Self-Replication and Self-Assembly for Manufacturing

Robert Ewaschuk[*] and Peter D. Turney[†] (corresponding author)

Abstract

It has been argued that a central objective of nanotechnology is to make products inexpensively, and that self-replication is an effective approach to very low-cost manufacturing. The research presented here is intended to be a step towards this vision. We describe a computational simulation of nanoscale machines floating in a virtual liquid. The machines can bond together to form strands (chains) that self-replicate and self-assemble into user-specified meshes. There are four types of machines and the sequence of machine types in a strand determines the shape of the mesh they will build. A strand may be in an unfolded state, in which the bonds are straight, or in a folded state, in which the bond angles depend on the types of machines. By choosing the sequence of machine types in a strand, the user can specify a variety of polygonal shapes. A simulation typically begins with an initial unfolded seed strand in a soup of unbonded machines. The seed strand replicates by bonding with free machines in the soup. The child strands fold into the encoded polygonal shape, and then the polygons drift together and bond to form a mesh. We demonstrate that a variety of polygonal meshes can be manufactured in the simulation, by simply changing the sequence of machine types in the seed.

---

[*] Institute for Information Technology, National Research Council of Canada, Ottawa, Ontario, Canada, K1A 0R6, rob@infinitepigeons.org

[†] Institute for Information Technology, National Research Council of Canada, Ottawa, Ontario, Canada, K1A 0R6, peter.turney@nrc-cnrc.gc.ca, (613) 993-8564 (corresponding author)

1 Introduction

Researchers have argued that one of the main objectives of nanotechnology is to manufacture products inexpensively, and that this goal can be effectively achieved by self-replication [2], [8], [9]. We believe that it is useful to develop computational simulations of self-replicating nanotechnology, as engineering tools to assist in the design of actual self-replicating machines.

In our earlier work with JohnnyVon 1.0, we developed a computational simulation of machines that join to form self-replicating strands (chains of machines linked by flexible bonds) [17]. These machines drifted about in a virtual liquid, simulated as a two-dimensional continuous space with Brownian motion and viscosity. There were two types of machines, which enabled a strand to encode an arbitrary bit string, by designating one type of machine as representing 0 and the other as 1. Although strand replication faithfully preserved the encoded bit strings, the information in the strings played no functional role in JohnnyVon 1.0. In effect, the simulation had genotypes (genetic code) without phenotypes (bodies). From the perspective of potential applications in manufacturing, the absence of phenotypes was a major limitation of JohnnyVon 1.0.

This paper introduces JohnnyVon 2.0, which builds on its predecessor by adding phenotypes to the simulation. The design of JohnnyVon 2.0 was partly inspired by the work of Seeman on building nanometer-scale structures with DNA [14], [15]. In living organisms, replication is based on DNA (the genotype) and the information encoded in DNA is used to build proteins (the major structural material of the phenotype). Seeman has shown that DNA can serve both as a device for self-replication (genotype) and (surprisingly) as a building material for nanoscopic structures and tools (phenotype). By choosing the appropriate sequence of codons, DNA can be programmed to self-assemble into a wide variety of structures, such as cubes, octahedra, one-dimensional strands, two-dimensional meshes, and three-dimensional arrays. Seeman discusses a variety of potential nanotechnological applications for these structures. For example, a three-dimensional DNA array could facilitate x-ray crystallography, by serving as a scaffolding for holding molecular samples in a regular lattice [15].

In JohnnyVon 2.0, there are four types of machines, drifting in a two-dimensional continuous space with Brownian motion and viscosity (i.e., a simulated liquid). The

machines exert spring-like attractive and repulsive forces on each other, but internally they are finite state machines. The input to each state machine is based on the presence or absence of bonds with neighbouring machines and on the internal states of bonded neighbours. The internal states govern when bonds are formed or broken and the angles at which bonded machines are joined, and thus determine whether a strand will form a straight line or fold into a specific polygonal shape.

Following the hint of Seeman's work, a strand in JohnnyVon 2.0 serves as both a genotype and a phenotype, at different stages in its career [14], [15]. Like living organisms (but unlike von Neumann's strategy [21]), JohnnyVon 2.0 takes a template-based approach to self-replication. A strand begins its career as a genotype. While acting as a genotype, the strand is approximately straight, so that it can provide a good template for replication. Brownian motion and interactions with other machines will cause the strand to bend slightly, because the bonds between the machines are flexible, but the system is designed so that forces will tend to straighten the strand. Later in its career, the strand may become a phenotype. When this happens, the bonding forces change, causing the bonding angles to alter, and the strand folds. This folding is (approximately) analogous to the way that proteins fold. A folded strand acts as a structural element and is no longer capable of replication.

A typical run of a JohnnyVon 2.0 simulation begins with a soup of unbonded machines and an initial unfolded seed strand of bonded machines. Free (unbonded) machines connect to the seed strand, eventually forming a double strand (two parallel strands). When the new strand is complete, the two strands break apart, and thus we have self-replication. A strand will continue to self-replicate until unbonded machines become scarce. When a strand has not encountered an unbonded machine for a relatively long period of time, the strand stops replicating and folds. The shape that it folds into depends on the types of machines in the strand and their sequential ordering. Folded strands drift in the virtual liquid and bond with each other, forming a mesh. The user can specify the shape of the holes in the final mesh by selecting the sequence of machine types that compose the initial seed strand.

In Section 2, we discuss related work with von Neumann's universal constructor, self-replicating loops, and artificial chemistry. The details of JohnnyVon 2.0's design are explained in Section 3, including the changes that have been made from JohnnyVon 1.0 [17]. We present our experiments in Section 4. Each experiment is a run of the

simulation with an initial seed strand. We demonstrate that a variety of polygonal meshes can be manufactured by varying the initial seed. Section 5 examines limitations of JohnnyVon 2.0 and discusses problems and projects for future work. Potential applications are suggested in Section 6 and we conclude in Section 7.

2 Related Work

Sipper provides a good survey of research on self-replication [16]. Some of the research involves actual mechanical devices and some is based on organic chemistry, but we restrict our discussion here to computer simulations of self-replication. We briefly review von Neumann's universal constructor [21], self-replicating loops [6], [10], [11], [12], [13], [18], [19], and artificial chemistry [4], [5].

2.1   Universal Constructor

Von Neumann's approach to self-replication was to design a universal constructor, which could build anything, and therefore could build itself as a special case [21]. He described five different models (i.e., five different kinds of simulations), with varying levels of realism and concreteness. The design of the universal constructor was only worked out in detail for the cellular automata model, which was the most abstract of the five models.

In von Neumann's cellular automata model, the universal constructor was composed of a group of several thousand cells that begin in a specific configuration of initial states. Another line of cells acts as a kind of tape, which is read by the universal constructor. For any given finite configuration of cell states, there is a tape that can cause the universal constructor to build the given configuration. As a special case, there is a tape that can cause the universal constructor to build a copy of itself, thereby self-replicating.

2.2   Self-Replicating Loops

Langton demonstrated self-replication in a cellular automata model that was much simpler than von Neumann's model [6]. He achieved this simplification by designing a constructor that could construct only itself, instead of trying to make a universal constructor. His cellular automata model had eight states instead of twenty-nine and his constructor was composed of a group of about a hundred cells in a specific initial configuration, instead of a group of several thousand cells.

In Langton's model, the cells of the constructor are arranged in a loop. The states of the cells in the loop go through a cycle, periodically creating a copy of the original loop. Starting from the initial loop, increasing numbers of copies spread across the grid.

The idea of self-replicating loops in cellular automata models has since been developed further by many researchers [11], [12], [13], [19]. Self-replicating loops have exhibited many interesting behaviours, including evolution [11], [13] and self-repair [19]. Tempesti has described a self-replicating cellular automata model that can perform computations and build structures [18]. Morita and Imai have shown how a self-replicating cellular automata model can encode a variety of shapes of loops, beyond the usual square loops [10].

2.3 Artificial Chemistry

Hutton introduced self-replication in an artificial chemistry simulation, using a template-based approach [4]. A chain of molecules forms a template against which other molecules bond, similar in concept to JohnnyVon 1.0 [17]. A run of the simulation begins with a seed chain in a soup of free molecules. By a series of chemical reactions, a parallel chain of molecules forms next to the seed chain. When the parallel chain is complete, it separates from the seed chain and the process repeats.

