge of blob is a merge-edge of holes. By replacing x by $\neg x$ we obtain:

$$\text{div}^E(x) = \forall^E(\text{frontier}^V(\neg x)) \wedge \forall^\diamond(x) \tag{7}$$

But $\text{frontier}^V(\neg x) = \text{frontier}^V(x)$, so we can factorize.

$$\text{meet}^E(x) = \forall^E(\text{frontier}^V(x)) \wedge (\forall^\diamond(x) \vee \forall^\diamond(\neg x)) \tag{8}$$

The expression $(\forall^\diamond(x) \vee \forall^\diamond(\neg x))$ means the rhombus' four vertices are either all full, or all empty. Equivalently, it can be computed as the absence of frontier edge within the rhombus: $\forall^\diamond(\neg\text{frontier}^E(x))$. We finally derive:

$$\text{meet}^E(x) = \forall^E(\text{frontier}^V(x)) \wedge \forall^\diamond(\neg\text{frontier}^E(x)) \tag{9}$$

5 A computing medium for the Voronoï Diagram (VD)

5.1 Triangular computing media

A computing medium is specified by a function updating a configuration:

Definition 2 Let $\tau_{i=1\ldots n}$ be some triangle-types and the product $\Gamma = \prod_{i=1}^n \tau_i$. A triangular computing medium is a function $f : \Gamma \mapsto \Gamma$.

A configuration of this medium is a vector of n fields of type τ_i, $i = 1 \ldots n$. The mapping f is the updating function. It is decomposed into n components: $(x_0, \ldots, x_n) \mapsto (f_0(x_0), \ldots, f_n(x_n))$, where each of the f_i uses the current configuration to compute one of the fields of the next configuration. Starting from an initial configuration $x^0 = (x_0^0 \ldots x_n^0)$, we iterate t times and obtain the configuration at time t: $x^t = f^t(x^0) = f(f^{t-1}(x^0)) = (x_0^t \ldots x_n^t)$. The sequence $x^0, x^1, \ldots, x^t$ represents the successive states of the medium. For a given i in $1 \ldots n$ a component $(x_i^t)_{t\in\mathbb{N}}$ represents the successive values of the field of type τ_i. It is stored on the medium's PEs and is therefore called a *mutable* field. The memory density of the medium is the number of bits that need to be stored, per vertex. It is the sum of the densities of τ_i, $i = 1 \ldots n$. In this introductory paper, we consider only media with a single mutable field, $n = 1$. Moreover, it will always be a boolV, thus the memory density is 1.

Exemple 1: The growing medium. This elementary medium is defined by $x \mapsto \text{neighborhood}^V(x) = \exists^V(\exists^E(x))$. From an initial set of seeds present in x^0, it let them grow, until they meet and merge, and fill the whole medium. We will transform it into a medium computing the VD, simply by stopping the growth just before the merges happen.

5.2 Exemple 2: Computing the discrete VD

Definition of discrete VD. Consider a boolV x encoding a set of vertices called seeds. The discrete Voronoï cell of one of the seeds, is the set of vertices strictly nearer to it than to other seeds. Note that this definition also holds for non-punctual seeds, i.e. x-blobs. The Voronoï Diagram (VD) is a representation of all the Voronoï

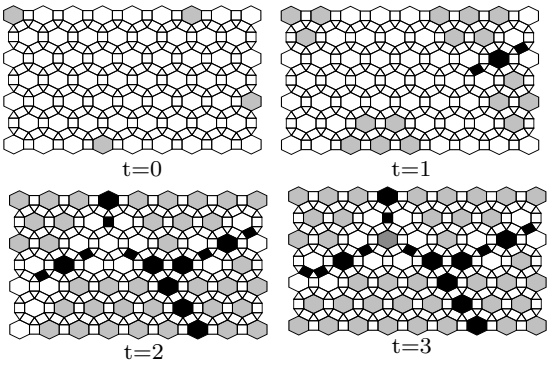


Fig. 13 Iteration of the hexagonal medium computing the strict Voronoï cells in a boolV field x^t (light gray). At $t = 0$, x^0 contains the seeds which grows to fill exactly the strict Voronoï Cell. Edges and vertices of merge(x^t) are in black. A multi-vertex is in dark gray.

