

# Self-organizing Text in an Amorphous Environment

**Orrett Gayle**

University of the West Indies, Mona  
orrett.gayle@uwimona.edu.jm

**Daniel Coore**

University of the West Indies, Mona  
daniel.coore@uwimona.edu.jm

In an amorphous computing environment, myriad irregularly located computing elements asynchronously execute a common program and communicate locally to produce some pre-specified emergent behaviour. We have implemented a mechanism for robustly generating patterns of self-organising text in an amorphous computing environment. Our method uses the Growing Point Language (GPL), presented in [1], and builds on ideas put forward there for constructing text-like patterns. One shortcoming of that technique was that patterns for producing a text character were sensitive to signals that were localised to the origin of the text. As a result, the methods for generating these patterns did not scale well enough to be able to produce arbitrarily many of them. We have found a way to allow the conditions that govern the formation of a character to propagate arbitrarily far from the starting point, thereby allowing us, in principle, to produce arbitrarily long concatenations of text characters.

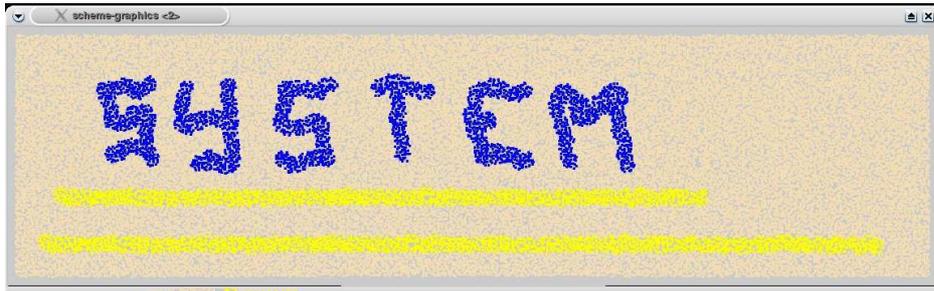
We used a pen metaphor, so that the description of our self-organising text is similar to text being drawn on paper, namely, by a series of interconnected strokes. Using this pen metaphor, characters may be combined via the GPL **network** abstraction in a perfectly natural way to produce words. We also implemented a general serializing mechanism which is used to ensure that the information used to produce one character does not interfere negatively with the production of the subsequent character. This mechanism permits long strings to be drawn reliably in a manner that scales well with the number of text characters to be drawn. Our simulations show that we can produce strings that are longer than we were previously capable of producing, and that short strings are produced more reliably.

## 1 Introduction

In an amorphous computing environment, myriads of simple computing elements interact locally, under the control of a common program to produce some pre-specified coherent behavior. The goal of amorphous computing is to find programming paradigms that allow us to engineer the emergence of coherent behaviors so that they can be used to produce self-organization on a massive scale, hitherto observed only in nature[1].

Several approaches to controlling the complexity of an Amorphous Computer have been developed [3, 7, 6] and includes recently published work [4, 2]. The Growing Point Language (GPL), developed by Coore [3], is especially suited for producing self-organizing patterns that are primarily topological rather than geometric.

As an application of GPL, Coore [3] showed how simple text characters could be made to self-organise. This was done merely as a demonstration of the possibilities with GPL. This implementation had a few problems, the most important of which was that the solution did not scale to accommodate arbitrarily large sequences of text characters.



**Figure 1:** A self-organizing “SYSTEM”. Each dot represents a processor. Its colour represents the state that it has assumed after running a common program. The initial conditions included assigning the state of the bottom line, and special status to three other processors located near the bottom left corner of the text.

We present here, an extension, which is scalable, of the method presented in [3], for generating text in an amorphous environment. Whereas the old method had difficulty producing strings longer than three characters, our enhanced version is capable of producing strings of six characters (see Figure 1) and probably more (simulation size limitations make it difficult to test on longer). One important product of this work is a serializing mechanism that provides a general purpose mechanism for generating repeated “spatial processes”.