Hutton's first approach was a cellular automata model [4], but the discrete space constrained the mobility of the simulated molecules, hence Hutton's second approach used a continuous space [5], like JohnnyVon 1.0 [17]. In Hutton's second model, molecules move in a continuous two-dimensional space, following linear trajectories until an obstacle (e.g., the container wall or another molecule) is encountered (i.e., the motion is a billiard ball model). When molecules make contact with each other, they undergo a chemical reaction that bonds them together, according to the rules of the artificial chemistry.

In Hutton's first model [4], the molecules only replicate, but in his second model [5], they also build a circular barrier, suggestive of a cell wall. Each time a chain replicates, the new chain builds a wall around itself.

The basic objects in Hutton's system ("molecules") are simpler than the basic objects in JohnnyVon ("machines"), which makes Hutton's system more computationally efficient. However, JohnnyVon has a richer virtual physics, including Brownian motion, attractive and repulsive fields, momentum, and viscosity. Hutton's molecules are points, with no

directional orientation, whereas JohnnyVon's machines are shapes with angular orientation, which can rotate, experience twisting forces, and have angular momentum. The richer virtual physics in JohnnyVon may be useful for simulations of manufacturing.

3 JohnnyVon 2.0

We first give an overview of JohnnyVon 2.0 and then describe the system in detail. We encourage the reader to begin by viewing Figure 5 in Section 4.1. This figure should make it easier to understand the following discussion.

3.1 Overview

The basic objects in the JohnnyVon simulation are called *machines*. There are four types of machines, numbered 1 through 4. All four types are shaped like a plus sign '+' and appear visually identical in the simulation (see Figure 5). Each machine has five arms, but two of the arms overlap, so the figures seem to show four arms. In the figures, the arms are represented by black lines.

The machines are mobile and can rotate at any angle, but it is convenient to describe them when they are rotated into a standard reference position, which we call the *canonical position*. In the canonical position, the shortest arm points down, the two longest arms point right and left, and the medium-length arm points up. Another short arm points up in canonical position, but it is hidden by the medium-length arm that points up. (The three machines labeled B, D, and F in Figure 1 are in canonical position. The fourth machine is upside-down.)

The two longest arms, pointing left and right when the machine is in canonical position, are called the *left* and *right* arms. When machines bond to form a strand, adjacent machines in the strand are bonded to each other at the tips of their left and right arms.

The *up* arm is the longer of the two arms that point up when the machine is in canonical position. When a strand replicates by forming a mirror strand, the machines in the mirror strand are bonded to their neighbours in the original strand at the tips of their up arms. Also, when the strands fold into polygons and join to form a mesh, the polygons bond to each other at the tips of their up arms.

The shorter of the two arms that point up, when the machine is in canonical position, is the *repellor* arm. This arm overlaps the up arm in the figures, so it is not visible. When a strand has completely replicated, repulsive fields are briefly activated at the tips of the

repellor arms. This splits the original strand from the mirror strand and pushes the two strands apart.

The short arm that points down in canonical position is the *overlap detector* arm. This arm is used to detect when two folded strands (e.g., polygons) overlap in a mesh.

Machines move about in a two-dimensional continuous space, bounded by a grey box. The centers of the machines are confined to the interior of the grey box. This space is called the *container*. A virtual liquid fills the container. The trajectory of a machine is determined by Brownian motion (random drift due to the liquid) and by interaction with other machines and the walls of the container. The liquid has a viscosity that dampens the momentum of the machines. When there are machines suspended in the liquid, we call it a *soup*.

The arms of the machines have attractive or repulsive *fields*. The range of a field is bounded by a circle. In addition to attracting or repelling, a field can also exert a bending force, which twists the machines to form a particular angle. A field's interaction (attract, repel, twist, or ignore) with another field is determined by many factors, including the type and state of each machine. The fields behave somewhat like springs. The center of every field is always at the tip (the outer end) of an arm.

The point at which the five arms meet is called the *middle* of the machine. This is not the machine's geometrical center, but it is treated as the center of mass in the simulation. For example, a rotational force will cause a machine to rotate about its middle.

Although the space is two-dimensional, machines can slide over one another, as if they were floating in a thin film of liquid. Machines interact with each other through their fields, rather than by direct contact. They do not experience direct collisions, but their fields can collide.

Machines can *bond* together when the field of one machine intersects the field of another. Not all fields can bond. This is described in detail later. The machine that is bonded to the up (left, right) arm of a given machine is called the up (left, right) *neighbour*. A chain of machines joined by left arm to right arm bonds forms a *strand*.

During replication, the bond angles are such that the replicating strand tends to be straight. Brownian motion and other forces perturb the strand, so it cannot be perfectly straight, but twisting forces in the bonding fields tend to straighten the strand, so it is

rarely far from being straight. We call these approximately straight strands *unfolded* strands or *genes* (a strand in its genotype state).

Under specific circumstances (described in Section 3.4.1), each bond in a strand will change its angle, causing the strand to fold. Such strands are called *folded* strands or *phenes* (a strand in its phenotype state). A phene will fold to form a closed loop. A group of phenes can bond together, forming a *mesh*.

A machine with no bonds is called a *free* machine. A typical simulation begins with a soup of free machines and a single seed gene. The *seed gene* is an unfolded strand that initiates the process of self-replication. The first child of the seed gene forms the *seed phene*, which acts as a starting point for the growth of the mesh. This ensures that one seed phene will yield only one mesh.

A left-neighbour-to-right-neighbour bond is a *sideways* bond. Machines in a strand (both phenes and genes) are joined by sideways bonds. An up-neighbour-to-up-neighbour bond is an *up* bond. Phenes in a mesh are joined by up bonds. During replication, a parent gene is joined to its partially constructed child gene by up bonds.

The JohnnyVon simulation proceeds in a sequence of discrete time steps. The initial configuration is called *step* 0 or *time* 0. The *state* of a machine is the combination of internal information (counters, bonds, and other state variables) and external relationships (position, rotation, and velocity) that determines the behaviour of a machine. A *counter* is a special piece of information stored in each machine that normally increments during each time step. Each machine has several counters. States are described in more detail in Section 3.3.

Each bond has a desired angle (which changes when a strand folds). The two machines that participate in each bond have a *tolerance* for the difference between the current angle and the desired angle. Forces can push bonds out of tolerance. Bonds that are consistently out of tolerance can break.

One of our main design objectives was to make JohnnyVon 2.0 programmable by the user, by specifying the initial configuration of the simulation, without making any changes to the rules of the virtual physics or the behaviours of the machines. By selecting the types of machines in the seed gene and specifying their sequential order, the user can program JohnnyVon to make a variety of polygonal meshes. In a manufacturing application, we envision the user selecting the desired machines from four bins, joining

them together to make a seed gene, and then dropping the seed into a vat of free machines. The user would be able to manufacture a variety of products without making any modifications to the individual machines.

With only one type of machine, the user would be able to program the simulation only by specifying the length of the seed gene. With two types of machines (as in JohnnyVon 1.0 [17]), programs can use more efficient binary coding. In principle, two types of machines would be sufficient for programming mesh construction, but we found that four types provide a good balance of simplicity and programmability. Four types are enough to encode a variety of shapes (see Table 6 in Section 3.4.2 and the experiments in Section 4), yet four types are not so many that the system is unwieldy. The four types of machines are somewhat analogous to the four amino acids in DNA (adenine, thymine, guanine, and cytosine), which encode programs for building proteins.

3.2   Basic Modifications

This subsection describes the core changes in the simulation that were made from JohnnyVon 1.0 to JohnnyVon 2.0. (Sections 3.3 and 3.4 discuss new features, as opposed to modified features.)

3.2.1   Variable Field Sizes

Bonds between machines are formed by spring-like attractive fields. Part of the mechanism that was in place to support mutation in JohnnyVon 1.0 was a variable field size. In certain circumstances, the field would be small, permitting rare accidental bonds, while other times it would be large, to strengthen intentional bonds. The accidental bonds were a cause of mutation (replication errors), whereas the intentional bonds were part of faithful replication.