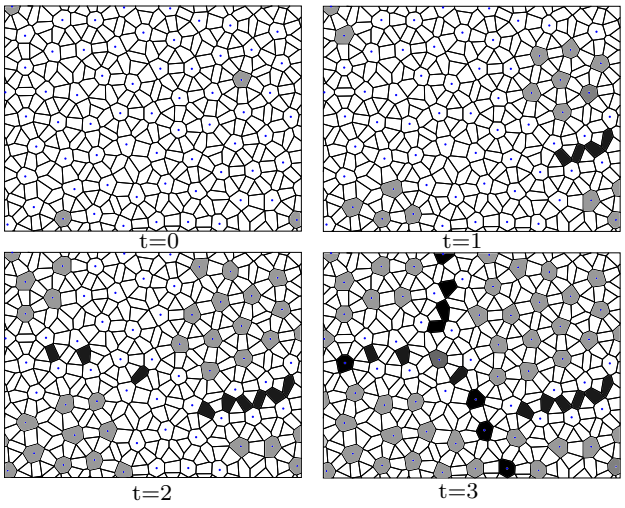


Fig. 14 Iteration of the amorphous medium of fig. 1 computing the strict Voronoï cells, graphical conventions are identical to those of preceding figure.

cells. In the continuous case, it can be drawn as a set of polygons. In the discrete case, the representation is less obvious due to the following discrete artifact: If two nearby seeds are at odd (resp. even) distance from one another, their Voronoï cells are separated by a vertex (resp. an edge). To take into account both cases, we represent the VD as a set of vertices (a boolV) together with a set of edges (a boolE). In the following definition, the distance considered is the VEF-distance which is the hop-count between vertices, edges, and faces:

Definition 3 Let x be a boolV. The discrete Voronoï Diagram of x-blobs is the set of edges and vertices equidistant to at least two nearest x-blobs.

The VD is thus a combination of a boolE and a boolV: VD(x)=(VD^V(x), VD^E(x)). Let closure^V : boolV$\times$boolE$\mapsto$boolV be defined as closure$^V(x, y) = x \vee \exists^V(y)$. We remark that closure^V(VD(x)) encodes VD(x) using only vertices. It also has the key topological property of separating the seeds. This makes it another legitimate representation of the VD, using only vertices, that we call the *vertex-VD*.

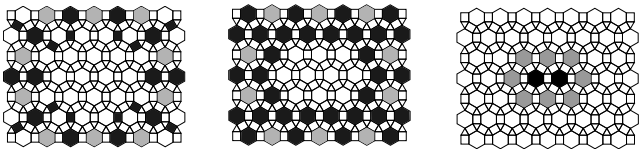
The VD medium. It uses a single mutable field x which is a boolV. Instead of marking the vertex-VD, x will mark the complement. It consists of Voronoï cells deprived of vertices adjacent to another distinct Voronoï cell. We call them "*strict Voronoï Cell*". At time $t = 0$, x^0 represents the seeds. The update re-use the growing medium's update, except it does not grow on closure^V(merge(x)) where merge(x) = (merge^V(x), merge^E(x)); In other words, the VD-medium is defined by $x \mapsto$ neighborhood(x) $\wedge \neg$closure^V(merge(x)). Fig. 13 (resp. fig. 14) illustrates three iterations of this update, for a crystalline (resp. amorphous) medium. Since merge-points were defined precisely to avoid merging blobs, the growth of blobs is stopped when the growing blobs come close to each other, one or two vertices away. As a result, each seed grows until it fills exactly its associated strict Voronoï cell. Convergence happens in time t_c smaller than half the diameter of the medium.

Theorem 1 *The medium $x \mapsto$neighborhood(x)$\wedge$ $\neg$closure^V(merge(x)) fills exactly the strict Voronoï cell of all the seeds present in the initial configuration.*

Proof. We first notice that the vertex-VD is preserved from one iteration to the next: vertex-VD(x^{t+1}) = vertex-VD(x^t). This is shown by decomposing vertex-VD(x^t) in two parts:

1. The part not adjacent to x^t is preserved because growth is uniform.
2. The part adjacent to x^t is precisely C_t = closure^V (merge(x^t)), which means it is already detected as part of the vertex-VD at time t and remains empty.