## 2 Background

The Growing Point Language (GPL) is a programming language for specifying interconnect topologies in a coordinate-free way. The behaviour of a GPL program is determined by its instructions, the domain (collection of processing elements) on which it is executed and a set of initial conditions. The principal concept in GPL is the growing point, which describes a path in the GPL domain. At any given time, an instance of a growing point resides at a single location in the domain, called its *active site*. A growing point has a tropism, which is expressed as an affinity for either increasing, decreasing, or constant pheromone concentrations in the vicinity of the growing point. As a growing point's active site moves from one location to a neighbouring one (according to its tropism), we say that the growing point *propagates*. An active site may *secrete* a pheromone, which initiates a process of diffusion [8] that is centred at the active site. When the active site ceases execution at a location, and it does not propagate to a neighbouring location, then we say the growing point has *terminated* (at that location).

The *trajectory* of the growing point is the sequence of locations that its active site visits. The active site may also deposit some *material* at its current location. The material may be detected by other active sites, and may be used to influence their decisions. In this way, growing points can be defined to have influence over other growing points in important ways. For example, we can determine where the trajectories of two growing points intersect by having each one *sense* the material deposited by the other.

The definition of a growing point dictates the topology of its path, however the geometry of a particular path that a growing point produces, being dependent on the geometry of the domain, is determined only after the growing point has been invoked at some location.

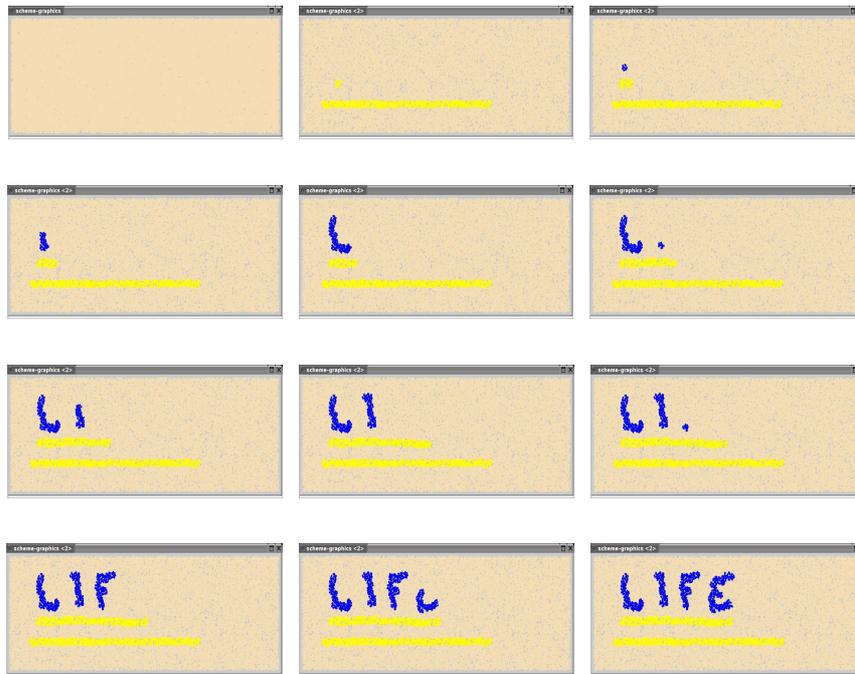
Logical groupings of growing points in a GPL program are called *networks*. A growing point can be viewed as a relation between the location of its invocation and the location(s) of its termination. The network abstraction is an extension of this idea: it defines a relation between two sets of locations called the *inputs* and the *outputs*. The inputs represent the locations from where growing points may be invoked and the outputs represent a subset of termination locations of those growing points through which the network may interface with other networks. Compound networks can be defined by allowing the outputs of one network to act as the inputs of another; such a connection is called a cascade of networks.