In JohnnyVon 2.0, field sizes do not change. Fields attract, repel, or ignore other fields, but they have a constant circle of influence. In situations where the original version has small fields, the new version has inactive fields. This modification substantially reduces the likelihood of mutations.[1]

---

[1] Mutations are desirable when modeling biological life, but may be undesirable when modeling manufacturing processes.

3.2.2 Physical Constants

The physical constants for viscosity, Brownian motion, and motion dampening were changed to suit the new requirements. The values of these constants were experimentally tuned to achieve our design objectives while maximizing the speed (computation efficiency) of the simulation.

3.2.3 Arms and the Machine

In JohnnyVon 1.0, each machine was shaped like a capital letter 'T'. Each machine had four arms, but two of the arms overlapped (along the vertical bar of the T), so the figures in the paper seem to show three arms [17]. Each arm had an attractive or repulsive field with a circular shape, centered on the tips of the arms. The fields were colour coded, and we named the arms according to the colours of their associated fields.

In JohnnyVon 2.0, each machine is shaped like a plus sign '+' (see Figure 5 in Section 4.1). Each machine now has five arms, but two of the arms overlap, so the figures seem to show four arms. It is no longer convenient to refer to the arms by the colours of their fields. We now refer to the arms by their relative positions (up, left, right), when the machine is rotated into a canonical position. The lengths of the arms have been modified to facilitate building meshes.

3.3 Machine States

The state of a machine is represented by a vector. The vector elements that represent internal aspects of the machine are all discrete. The vector elements that represent external relationships between machines are mostly continuous. The discrete, internal elements are governed by state transition rules that are applied in discrete timesteps. The continuous, external elements are governed by the laws of the virtual physics. The physical laws are inherently continuous, but they are necessarily approximated discretely in any computational simulation.

Internal state information includes various flags, counters, and state variables, as summarized in Table 1. External state information includes spatial location and orientation, angular velocity, linear velocity, the presence or absence of bonds with other machines, and bonding angles, as given in Table 2. Some derived variables are shown in Table 3. The derived variables are calculated from state variables.

Insert Table 1 here.

Insert Table 2 here.

Insert Table 3 here.

Machines that are directly bonded together can sense each other's states. This is analogous to cells in cellular automata, which can sense the states of their immediate neighbours in the grid. The state transition rules and the virtual physics are local, in the sense that there is no global control structure. No machine can directly sense the state of another machine unless they are directly connected, although state information can be passed neighbour-to-neighbour along a strand. No machine can directly exert a force on another machine unless the circular boundaries of their fields overlap, although forces can be passed neighbour-to-neighbour along a strand.

Most of the state transition rules and physical laws in JohnnyVon 2.0 are carried over from JohnnyVon 1.0 without change. The details of JohnnyVon 1.0 are fully described elsewhere [17]. The changes we made in JohnnyVon 2.0 were outlined above, in Section 3.2.

3.4   New Rules

The following subsections describe the rules that are new in JohnnyVon 2.0.

3.4.1   Folding

The leftmost machine in an unfolded strand determines when the strand will fold. A machine knows it is leftmost when it has a *right-neighbour* but no *left-neighbour*. As part of its internal state, each machine maintains a *fold-counter*. After a strand has replicated and split, the *fold-counter* in each machine in the newly formed strand starts counting. When a machine gains an *up-neighbour*, it triggers a *reset-counter* signal. This signal is passed to the *left-neighbour,* down the strand, until it reaches the leftmost machine. When the leftmost machine receives the signal, it resets its *fold-counter* to 0. This is illustrated in Figure 1. In this way, as long as a replicating strand continues to receive new *up-neighbours*, it will not fold up. If the strand successfully replicates, the leftmost *fold-counter* in each of the two strands (the child strand and the parent strand) is set to 0.

Insert Figure 1 here.

Once the *fold-counter* in the leftmost machine hits a fixed upper limit, that machine triggers a *fold-now* signal. This signal is passed down the strand, setting the *folded* flag as it goes. This causes the strand to fold up according to the types of each of the two machines involved in a sideways bond.

When a strand folds, there are typically some *up-neighbours* attached to the folding strand, as the folding strand has usually partially replicated itself. Machines with an *up-neighbour,* but having a false (0) *replicated* flag, monitor their *up-neighbour*'s *folded* flag. If this flag becomes true (1), then such a machine will set its own *shatter* flag to true, and thus release all its bonds. (Shattering is described in detail in 3.4.7.)

Machines with the *seed-gene* flag set to 1 never fold, thus the initial seed strand is always available, to continue replicating as soon as there is a supply of free machines of the right types. This is a safeguard against situations in which a temporary scarcity of free machines persists for longer than the fixed upper limit on *fold-counter*.

3.4.2   Angles and Bonds

In contrast to JohnnyVon 1.0, we now have four types of machines instead of two types, and the type of a machine affects its behaviour in both genes and phenes. In genes, the types govern bonding during replication, where the rule is simply, "Likes attract, others are ignored." More formally, the bonding rule for up bonds in genes is given in Table 4.

Insert Table 4 here.

This rule implies that each replicated strand is a mirror of its parent, rather than an exact copy. For example, a strand of types 1-2-3, reading left to right, in canonical position, will replicate as 3-2-1. The phenes that we demonstrate here (in Section 4) have this simple symmetry (i.e., the strands and their mirror images both fold into the same polygonal shapes), so this is not a problem.

The rules for up bonds in genes are different from the rules for up bonds in phenes. In phenes, only certain combinations of types will bond on their respective up arms, as given in Table 5.

Insert Table 5 here.

Any type of machine can form a sideways bond with any other type, in both genes and phenes, but a left arm must bond with a right arm (i.e., no left-left nor right-right bonding is allowed). In phenes, the types on each side of a sideways bond govern the angle the bond will take when the strands have folded, as specified in Table 6. This means that the types involved in each bond control the shape that the folded strand will take.

Insert Table 6 here.

Table 6 shows the sideways bond angle formed by each pair of machine types, when a strand is in its phenotype state; that is, when *folded* is set to 1 (true). When a strand is in its genotype state (*folded* is 0 for all machines in the strand), the sideways bond angles are all 0° (straight). Up bond angles are always 0° (ignoring random perturbations, from Brownian motion, for example).

The blank cells in Table 6 represent combinations of types for which we have not yet found a use. From Table 5, it can be seen that a type-3 machine cannot form an up bond with another type-3 machine when they are in phenes. Therefore we could control the shape of a 3-3-…-3 phene by specifying any desired value for the angle of 3-3 bonds in Table 6, but the resulting phenes would not be able to form a mesh. Similarly, we could control the shape of a 3-4-3-4-…-3-4 phene by giving any desired value for the angles of 3-4 and 4-3 bonds in Table 6, but the resulting phenes can form up bonds in multiple ways (3-4, 4-3, and 4-4; see Table 5). Thus we have limited control over the shape of the mesh that the phenes will form.

In Table 6, it can be seen that all sideways bonds involving type-1 machines are straight. This allows us to expand the size of a phene, without changing its shape, by inserting a sequence of type-1 machines along each edge (see Figure 2). However, it does not allow polygons with exactly two machines on each side, since expansion requires at least one type-1 machine inserted between two other machines.

Insert Figure 2 here.

Given the angles that are available to us in Table 6, some polygons (e.g., octagons and squares) require two types of machines, while others (e.g., triangles and hexagons) involve only one type. We chose to restrict the JohnnyVon 2.0 to four types of machines, in order to demonstrate that a small number of components can be combined to build a variety of structures (like Lego blocks). The angles that we chose make it easy to build triangular meshes. It may seem inconvenient to require two types of machines to build octagonal meshes, but in fact, having two types of machines is helpful with octagons. An octagonal mesh has both octagonal holes and square holes (see Image 4 in Figure 6 in Section 4.2). With two types of machines, we can prevent octagonal phenes from filling in the square holes in the mesh.

A single type of machine would be sufficient to create squares, but using two types permits rectangles that will mesh correctly. With two types of machines, the long and short sides of the rectangle can be distinguished by type, so that two sides will bond together only if they have the same machine type, and thus the same length (see Figure 3).

Insert Figure 3 here.