The configuration $(x^t)_{t\in N}$ is increasing and will therefore converge at a time t_c. C_t is included in the vertex-VD of x^t and also keeps growing. At time t_c, x^{t_c} is surrounded by C_{t_c}, otherwise it would keep growing at t_c+1. The connected components of vertices within the complement of C_{t_c} are of two types: either totally empty or totally marked. The totally marked components partition (x^{t_c}), and each one contains a seed. They are precisely the strict Voronoï Cell of x_{t_c}-blobs, because their vertex-VD contains C_{t_c}, and C_{t_c} surrounds them, and therefore separates them from the rest of the medium. Furthermore, since the vertex-VD is preserved, so are the strict Voronoï cells, which is the complement. So the strict Voronoï cells of x_{t_c} are also the strict Voronoï cells of x_0.



(a) (b) (c)

Fig. 15 Large multi-vertex component. (a) Some seed (gray), meeting-points (black). Strict Voronoï cells are limited to the seed themselves (b) closure^V(merge(x^{t_c})) (black) separates vertice in strict Voronoï cells, plus (c) a zone including two multi-vertice (black) equidistant to six seeds.

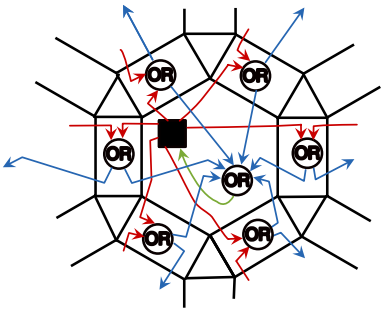


Fig. 16 A tile of the circuit compiled from the growing medium for the hexagonal lattice. The filled square is a flip-flop 1-bit register. The circles are logical ORs.

Robustness Our computation of the VD is robust: we can choose not to update a PE's flip-flop with a small uniform probability. This will generate random fluctuation in the uniform growing and will result in computing a non-deterministic VD, but correct on average. Failure of computing the next state can thus be considered a valid option, and this probably facilitates things when considering synchronization.

Multi-vertices. Let us study the unmarked components in the complementary of C_{t_c}. They are part of the vertex-VD as we defined it. They are surrounded by C_{t_c} which is also unmarked, therefore, they can be detected as outside$^V(x^{t_c})$. They contain vertices equidistant to at least three seeds, because vertices equidistant to only two seeds will be reached by the two waves sent by those two seeds, ane end up being marked in black. We call those "*multi-vertices*". In fig. 13 and fig. 14 , we choose a set of seeds to illustrate the apparition of such an unmarked component, and indeed, it contains a multi-vertex equidistant to three seeds. As fig. 15 shows, a multi-vertex component can be arbitrary big when the seeds are regularly spaced on a big discrete circle so that the circle's center is equidistant to all the seeds. In the discrete case, such situations can occur with a non-zero probability, and we do witness them every now and then. This demonstrates that the discrete VD is not necessarily "thin" as in the continuous case.

6 Compilation in synchronous sequential circuits

High-performance simulation of a triangular medium is obtained by first compiling it as a circuit. This compilation also enables a comparison with Cellular Automata (CA), the normal way of specifying crystalline computing media.

Cellular circuits. A synchronous sequential circuit is made of gates, wires, and memory elements. We use flip-flops for memory. The output of each flip-flop only changes when triggered by the clock pulse, so changes to the logic signals throughout the circuit all begin at the same time, at regular intervals, synchronized by the clock. A given triangular medium is compiled into a circuit, by putting together circuit parts associated with each of its operations. For example:

- a multi-cast is a fan-out wiring
- A boolean transfer communication is a wire crossing a simplicial tile.
- A boolean reduction is a tree of binary logic gates.
- A boolean mutable field is a flip-flop.

The compiled circuit is called a *cellular circuits*, the word "cellular" conveys the idea of translation-invariance implied by triangle-types. Fig. 16 represents a tile of the cellular circuit compiled from the growing medium.

Complexity of a cellular circuit. It is measured by three quantities:

1. The radius is the VEF-distance to the furthest simplicial tile taking part in the update of a given tile. It measures how far information must travel.
2. The memory density is the number of flip-flops needed, per vertex.
3. The gate density is the number of binary gates needed, per vertex.