## 3 Implementation

The program directives for generating text are arranged in three layers of abstractions. At the lowest level are the most primitive operations required for drawing line segments and rays in various directions relative to a reference line. Note that there is no *a priori* knowledge of vertical and horizontal directions. Horizontal and vertical directions are relative to the reference which is estab-

lished from the initial conditions of the environment. We use a metaphor of a pen making strokes to define the growing points that produce line segments relative to the reference line. The second layer of abstraction is the collection of character-shape descriptions. These are implemented as GPL networks, which are composed from the pen stroke growing points of the lower layer. The third layer is the word pattern itself: it is implemented as a GPL network that is formed by cascading letter networks. When the programmer wishes to construct a self-organising word, she defines a cascade of letter networks to spell the word she wants, configures a domain to indicate where the bottom left corner of the network and the reference line should be, and executes the program on the domain.

### 3.1 Word Organization



**Figure 2:** The evolution of a self-organizing text pattern. Colours represent material that has been deposited at each point (default colour = no material, black = ink material)

As an example, let us consider how to define the word “LIFE” as a self-organising piece of text, using our GPL libraries. Figure 2 illustrates the execution of this program through a sequence of snapshots of the domain. The

colour of a processor is an indication of the material that has been deposited (by growing points) at that processor's location.

We first defined each character as a GPL network of two inputs and two outputs. String patterns are created by defining a character sequence as a cascade of the individual character networks. The network definition below shows how this example was constructed. Observe how intuitive the definition of the text appears at this highest level of abstraction.

```
(define-network (LIFE (txt-in line-in) (txt-out line-out))
  (==> (txt-in line-in)
    L txt-director I txt-director F txt-director E
    (txt-out line-out)))
```

The symbol `==>` is an abbreviation for cascade. In this network definition, a network called `LIFE` has two inputs (`txt-in` and `line-in`) and two outputs (`txt-out` and `line-out`). This tells us that in order to start the program, we will need to supply two points of the domain that will behave as these two input locations. The expression starting with `==>` says to use the two inputs to the `LIFE` network as inputs to the `L` network (which is a network that has two inputs and two outputs), then use its outputs as inputs to the `txt-director` network, whose outputs are then used as inputs to the `I` network and so on, until the outputs of the `E` network are supplied as the outputs of the overall `LIFE` network. The purpose of the `txt-director` network is to restore the conditions necessary to draw the next character; it will be explained in more detail shortly. Note that the cascade combinator would allow us to build even bigger networks by combining word networks in the same way that letters were combined to make words.

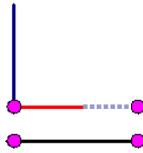
### 3.2 Character Networks

The second level of our abstraction hierarchy is the network definitions for the characters themselves. Here is the definition for the `L` network, and Figure 3 illustrates its execution.

```
(define-network (L (txt-in line-in) (txt-out line-out))
  (at txt-in
    (start-gp up-pen CHAR-HEIGHT)
    (--> (start-gp right-pen ( / CHAR-WIDTH 2) )
      (--> (start-gp lift-pen:right ( / CHAR-WIDTH 2))
        (->output txt-out))))
  (at line-in
    (--> (start-gp base-line CHAR-WIDTH)
      (->output line-out))))
```

Here, the `L` network is implemented directly in terms of growing points. The `at` command allows us to initiate growing points from a network's input. The `-->` symbol is an abbreviation for the `connect` command, which allows the termination point of the first growing point started in the expression to serve as the initial point of the growing point(s) that follows. The `->output` command causes a point to behave as the output of a network, which means that it will initiate any activities specified by a network for an input to which it is connected (via a network cascade).

The growing points `up-pen` and `right-pen` have been defined with tropisms that, when combined with the correct initial conditions, will produce trajectories that match their names. The expressions `CHAR-HEIGHT` and `CHAR-WIDTH` represent constants that determine the dimensions of a character in terms of neighbourhood hops. Based on this explanation, it is not hard to see that the character `L` is defined by drawing two growing points from the same originating point: one portion up for the height of the character, and the other to the right for about half of the width (see Figure 3). All that remains for us to explain is how the growing points such as `up-pen` and `right-pen` are defined.