Though they could use two types, hexagons will form a mesh faster with only one type. Furthermore, in order to get the desired behaviour with four machine types, exactly one of the squares or hexagons had to use only one machine type, and the other had to use two. While it would be possible to have an irregular hexagon, this seems much less natural than a rectangle.[2]

Table 5 shows how machine types control phene bonding in meshes. The bonding rules in Table 5 were designed so that octagonal meshes will form correctly. They prevent octagonal phenes from filling square holes in the mesh.

Additional rules were created to support expansion of the size of a phene, without changing its shape, by inserting a sequence of type-1 machines along each edge. These rules are based on the derived variable *bend-location*. In a phene, a given machine can have either a straight (type 1) or bending (types 2, 3 or 4) machine bonded to its left or right arm. By looking at its neighbours, it can determine where it is in the context of the

---

[2] In a regular polygon, all sides have the same length. Rectangles are irregular.

phene, and hence it can calculate the value of *bend-location*. Certain values of *bend-location* can override the rules in Table 5 and disallow a bond that would otherwise be permitted. Table 7 explains the meaning of the different values of *bend-location*. Table 8 shows how *bend-location* affects bonding.

Insert Table 7 here.

Insert Table 8 here.

When the up field of a machine in one phene overlaps with the up field of a machine in another phene, the rules in both Table 5 and Table 8 must be satisfied before the up fields can bond. If there are no type 1 machines in the phene (i.e., all sideways bonds are bent; the shape is not expanded), then *bend-location* must have the value 3 for all machines, and thus (by Table 8) up bonds depend only on the machine types (Table 5).

In Table 7, we are assuming that the phene forms a closed loop. The error correction system will destroy open loops (Section 3.4.4). When a machine has no left or right neighbour (because it is at the end of an open strand), we treat the missing neighbour as if it were a bending type.

3.4.3   Overlap Detection

Because of flexibility in the mesh, and in individual phenes, two phenes can sometimes join a mesh in such a way that their desired positions overlap. This can be most easily seen by imagining a mesh of hexagons that is complete except for a single gap. Suppose that two hexagons jostle (by Brownian motion) into the gap, with slightly different alignments. At roughly the same time, one forms a bond with the phene above the gap, and one with the phene below the gap. As they straighten towards their ideal position (due to twisting forces on their up bonds), each phene may pick up new bonds around the edge of the former gap (see Figure 4).

Insert Figure 4 here.

If ignored, this problem spreads, since there are now unbonded up arms on each of the two overlapping hexagons, which permit new hexagons to join the mesh, overlapping

those around the former gap, and each of these hexagons in turn can bring in more overlapping hexagons.

To address this problem, an arm was added to the machines (new since JohnnyVon 1.0), called the *overlap detector arm*. In a folded strand, it points towards the center, and will only bond with other overlap detector arms. Both machines must be oriented in the same direction (up to a fixed degree of tolerance), and both machines must have their *in-mesh* flag set to true.

When an overlap detector bond is formed between two machines (necessarily in two different phenes), one machine (chosen arbitrarily) sets its *unfold* signal to true. This signal propagates to its left and right neighbours, setting *folded* to false as it goes. It also breaks the overlap bond that triggered it. The resulting unfolded strand behaves exactly like a newly replicated strand. It tries to replicate until *fold-counter* exceeds its limit, and then it folds up again.

In summary, when two phenes compete for the same gap in a mesh, one of them is forced to become a gene. Converting one of the phenes to a gene, instead of leaving it as a detached phene, allows time for it to drift away from the problematic area, or for the remaining phene to fill up the open bonds in the mesh.

3.4.4   Stress Detection

Another type of error can occur in a mesh, again due to the flexibility of the mesh. In this case, we can imagine five triangles bonding to form a pie shape, missing only one more triangle to form a hexagon. However, instead of a new triangle coming in to fill the gap, the two triangles on either side of the gap jostle together, forming a stressed pentagon rather than a hexagon. This pentagonal mesh may be part of a larger mesh, and thus some of the stress may be distributed through the larger mesh. This problem can be partially addressed by increasing the strength of some of the fields and decreasing the tolerance of some of the bonding angles, but it becomes increasingly hard to prevent as the mesh grows.

To detect this kind of problem, each machine maintains a *stress-counter*, which increments each time interval when the machine is not *in-tolerance*, and is reset whenever the machine is *in-tolerance*. This counter can be used to detect cases in which the mesh is stressed because phenes have bonded incorrectly. When the counter exceeds a fixed maximum, it causes the stressed phene to unfold, by setting its *unfold*

16

signal to true. The signal propagates to left and right neighbours, setting *folded* to false and dropping up bonds as it spreads through the neighbours.

The *in-tolerance* variable is purely local. Its state depends only on the relation between a machine and its immediate neighbours. Likewise, the *unfold* variable is purely local, but the information that it conveys can spread through each machine in a phene. When one machine's *unfold* variable becomes true, its immediate neighbours sense this, and then set their *unfold* variables to true.

3.4.5   Seeding the Mesh

In the initial seed gene, *seed-gene* is set to true, but it will be set to false for all of the child genes. If a machine has a true *seed-gene*, it will never trigger the *fold-now* signal. Since the initial seed gene will never fold, there will always be a strand that can continue replicating whenever free machines become available.[3]

The first child of the seed gene, and only the first child of the seed gene, becomes the seed phene. When the *seed-phene* flag is set to true in a strand, it does *not* mean that the given strand is the seed phene; it means that the next child of the given strand will become the seed phene. In the initial seed gene, *seed-phene* is set to true. When the seed gene first replicates, its child examines its parent's *seed-phene* flag and observes that it is set to true. The child then sets it *in-mesh* flag and its *folded* flag to true and it immediately folds to become the seed phene. The parent (the initial seed gene) then sets its *seed-phene* flag to false, so that its future children cannot become seed phenes. (When we say that a strand sets a flag to a value, it is a shorthand way of saying that every machine in the strand sets the flag to the value. Strands do not have flags of their own, other than the flags of their component machines.)

Every other gene, created after the first child, will begin its career with its *in-mesh* flag set to false. If two phenes meet with their *in-mesh* flags set to false, they cannot bond

---

[3] When the seed gene replicates, all of the information that it encodes, by the sequential order of the machine types in the strand, is also replicated. However, the child strand will have its *seed-gene* flags set to false, so it might be argued that the seed gene has not fully self-replicated, in a very strict sense. The purpose of the *seed-gene* flag is merely to ensure that replication will continue after a hiatus in the supply of free machines. This is not essential to the simulation. We could remove this flag without causing any major problems. Therefore full self-replication, in a very strict sense, is readily attainable, if desired, by a minor modification to JohnnyVon.

together. A phene can only bond to another phene if the other phene has its *in-mesh* flag set to true. When a machine (in a phene) with a false *in-mesh* flag meets a machine (in a second phene) with a true *in-mesh* flag, they bond (assuming they meet all the conditions in Section 3.4.2), and a signal propagates through the first phene, setting all of the *in-mesh* flags to true (but the signal only propagates from one machine to its sideways neighbour when their bond is *in-tolerance;* see Section 3.4.4). This ensures that the mesh can only grow from the seed phene.

3.4.6   Tolerances

Each machine will only form up bonds if all existing bond angles are within a certain tolerance. That is, if a machine's sideways bonds are at angles significantly different from the desired angles (i.e., the angles given by the rules in Section 3.4.2), then no up bonds will form during the current timestep. This prevents unintended up bonds during vulnerable times, such as during splitting or folding.

3.4.7   Shattering

There are a number of ways that a gene or phene can break. For example, during splitting, the phase of self-replication when two genes are pushed apart by their repellor arms, if one of them hits the wall of the container at an angle, it puts significant strain on the whole strand. As another example, an error in a mesh can eventually lead to enough strain to pull a phene apart (see Section 3.4.4).

If a machine loses a bond unexpectedly (which is any time other than when splitting or unfolding), or if it notices that its neighbour has folded, then the *shatter* flag is set to true. When a machine observes that its neighbour's *shatter* flag is true, the machine may respond by setting its own *shatter* flag to true. We say that the first machine is the *source* of a *shatter* signal that was *received* by the second machine.

The *shatter* signal always propagates through sideways bonds, setting the *shatter* flag to true in *left-neighbours* and *right-neighbours*. The shatter signal may also propagate to an *up-neighbour*, but only if the source machine has *replicated* but not *folded*. If the neighbouring machine has *replicated*, something went wrong with the split (two *replicated* machines should not be bonded before they're both folded); on the other hand, if the neighbour's *replicated* flag is false, then it may be part of an incomplete copy, and thus should be abandoned.