The gate density measures the simulation time-cost. The 6-input logical OR of figure 16 needs a tree of 5 binary OR gates. The OR gate in the edge tile is binary, the density of edges is 3. The total density is 5+3=8 OR-gates. In general, for a s-locus l, the gate density $g(l)$ of a logical reduction towards l is:

$$g(l) = (\text{fanout}(l) - 1) * \text{density}(l) \tag{10}$$

Folded form of cellular circuits. A compact way of representing cellular circuits is shown in fig 17 (a), where it is applied to the growing circuit. A fan-out multi-cast is drawn as a single thick wire. When a transfer happens, it is illustrated using a single simplicial tile instead of

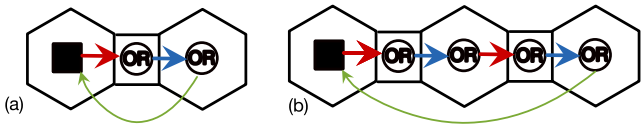


Fig. 17 Folded representation of the cellular circuit for (a) $x \mapsto$ neighborhood(x) (b) $x \mapsto$ neighborhood$^2(x)$. Thick arrows represent multicast wires to all neighbors, and thin arrows a single wire within the same tile.

all the neighbors. As the computation progress, new tiles need to be drawn, when the radius is increased. The update of a mutable field connects the most distant tile (with the highest radius), with the initial tile containing the corresponding flip-flop. The folded form allows us to directly read that the radius is 2. Folding enables to represent complex cellular circuits, such as the *accelerated growing-circuit* family parameterized by k: $C_k = x \mapsto \text{neighborhood}^k(x)$. It does k iterations of growing in one single update. It is shown in fig 17 (b), for $k = 2$. We will now use this family $(C_k)_{k>0}$ to illustrate a property of triangle-types.

Processing high radius, a comparison with CAs. The concept of radius also exists for CAs: it is the radius of the neighborhood considered for computing the next state. A neighborhood of radius r contains $O(r^2)$ vertices. With CAs, for executing one iteration, all these PEs are considered one by one. Therefore the cost of simulation grows quadratically with r. In contrast, with triangle-types, one iteration is decomposed into r μ-steps, each of radius 1. This achieves a linear complexity of only $O(r)$ instead of $O(r^2)$. The C_k family illustrates this fact: it has a radius $2k$ and a gate density of $8 * k$, also linear in k. Why is there such a miraculous gain of complexity? Each μ-step includes a simplicial reduction interleaving computation with communication; in this way, a computation done for one vertex immediately benefits neighbor vertices.

CAs usually use the minimum radius of 1, which means only the immediate neighborhood is considered. In contrast, with triangle-types, computing fields with a high radius is the normality, not the exception. A different portion of the landscape can be explored. The third simulation filmed in [9] uses a radius of 25.

The VD circuit. The folded form of the VD circuit is shown in fig 18. The radius is 4. The gate density is 55. It is obtained by summing over the reduction, using formula 10, and also deriving an implementation in binary logical gates, of "SUM/2" and "> 2". Note that this number 55 measures the complexity of the computation in a more precise way than just the number of states which is the traditional measure applied to CAs.

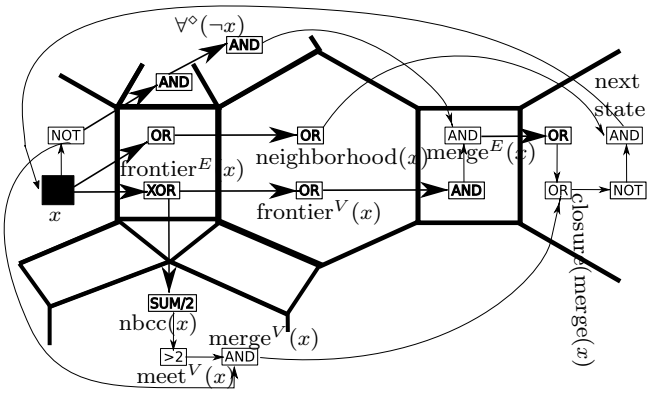


Fig. 18 Folded form of the VD circuit. thin gates are fed by thin arrows, they represent non-spatial computation within the same tile.

The folded form also illustrates how modularity can improve performance by factorizing computation. The auxiliary field $x \mapsto \text{Frontier}^E(x)$ is computed once, but reused twice for computing $x \mapsto \text{merge}^V(x) = \text{meet}^V(x) \wedge \neg x$ and $x \mapsto \text{merge}^E(x)$.