**Figure 3:** Illustration of the `L` network

### 3.3 Determining which way is up

Two pheromones, *base-line-long* and *dir-pheromone*, are responsible for establishing the vertical and horizontal directions, respectively. We construct a reference line that secretes *base-line-long* with sufficient strength that at a distance of `CHAR-HEIGHT` hops away from the line, processors can still detect a non-zero concentration of *base-line-long*. The reference line grows along with the characters being drawn, as could be seen in the definition of the `L` network<sup>1</sup>. The growing points for drawing vertical lines have tropisms that are sensitive to *base-line-long*: away from the line is *up* and towards the line is *down*. For example, the essential parts of the `up-pen` definition are:

```
(define-growing-point(up-pen length)
  (material ink)
  (tropism (ortho- base-line-long))
  ... ; other characteristics omitted
  (actions
    ...
    (when (< length 1) (terminate))
    (default (propagate (- length 1))))))
```

The tropism of `up-pen` causes it to move away from the reference line. At each active site, the code in the `actions` clause is executed. The parameter `length` is decremented on each step of the active site until it finally gets to 0,

<sup>1</sup>Note that we actually draw the reference line ahead of the character that appears above it, so in fact, the part of the reference line that is drawn at any given moment is always to the right of the character that is drawn simultaneously.

when the growing point terminates. The overall effect is to draw a vertical line upwards, whose length is the number of hops specified as the `length` parameter at the time the growing point instance was started (with a `start-gp` command).

### 3.4 Going the right way

The initial deposit of *dir-pheromone* is produced from a point specified in the initial conditions, which was always to the left of the first character network. So, a constant concentration of *base-line-long* pheromone indicates a direction parallel to the reference line; the direction of increasing concentrations of *dir-pheromone* is left and the direction of decreasing concentrations is right.

The challenge of making a scalable process to produce arbitrarily long texts reduced to being able to replenish the source of *dir-pheromone* after each character was drawn. In this way, we get a kind of inductive property on our text strings: if the current character has sufficient information to determine left/right and up/down, then the subsequent character will also. Then, if we ensure that our initial conditions provide enough information for the first character to be formed properly, then the remainder of our text will be also.

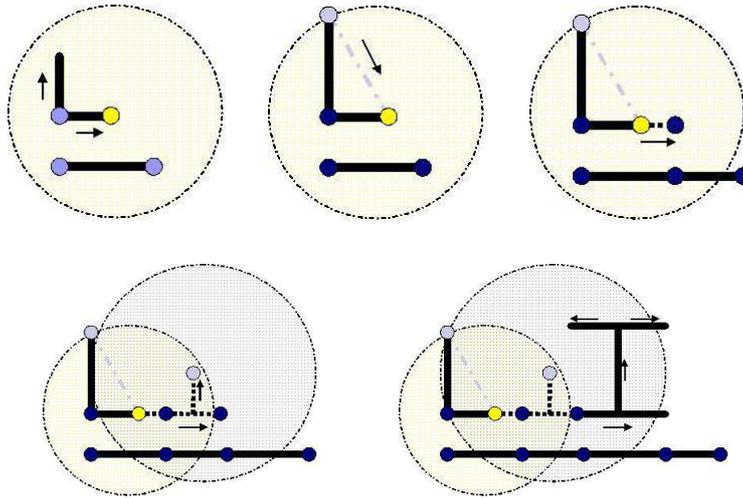
### 3.5 Propagating direction information

The earliest attempt at generating self-organising text with GPL did not even bother to attempt to establish this inductive property: it used a one-time initial secretion of *dir-pheromone* over a range far enough to cover the distance of all of the characters to be drawn. The initial attempts at achieving this inductive property, described in [3], embedded replenishing secretions of *dir-pheromone* within the definitions of the character networks, usually defined to occur from somewhere near the centre of the character. This had two problems: the definition of a character was now more complex, and the replenishing secretion interfered with the construction of the current character.