When a machine's *shatter* flag is true, it drops all of its bonds (the discrete timesteps ensure that the state is propagated, even if the bonds are broken during that time step, since machines consult their neighbours' states as they were at the beginning of the step). When the bonds have been dropped, it then sets *folded, seed-gene,* and *replicated* flags to false and becomes a free machine.

The *shatter* mechanism is not a subtle way to handle errors, but we have found it to be effective. In our simulations, shattering is relatively rare. This error correction mechanism is similar to Sayama's method for handling errors in self-replicating loops [12], [13].

3.5   Implementation

JohnnyVon 2.0 builds directly on the original JohnnyVon 1.0. Both systems are written in Java and their source code is available under the GNU General Public License (GPL) at http://purl.org/net/johnnyvon/.

4 Experiments and Discussion

In our first experiment, we demonstrate the construction of a small mesh of triangles, highlighting several important points in the replication and assembly. In the next set of experiments, we demonstrate replication and assembly of meshes built from each of the supported polygons, with one machine per side. We then show a mesh of polygons with more than one machine per side, a 3×1 rectangle and a triangle with three machines per side. Finally, to demonstrate scalability, we show a large mesh of triangles.

In the following figures, the inner grey square represents the container. The middle of a machine must stay inside the grey square. (It takes less computation to check whether the middles are within bounds than to check all of the arms.)

4.1   Self-Replication and Self-Assembly

In Figure 5, the images show a typical run of JohnnyVon 2.0. The run starts with a soup of 54 free type-2 machines and a seed gene of the form 2-2-2, and it ends with a triangular mesh.

Image 1: This shows the initial configuration. Each of the free machines is in a random position and the seed gene is in the center (it is the strand of three machines, forming a straight line).

Image 2: After 2,385 steps, the first replication is complete. We see two genes, immediately after they have split and their repellor arms have pushed them apart.

Image 3: The first child of the seed gene is folding up, to become the seed phene. The seed gene has already begun a second copy.

Image 4: By time 44,235, nearly all of the free machines are now attached to genes. Since there are so few free machines left, most of these genes cannot complete self-replication. As some of the incomplete strands' *fold-counters* hit their upper limit, they will fold and release free machines, allowing other genes to complete self-replication.

Image 5: We can see the second phene forming. In this image, it has not completely folded; the triangle has a small gap at the top.

Image 6: Slightly more than 3,000 timesteps later, the new phene has bonded with the seed phene.

Image 7: Now many more triangles have folded and joined the mesh. Two triangles have not yet joined (one is in the lower left corner and the other is near the center).

Image 8: The mesh is almost complete. In the bottom on the right, there is a pentagonal arrangement of five triangles. This would eventually be corrected (by one of the triangles releasing and unfolding; see Section 3.4.4), although the container is just barely large enough to hold a mesh that includes all of the machines, and thus errors may continue to form even as they are corrected. In a situation where the container constrains the mesh, it is possible for a machine to get attached to a mesh in such a way that it can never reach an equilibrium where all of its bonds are in tolerance, since the conditions for accepting new bonds are much looser than the conditions for detecting stress.

Insert Figure 5 here.

In this simulation, the container is relatively small, and therefore Brownian motion is relatively strong. With strong Brownian motion, free machines are quickly distributed throughout the container, thus a replicating strand has a steady supply of free machines. The small container also means that the phenes never have far to go to join the mesh, and will quickly be bumped into the right position. In a larger container, replicating strands will consume the machines in their local area, and then replication slows until diffusion replenishes the supply. It also takes longer for phenes to find a place where

they can join the mesh. We could speed up the action in a larger container by increasing the Brownian motion (i.e., turning up the heat), but that could damage the mesh.

4.2   Simple Polygonal Meshes

In Figure 6, we show assembled mesh structures. Four different regular polygonal meshes are shown, all with sides that are one machine in length. Each of these four simulations was started with a single seed strand and was executed until the mesh was well developed. The scale of the images in Figure 6 is different from the scale of the images in Figure 5. These simulations use a container about nine times larger in area than the simulations in Figure 5. Table 9 summarizes the four simulations in Figure 6.

Insert Figure 6 here.

Insert Table 9 here.

4.3   Fancy Meshes

In Figure 7, Image 1 (timestep 691,900) shows a mesh built of rectangles, rather than regular polygons. Because squares and rectangles use two types of machines (see Section 3.4.2), the rectangles only join the mesh if they are correctly oriented. The seed for Image 1 was 2-4-2-1-2-4-2-1.

Image 2 (timestep 78,800) shows large triangles. The seed was 2-1-2-2-1-2-2-1-2. The bonds between type-2 machines fold to form the corners, while the type-1 machines provide the extension to make these triangles larger. In principle, each phene can be made arbitrarily large using this approach.

Insert Figure 7 here.

4.4   Large Mesh

The image in Figure 8 demonstrates that meshes can grow correctly beyond a small number of triangles. The seed was 2-2-2. The mesh contains 234 triangles.

Insert Figure 8 here.

4.5   Discussion

The above experiments demonstrate that four types of machines are sufficient to enable the user to program a variety of meshes. By specifying the seed gene, the user can create triangular, square, hexagonal, or octagonal meshes (Section 4.2). The experiments in Section 4.3 show that the sizes of the holes in the mesh are infinitely adjustable (in principle; in practice, there will be computational limits). For rectangular meshes, an infinite number of shapes are possible, by adjusting the ratios of the lengths of the sides (Section 4.3).

To program JohnnyVon to build a mesh, the most important information is in Table 6. Looking at this table, the user can easily predict the shape that will be produced by any given seed. For example, consider the seed 2-4-2-1-2-4-2-1. Take eight sticks of equal length and put them on a flat surface. Reading the seed from left to right, let the first stick be "2". Place the second stick end-to-end with the first stick, and let it be "4". Arrange the two sticks so that their angle is 90°, as gi ven in Table 6. Continue in this manner until all eight sticks are arranged. The resulting shape will be a rectangle (see Figure 3).

When programming a mesh, note that some seeds will not form a closed shape, which will trigger the stress detection (Section 3.4.4). This may be viewed as a bug in the user's program or as a limitation of JohnnyVon 2.0.

5 Limitations and Future Work

JohnnyVon 2.0 has several minor limitations. For example, phenes must be closed for the system to work correctly. Although a hexagon composed of five machines and a gap in the sixth side can form a mesh, the error correction system would destroy the resulting mesh. Because closure of the phenes increases their rigidity, a mesh built of open phenes would be more flexible, and there may be other interesting effects.

The variety of phenes in JohnnyVon 2.0 is also somewhat limited. Our original goal, to support all regular polygons that tile the plane (triangles, squares and hexagons), is satisfied. JohnnyVon 2.0 also supports partial tiling with octagons (square gaps are left in the mesh), and full tiling with rectangles. However, we would now like to support concave shapes (e.g., stars), as well as more general polygons. It would be interesting to enable Penrose tilings and Kepler tilings [3].

The replication phase takes much longer with two (or more) types of machines than it does with one, since each free machine has fewer places to bond correctly (equivalently, each machine in the strand has fewer free machines available with which it can bond). Supplying two (or more) times as many machines increases the computation per timestep (roughly quadratically). However, JohnnyVon 2.0 should be parallelizable. This is another area for future work.

Like JohnnyVon 1.0, version 2.0 still runs on a standard desktop computer, thanks in part to improvements in hardware since the development of version 1.0. However, there were many experiments we wanted to try (e.g., polygons with 4 or 5 machines per side) that were not practical, given our available hardware and our patience. This problem can be addressed by improving the efficiency of our implementation, converting the code to a more efficient language than Java (which is likely to make it much less portable), parallelizing the code, or obtaining better hardware.

The computational complexity of the simulation increases with the size of the phenes, since each phene must be jostled to a place near where it belongs, and larger phenes move more slowly. Meshes of large phenes require many timesteps to be constructed. The problem may be alleviated by increasing the Brownian motion or decreasing the viscosity of the simulated liquid, but each of these solutions presents new problems. We have tuned the physical parameters, in an effort to balance these conflicting concerns. The current settings of the physical parameters appear to strike a good balance, but there is likely room for further improvements.