High-Performance circuit simulation. It can be obtained using a specific crystalline medium, namely the hexagonal lattice with 64 columns. This lattice is updated row by row, in a pipeline way. This enables two optimizations:

1. The SIMD capability of standard PC is exploited: 64 binary logic gates are evaluated in a single corresponding logic operation on long integers; similarly 64 bits are communicated with a single bit rotation.
2. Rows of generated intermediate fields are consumed at the same rate as they are created. Only r rows have to be stored, where r is the radius.

On a 2018 Asus laptop, with a 1,5 GHz i7 processor we measured over 64 gate-updates per CPU clock cycle, thus proving that the SIMD optimization can indeed create a speed-up of 64.

In-medium simulation. Recall that only vertices correspond to real PEs, the other locus: edges, faces, and t-locus, are handled as virtual PEs. Ultimately the simulation must be conducted in the computing medium itself. The gates assigned to a virtual PE must be reassigned to the nearest real PE, nearest in hop-count. It two real PEs happen to be at an identical distance, a fixed rule can be used, taking into account position in space. It must make sure that each real PE is assigned approximately the same number of virtual PEs.

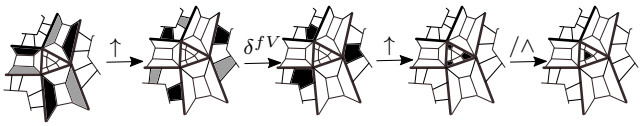


Fig. 19 The different steps for computing the vortex field from the sign field, following equation 11. Black (resp. gray) encodes true, (resp. false).

7 The vortex function

We provide another example of field functions to confirm the general applicability of triangle-types. It applies a transfer-reduction, which we did not use yet. It also illustrates a more elaborate information flow with two transfers.

We often have to compute distance fields towards moving sources, encoded on int3V. The method is detailed in [14]. When a bug occurs, it produces a triplet of values d_1, d_2, d_3 on the three vertices around a given face, verifying $d_1 < d_2$, $d_2 < d_3$, $d_3 < d_1$. This incoherence is possible because the distances are encoded using 3 bits, and comparison is done modulo 8. For example, the values $d_1 = 0$, $d_2 = 3$, $d_3 = 6$ verify the three inequalities. When such a bug happens, the face becomes the center of a vortex, around which distance values start to increase endlessly. We detect vortex in order to stop the execution immediately and identify the bug. The vortex field identifies faces, it is a boolF. It is computed from a boolvE representing the sign of the distance gradient on two bits. The pair of boolvE on each side of an edge can be (0,1), (1,0) or (0,0) depending wether the distance on one side is strictly greater, strictly smaller or equal to the other side. A transitive inequality $d_1 < d_2$, $d_2 < d_3$, $d_3 < d_1$ causes as an alternation of 0s and 1s on the three booleV-pairs around a vortex i.e, the value reads as 0,1,0,1,0,1, or 1,0,1,0,1,0. This simple pattern is detected by applying a first layer of XOR gates, and a second layer of binary AND gates which check that the XORs are all true. XORs are applied to two kinds of pairs:

1. The pair of two boolvE on each side on an edge.
2. The pair of two boolvE adjacent to a given vertex.

A closer look shows that the first kind of XORs is not required, For the second kind, we need to first do a transfer, and then apply a transfer-reduction δ^{fV}. The mapping from sign to vortex is computed as follows:

$$\text{vortex} = /^{\wedge} \circ \uparrow \circ \delta^{fV} \circ \uparrow (\text{sign}) \qquad (11)$$

The different steps of this composition are illustrated in fig. 19. The folded circuit is represented in fig. 20. The XOR density is 6, the AND density is 4, in total the gate density is 10.

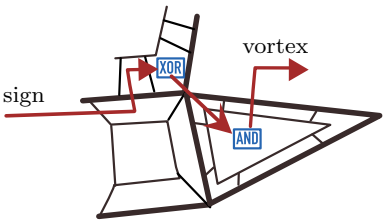


Fig. 20 Folded form of the vortex circuit. Transfer tiles need to be represented.