In our current approach, we defined a network, called `txt-director`, that can be cascaded with character networks, whose sole function is to secrete *dir-pheromone* far enough to provide guidance to one character. We then cascade an instance of the `txt-director` network in between successive character networks. This took care of the replenishment of *dir-pheromone*, but not of its interference with the current character.

Our solution to this second problem was to define a pair of growing points that implemented a synchronization mechanism. This was used to ensure that the whole character would be completely drawn before the subsequent network started. To accomplish this, the point to become the output of the character network secretes a homing pheromone. The final segment of the character's pattern is then sequenced (using `connect`) with a growing point that seeks out the homing pheromone. Upon finding the source of the homing pheromone, the growing point terminates and yields the output point of the network. Since this network is cascaded onto a `txt-director` network, the overall effect is that the

secretion of *dir-pheromone* for the subsequent character does not take place until the current character has been completely drawn.



**Figure 4:** The Serializing Mechanism. The top diagrams illustrate the formation of the ‘L’ character, and the serializing growing ensuring that the ‘L’ is complete before its output point is defined. The bottom diagrams illustrate the role of the `txt-director` network (shown with a dotted line) between consecutive characters. The oval shows the extent of secretion of *dir-pheromone*

## 4 Results and Discussion

We have presented a powerful means for controlling a complex system in the GPL language. We have also shown how it can be specifically used to generate complex patterns such as text characters. The methods we have used are an improvement on previous methods because we are able to generate longer strings, and short strings more reliably than we were previously able to do. Our implementation is also modular in that both the production of the characters as well as the maintenance of the necessary signals have been captured by the same abstraction mechanism. This gives us the power to combine them freely and robustly, in much the same way that a digital circuit designer may work with logic gates over transistors. We thus show that with the appropriate module definitions, it is still possible to apply traditional engineering techniques to controlling complex systems.

The major disadvantage of our approach is the fact that the growing points involved in the serializing mechanism are hard-coded into the the character networks, which means that changing the font, size or style, of our characters may require tweaking these growing point. It is important to note that the potential for interference between networks will always be present because there are a finite number of pheromones and an unbounded number of uses of them in generating a pattern that has unbounded extent. The means of resolving this interference does not have to be by synchronization, for example, we could arrange to have pheromones alternate roles in the same networks. This idea would cause a doubling of space required to store the program, though parameterised tropisms as implemented in [5] could be used to recover that space.

Qualitatively, we have shown that non-trivial patterns can be engineered to emerge in a complex system from relatively simple interactions between the system's elements. The techniques we have used can probably be generalised to solve other types of pattern formation problems – the hard part is finding the right building block to abstract as a GPL network. One of the shortcomings of our methods is that we rely on subjective evaluations for assessing the success of the formation of a pattern. We would, one day, like to have a tool that is capable of giving us more objective measurements.

## Bibliography

- [1] ABELSON, Harold, Don ALLEN, Daniel COORE, Chris HANSON, George HOMSY, Jr. THOMAS F. KNIGHT, Radhika NAGPAL, Erik RAUCH, Gerald Jay SUSSMAN, and Ron WEISS, “Amorphous computing”, *Commun. ACM* **43**, 5 (2000), 74–82.
- [2] BEAL, Jacob, “Programming an amorphous computational medium.”, *UPP*, (2004), 121–136.
- [3] COORE, Daniel, *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*, PhD thesis MIT (1999).
- [4] COORE, Daniel, “Abstractions for directing self-organising patterns.”, *UPP*, (2004), 110–120.
- [5] D’HONDT, Ellie, and Theo D’HONDT, “Amorphous geometry”, *Proceedings of the 2001 European Conference on Artificial Life (ECAL2001)*, (2001).
- [6] KONDACS, Attila, “Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation”, *International Joint Conference on Artificial Intelligence* (2003).
- [7] NAGPAL, Radhika, *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*, PhD thesis MIT (2001).

- [8] PEARSON, John E., “Complex patterns in a simple system”.