JohnnyVon would benefit from increased realism and increased programmability. Although JohnnyVon 2.0 provides a moderate level of programmability, it is not as programmable as we would like. One problem is that the mesh grows without control. Sometimes the mesh is relatively dense (as in Image 2 of Figure 6) while at other times it has many gaps (as in Figure 8). We would like to add a programmable mechanism for controlling the final size and shape of the mesh, and for avoiding meshes with large gaps (or for deliberately creating gaps, which may be useful for some applications).

The growth of a mesh from a seed phene may be viewed as a type of dendrite growth, such as is observed in Diffusion-Limited Aggregation (DLA) models (a class of fractal growth models) [22]. The literature on fractal growth models may suggest some useful methods for preventing dendrite growth and encouraging the formation of a filled mesh [20]. It may be possible to modify JohnnyVon so that the user can specify the desired

fractal dimension of the mesh. A filled mesh (in the current two-dimensional simulation) would have a fractal dimension of 2, and a dendritic mesh would have a fractal dimension somewhere between 1 and 2.

In the context of JohnnyVon's virtual physics, it may be meaningful to define a universal constructor (see Section 2.1). For example, we might say that a universal constructor would be capable of building any two-dimensional structure that can be constructed from a finite number of machines, such that up arms are bonded to up arms and left arms are bonded to right arms. The design for the structure should be encoded in a seed gene. Ideally, the seed would contain many fewer machines than the final structure, although this may not be possible when the final structure lacks a regular pattern. Much further work is required to make a universal constructor in the JohnnyVon model.

JohnnyVon 2.0 also provides a moderate level of realism, but again it is not as realistic as we would like. Our attractive and repulsive forces are somewhat unlike electrical or magnetic attraction and repulsion. The JohnnyVon simulation also does not attempt to model conservation of energy. Arbesman has recently done some interesting work on computational simulation of artificial life with conservation of energy [1].

Other steps towards increased realism would be to extend the simulation to three dimensions and to model the physics of the internal operations of the machines. Currently the external relations between machines are governed by a simple virtual physics, but the internal operations are described by abstract finite automata. However, both of these steps to realism would involve a significant increase in computational complexity.

6 Applications

In our previous work, we suggested that JohnnyVon 1.0 provided a plausible mechanism for nanoscale manufacturing [17]. A vat of liquid containing free machines would be seeded with a single strand, soon resulting in a vat full of copies of the seed strand. JohnnyVon 2.0 takes this application one step further, beyond self-replication to programmable construction of meshes. Since the user has some control over the size and shape of the holes in the mesh, we can imagine these meshes being produced for filtration, insulation, or simply as a kind of cloth.

If we can create a mechanism for controlling the size and shape of the mesh, more applications become possible. Since the system is accurate and self-correcting, pieces of cloth could be created exactly to specification, down to the size of a single machine.

The design of the individual machines in JohnnyVon is too complex to be implemented at the nanoscale, given the current state of the art of nanotechnology. However, the machines appear to be less complex than typical virsuses, which range in size from about 20 to 250 nanometers. Nanoscale is generally defined as about 1 to 100 nanometers. As mentioned earlier, JohnnyVon 2.0 was partly inspired by Seeman's work with DNA [14], [15]. A single DNA molecule is about 2 nanometers wide. JohnnyVon's design is somewhat different from Seeman's work, so it would not be accurate to describe his work as an implementation of JohnnyVon, but it does seem reasonable to say that JohnnyVon could be implemented at the nanoscale, given improvements in the state of the art.

On the other hand, it would not be difficult to implement JohnnyVon at the human scale, with current technology, using small robots (such as the Khepera robot) to implement the four types of machines.[4] This might be useful for research purposes, to test designs at a level between software simulation and nanoscale hardware.

7   Conclusion

JohnnyVon 1.0 demonstrated self-replication in a continuous two-dimensional space with virtual physics. JohnnyVon 2.0 goes beyond its predecessor by introducing a user-programmable phenotype, consisting of a variety of meshes. JohnnyVon 2.0 is more realistic than cellular automata models [6], [10], [11], [12], [13], [18], [19], more programmable than artificial chemistry models [4], [5], and more computationally tractable than von Neumann's universal constructor [21]. However, there is still much room for improvement in the degree of physical realism of the simulation and in the degree of programmability of the phenotype.

Like its predecessor, JohnnyVon 2.0 is a local model. There is no global data structure that represents strands or meshes; these are emergent entities that arise from the interactions of the basic elements (the machines). Each machine is autonomous and can only sense its immediate neighbours. Control is local, distributed, and parallel.

---

[4] http://www.k-team.com/robots/khepera/khepera.html

From four different types of machines, JohnnyVon can produce four different polygonal meshes, with an infinite number of possible sizes (as per Section 4.3). The user can specify the mesh that will be produced by encoding the desired size and shape in the initial seed, without making any changes to the physics of the simulation. Errors in replication and in mesh formation are automatically detected and corrected, using purely local mechanisms.

JohnnyVon 2.0 also avoids the "grey goo" scenario of self-replicating nanobots run amok. Replication and assembly are inherently limited by the supply of machines; when the free machines have all bonded, the process stops.

Acknowledgements

References

[1] Arbesman, S. (2004). *Erg: A Computational Energetics as an Approach to the Study of the Origins of Life.* Senior Thesis, Computer Science Department, Brandeis University.

[2] Drexler, K.E. (1992). *Nanosystems: Molecular Machinery, Manufacturing, and Computation.* New York: Wiley.

[3] Grunbaum, B., and Shephard, G.C. (1986). *Tilings and Patterns.* W.H. Freeman.

[4] Hutton, T.J. (2002). Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8, 341-356.

[5] Hutton, T.J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In J. Pollack *et al.*, eds., *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, 444-449.

[6] Langton, C.G. (1984). Self-reproduction in cellular automata. *Physica D*, 10, 134-144.

[7] Maynard Smith, J., and Szathmary, E. (1995). *The Major Transitions in Evolution.* Oxford University Press.

[8] Merkle, R.C. (1992). Self replicating systems and molecular manufacturing. *Journal of the British Interplanetary Society*, 45, 407-413.

[9] Merkle, R.C. (1994). Self replicating systems and low cost manufacturing. In *The Ultimate Limits of Fabrication and Measurement*, M.E. Welland, J.K. Gimzewski, eds., Dordrecht: Kluwer, pp. 25-32.

[10] Morita, K., and Imai, K. (1997). A simple self-reproducing cellular automaton with shape-encoding mechanism. In C.G. Langton and K. Shimohara, eds., *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems,* 489-496. Cambridge, MA: MIT Press.

[11] Reggia, J.A., Lohn, J.D., and Chou, H.-H. (1998). Self-replicating structures: Evolution, emergence and computation. *Artificial Life*, 4, 283-302.

[12] Sayama, H. (1998). Introduction of structural dissolution into Langton's self-reproducing loop. In C. Adami, R.K. Belew, H. Kitano, and C.E. Taylor, eds., *Artificial Life VI: Proceedings of the Sixth International Conference on Artificial Life*, 114-122. Los Angeles, CA: MIT Press.

[13] Sayama, H. (1999). A new structurally dissolvable self-reproducing loop evolving in a simple cellular automata space. *Artificial Life*, 5, 343-365.

[14] Seeman, N.C. (2003). DNA in a material world. *Nature*, 421 (January 23), 427-431.

[15] Seeman, N.C. (2004). Nanotechnology and the double helix. *Scientific American*, 290 (6) (June), 65-75.

[16] Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life,* 4 (3), 237-257.

[17] Smith, A., Turney, P., and Ewaschuk, R. (2003). Self-replicating machines in continuous space with virtual physics. *Artificial Life*, 9, 21-40.

[18] Tempesti, G. (1995). A new self-reproducing cellular automaton capable of construction and computation. In F. Moran, A. Moreno, J.J. Morelo, and P. Chacon, eds., *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life (ECAL95),* 555-563. Berlin: Springer-Verlag.

[19] Tempesti, G., Mange, D., and Stauffer, A. (1998). Self-replicating and self-repairing multicellular automata. *Artificial Life*, 4, 259-282.

[20] Vicsek, T. (1992). *Fractal Growth Phenomena*. Second Edition. Singapore: World Scientific.