8 Conclusion

Programming a fine-grained 2D computing medium such as 2D Cellular Automata (CA) or more generally non-crystalline amorphous media is a difficult task. More specifically, simulating large objects which are extended in space also requires exploiting the 2D topology and this increases the difficulty. This paper presented a parallel data structure called "triangle-type" that simplifies the programming task, in this particular context. The triangle-type assumes that the Processing Elements (PEs) of the medium are connected as a triangle mesh to expose the 2D-topology. The primitive triangle-types are fields of values located on the vertices, edges, and faces of the triangle mesh, as well as three pairs of points located between vertices, edges, and faces. All those points are handled as virtual PEs. The primitive operators do simple processing of fields using one-to-one communications, multi-casts, and reductions.

Throughout this paper, we focused on demonstrating how triangle-types enable a simple *modular* way of programming. We started by pointing out small combinations of operators that realize meaningful macro-operators. We then assembled macro-operators to produce more complex computations: First, we computed the blob-predicate which specifies points in space where a simulated large object can extend without merging with other objects, or diminish without dividing itself into two objects. Second, we re-used the blob-predicate to program a computing medium that converges to the Voronoï Diagram (VD) of a set of initial seeds. The update function of this medium specifies uniformly growing the seeds until just before merging occurs. This technique was inspired by [2,18] who propagated waves on a 2D-grid CA and defined the VD when the waves collide. With triangle-types, we computed it on the hexagonal grid, and more generally, on any triangle mesh. Furthermore, the simple algorithmic essence of the wave technique is captured, and neighbors need not be considered one by one, as is typically done with CAs. This illustrates how a more abstract programming style is enabled. The VD is not a trivial example; many CA publications have been created on this subject. How-

ever, this exemple probably may not convey the correct idea about the general applicability of triangle-types. We provide another example of a function that uses a transfer-reduction and computes "vortex".

More generally, what is the scope of triangle-types? Let us first make explicit one limitation. Our ultimate target hardware is amorphous computing media. Their lack of structure is such that neighbors of a given PE cannot be distinguished. In this context, the only way of computing in a deterministic way is to use reductions on values provided by neighbors. Indeed, a reduction applies a commutative, associative operation. Thus, the order in which the neighbors are considered does not matter. For crystalline media, this achieves a property known as *rotation-invariance* [22]. Totalistic CA [24], which sums the immediate neighbor's integer state, is an example of rotation-invariant CA. With triangle-types, we use any type of reductions, not just the sum. More fundamentally, we define 12 different types of neighborhoods on which to reduce: six for simplicial reductions, and six for transfer reductions. This increases expressiveness to the point that being forced to compute using only reductions is not considered a constraint. Quite the contrary, it feels natural. Rotation-invariance is not a limit for our purpose. We implement *artificial physics* laws, so we always compute rotation-invariant quantities anyway. Embedding rotation-invariance in the operations themselves incorporates useful domain-specific information at the syntactic level which alleviates the programming task.

Does the scope of application of triangle-types include any rotation-invariant function? We are convinced of a more general property. Our experience is that as soon as processing can be specified in terms of the 2D geometry, then it can be efficiently programmed using triangle-types. This encompasses real or artificial physics, as well as digital image processing. We have been using triangle-types for over 12 years, and solving many problems such as distances to moving targets [14], convex hulls [17], Gabriel graphs [16], homogeneization in 1D [15] and 2D [9]. Above all, we have assembled a complex computing medium that can simulate Self Developing Network (SDN) of membranes. The goal of this SND-medium is to broaden the scope of what can be computed on a medium and ultimately reach general-purpose computing. An execution, interpreting a flow of host-instructions dictating the development of a virtual 2D-grid of membranes can be viewed in [9].

Expressiveness itself is not the only ingredient necessary for a good programming layer of computing media. The simulation means updating thousands of PEs, over hundreds of time steps, which can be very time-consuming. High performance is needed for interactive simulation, and this is the second crucial advantage of triangle-types. Therefore we also described how 2D-type can be compiled into circuits whose execution exploits the SIMD capabilities of a standard laptop. The VD-medium uses 1 bit of state, 55 gates per PE. It is simple enough to be run in an interactive mode, without optimization. In contrast, the SDN-medium uses 77 bits of state and 13878 gates, and it requires the SIMD optimization for preserving interactivity.

In summary, triangle-types enable the programming of complex computing medium, as well as their interactive execution. It thus opens the door to explore a new landscape of parallel models.