[21] von Neumann, J. (1966). *Theory of Self-Reproducing Automata.* Edited and completed by A.W. Burks. Urbana, IL: University of Illinois Press.

[22] Witten, T.A., and Sander, L.M. (1981). Diffusion-limited aggregation, a kinetic critical phenomenon, *Physical Review Letters*, 47, (Issue 19-9), 1400-1403.

Table 1. Variables for elements of the state vector that represent internal aspects of the machine.

Variable name	Range	Description
type	{1, 2, 3, 4}	• type of machine • static for a given machine
id	{0, 1, 2, …}	• unique identifier for machine • static for a given machine
fold-counter	{0, 1, 2, …}	• used to decide when the strand should fold
repel-counter	{0, 1, 2, …}	• during splitting, controls how long the repellor arms of a strand are active
stress-counter	{0, 1, 2, …}	• counts the time since a machine was last *in-tolerance*
strand-position	{1, 2, 3}	• used to decide where a machine is in a replicating strand • described in detail elsewhere [17]
split-state	{1, 2, 3, 4}	• used to determine when to split • described in detail elsewhere [17], except that *split-state* now has a fourth value, indicating that *shatter* should be set to 1 (true)
reset-counter	{0, 1}	• indicates that the fold-counter should be reset
fold-now	{0, 1}	• indicates that each machine in the strand should set its *folded* flag
unfold	{0, 1}	• indicates that a phene should unfold • this occurs when one phene overlaps another in a mesh
seed-gene	{0, 1}	• flag for identifying the seed gene • the seed gene never folds, in case more free machines become available for replication
seed-phene	{0, 1}	• flag for making the seed phene
in-mesh	{0, 1}	• indicates whether this machine is connected to the mesh
replicated	{0, 1}	• 1 (true) if and only if the machine has been through a successful replication. • in particular, machines in the seed gene have not replicated at the start of a simulation
shatter	{0, 1}	• indicates that the machine should break all bonds and return to being a free machine.
folded	{0, 1}	• if 1, the machine tries to form angular bonds with its left and right neighbours • if 0, the machine tries to form straight bonds with its left and right neighbours

Table 2. Variables for elements of the state vector that represent external relations.

Variable name	Range	Description
x-position	$\Re$	• state of the machine with respect to the container
y-position	$\Re$	• vary due to Brownian motion, viscosity, and forces from
angle	$[0, 2\pi]$	interactions between fields
x-velocity	$\Re$	
y-velocity	$\Re$	
angular-velocity	$[0, 2\pi]$	
left-neighbour	$\{0, 1, 2, …\}$	• identifier of the machine (if any) bonded to the named
right-neighbour	$\{0, 1, 2, …\}$	arm
up-neighbour	$\{0, 1, 2, …\}$	

Table 3. Derived variables that are calculated from state variables.

Variable name	Range	Description
in-tolerance	{0, 1}	• indicates whether each existing bond is within a certain (fixed) tolerance of the desired angle • used to help avoid making bonds when the machine is in a potentially unstable situation.
bend-location	{1, 2, 3, 4}	• indicates where a machine is located in a phene

Table 4. Pairs of machine types that will permit an up bond when in genes.

Bond	1	2	3	4
1	+			
2		+		
3			+	
4				+

Table 5. Pairs of machine types that will permit an up bond when in phenes.

Bond	1	2	3	4
1				
2		+		
3				+
4			+	+

Table 6. Folding angles for sideways bonds between two machine types when in phenes.

Angle	1	2	3	4
1	0°	0°	0°	0°
2	0°	120°	45°	90°
3	0°	45°		
4	0°	90°		60°

Table 7. The meaning of the different values of *bend-location*.

Value	Meaning of value
1	*right of bend:* right neighbour is straight (1) and left neighbour is bending (2, 3, 4)
2	*left of bend:* left neighbour is straight (1) and right neighbour is bending (2, 3, 4)
3	*in bend:* both left and right neighbours are bending types (2, 3, 4)
4	*extender:* both left and right neighbours are straight types (1)

Table 8. Pairs of *bend-location* values that will permit an up bond when in phenes.

bend-location	1	2	3	4
1		+		
2	+			
3			+	
4				

Table 9. Some basic observations about each image in Figure 6.

Image	Phenes	Seed gene	Timestep	Initial free machines	Phenes in mesh	Genes remaining
1	Triangles	2-2-2	246,000	201	50	5
2	Squares	4-2-4-2	498,600	200	24	1
3	Hexagons	4-4-4-4-4-4	448,600	160	20	1
4	Octagons	2-3-2-3-2-3-2-3	1,107,400	120	9	2
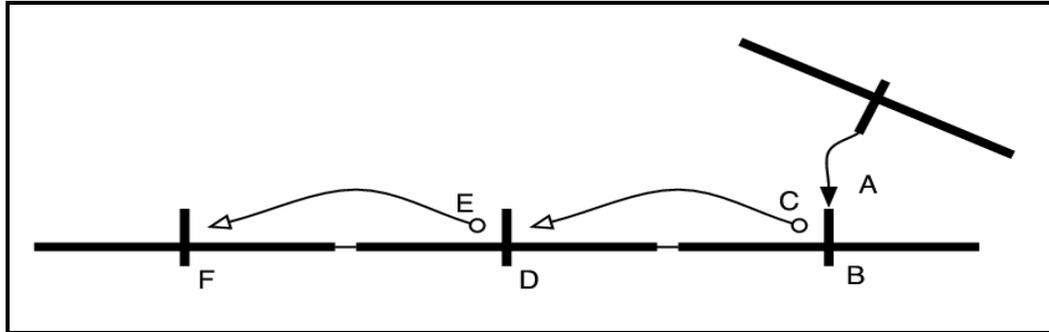
Figure 1. This diagram shows how *fold-counter* resets when a new machine joins a partially replicated gene. A free machine attaches to a machine in the strand (A), triggering the *reset-counter* signal and resetting *fold-counter* in the machine to which it attaches (B). The signal propagates (C) to the left neighbour (D), triggering its *reset-counter* signal and causing it to reset its *fold-counter*. In turn, the signal passes (E) to the next machine in the strand (F). The propagation stops at the leftmost machine (F).
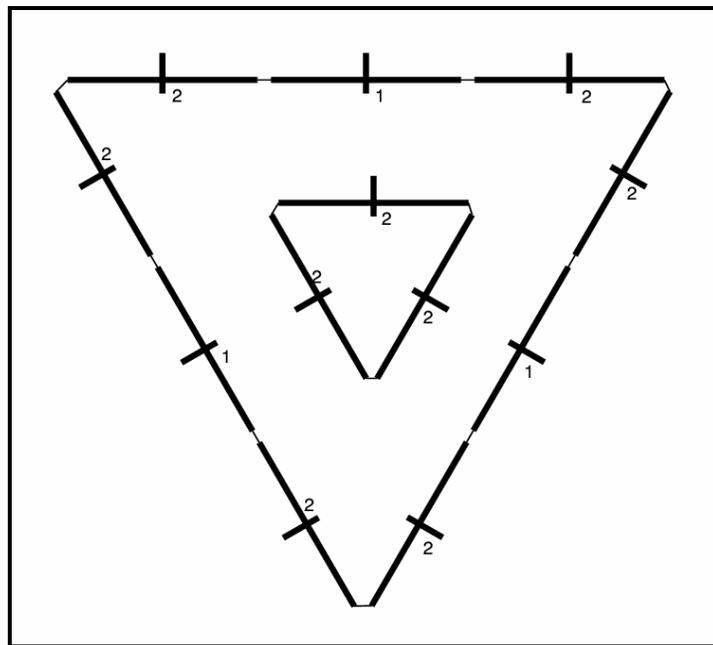
Figure 2. This diagram demonstrates the mechanism used to expand the size of a polygon. Machines of type 1 are inserted along the edges.
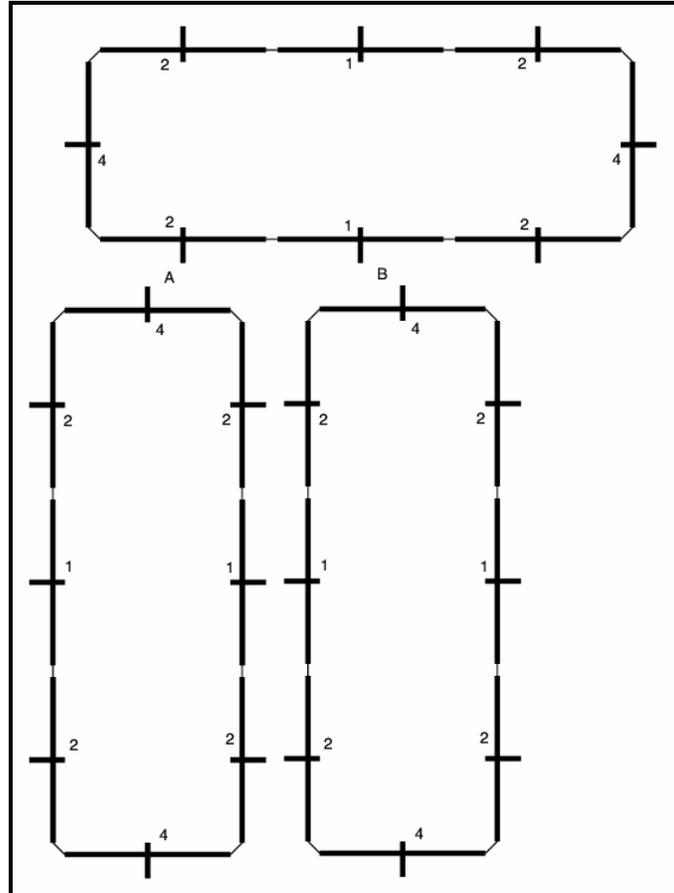
Figure 3. In this diagram, we can see why it is useful to have two types of machines for rectangles (types 2 and 4), in addition to type 1, for expanding size. The top phene is oriented incorrectly, relative to the mesh formed by the bottom two phenes. However, the top phene will not join at (A) or (B), because the types do not match (see Table 5). It will only join the mesh when it is oriented correctly.
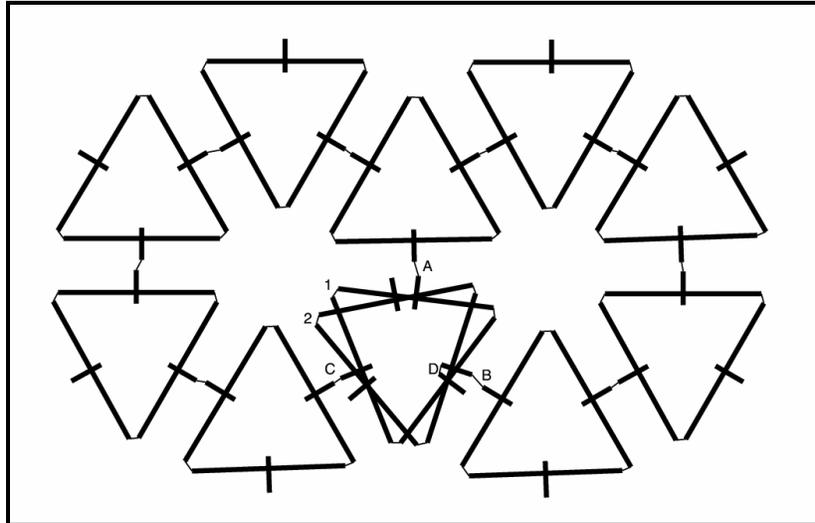
Figure 4. In this diagram, two phenes join a mesh, both trying to fill the same gap. Phene 1 has joined at (A) and (C). Phene 2 has joined at (B). Because these two phenes both have their *in-mesh* flags set to 1 (true), their overlap detector arms become active, and the two machines at (D) briefly bond their overlap detector arms. The machines then decide arbitrarily which one will set its *unfold* flag to true, causing the phene to which it belongs to unfold. The remaining phene will then pick up the bonds that were released by the unfolded phene.