Acknowledgment

We thank Luidnel Maignan for his fruitful comments.

References

1. Abelson, H., D.Allen, Coore, D., Hanson, C., Homsy, G., T. F. Knight, J., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Commun. ACM **43**(5), 74–82 (2000)
2. Adamatzky, A.: Voronoi-like partition of lattice in cellular automata. Mathematical and Computer Modelling **23**(4), 51 – 66 (1996)
3. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic (TOCL) **20**(1), 1–55 (2019)
4. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, pp. 436–501. IGI Global (2013)
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. Computer **48**(9), 22–30 (2015)
6. Bhattacharjee, K., Naskar, N., Roy, S., Das, S.: A survey of cellular automata: Types, dynamics, non-uniformity and applications. Natural Computing **19**(2), 433–461 (2020)
7. Chopard, B., Droz, M.: Cellular Automata Modeling of Physical Systems (01 1998)
8. Coore, D.: Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer. Ph.D. thesis, MIT (1999)
9. Gruau, F.: Video illustrating the medium for self developing network (2018), https://youtu.be/8yVkfD0_G9s or https://www.lri.fr/ ∼gruau/#development
10. Gruau, F., Eisenbeis, C., Maignan, L.: The foundation of self-developing blob machines for spatial computing. physica D:Nonlinear Phenomena **237** (2008)
11. Gruau, F., Maignan, L.: Spatial types: a scheme for specifying complex cellular automata to explore artificial physics. In: TPNC 2018. LNCS, vol. v, p. pp (2018)
12. Gruau, F., Malbos, P.: The blob: A basic topological concept for hardware-free distributed computation. In: UMC 2002. LNCS, vol. 2509, pp. 151–163 (2002)

13. Kahn, J.M., Katz, R.H., Pister, K.S.: Next century challenges: mobile networking for "smart dust". In: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking. pp. 271–278 (1999)
14. Maignan, L., Gruau, F.: Integer gradient for cellular automata: Principle and examples. In: SASO 2008. IEEE (2008)
15. Maignan, L., Gruau, F.: A 1D cellular automaton that moves particles until regular spatial placement. Parallel Processing Letters **19**(2), 315–331 (2009), http://dx.doi.org/10.1142/S0129626409000249
16. Maignan, L., Gruau, F.: Gabriel graphs in arbitrary metric space and their cellular automaton for many grids. ACM Trans. Auton. Adapt. Syst. **6**, 12:1–12:14 (June 2011). https://doi.org/http://doi.acm.org/10.1145/1968513.1968515, http://doi.acm.org/10.1145/1968513.1968515
17. Maignan, L., Gruau, F.: Convex hulls on cellular automata. In: Proceedings of the 9th international conference on Cellular automata for research and industry. pp. 69–78. ACRI'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1927432.1927440
18. Maniatty, W.A., Szymanski, B.K.: Fine-grain discrete voronoi diagram algorithms in l1 and l norms. Mathematical and Computer Modelling **26**(4), 71–78 (1997)
19. Rauch, E.: Discrete, amorphous physical models. International Journal of Theoretical Physics **42**(2), 329–348 (feb 2003)
20. Schlömer, T., Heck, D., Deussen, O.: Farthest-point optimized point sets with maximized minimum distance. In: Proceedings of the ACM SIGGRAPH (2011)
21. Toffoli, T., Bach, T.: A common language for "programmable matter" (cellular automata and all that). Bull. It. Assoc. Artificial Intelligence **14**(2), 187–201 (2001), http://pm1.bu.edu/ tt/publ/aiia.ps.gz
22. Toffoli, T., Margolus, N.H.: Invertible cellular automata: a review. Physica D: Nonlinear Phenomena **45**(1-3), 229–253 (1990)
23. Van Kreveld, M., Schwarzkopf, O., de Berg, M., Overmars, M.: Computational geometry algorithms and applications. Springer (2000)
24. Wolfram, S.: Statistical mechanics of cellular automata. Reviews of Modern Physics **55**(3), 601–644 (jul 1983)
25. Zhou, H., Jin, M., Wu, H.: A distributed delaunay triangulation algorithm based on centroidal voronoi tessellation for wireless sensor networks. In: MobiHoc '13. ACM (2013)