Image: 1

Time: 0

Image: 2

Time: 2,385

Image: 3

Time: 3,295

Image: 4

Time: 44,235

Image: 5

Time: 46,340

Image: 6

Time: 49,780

Image: 7

Time: 85,810

Image: 8

Time: 294,075
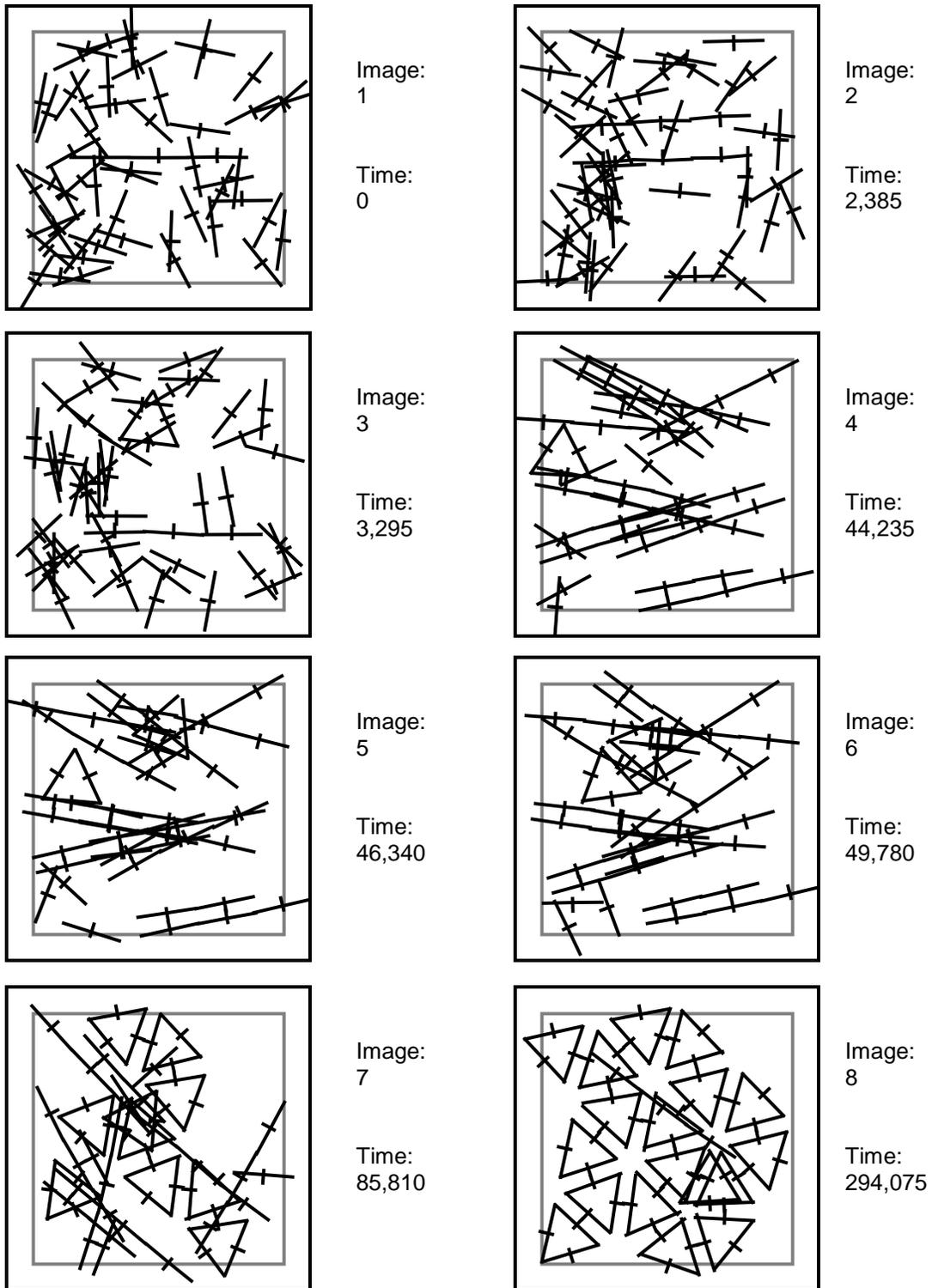
Figure 5. These images illustrate the experiment described in Section 4.1. The eight images are consecutive screenshots from a typical run of JohnnyVon, selected to show interesting stages in the simulation. The first image shows a seed gene in a soup of free machines and the last image presents a nearly completed mesh of triangles.
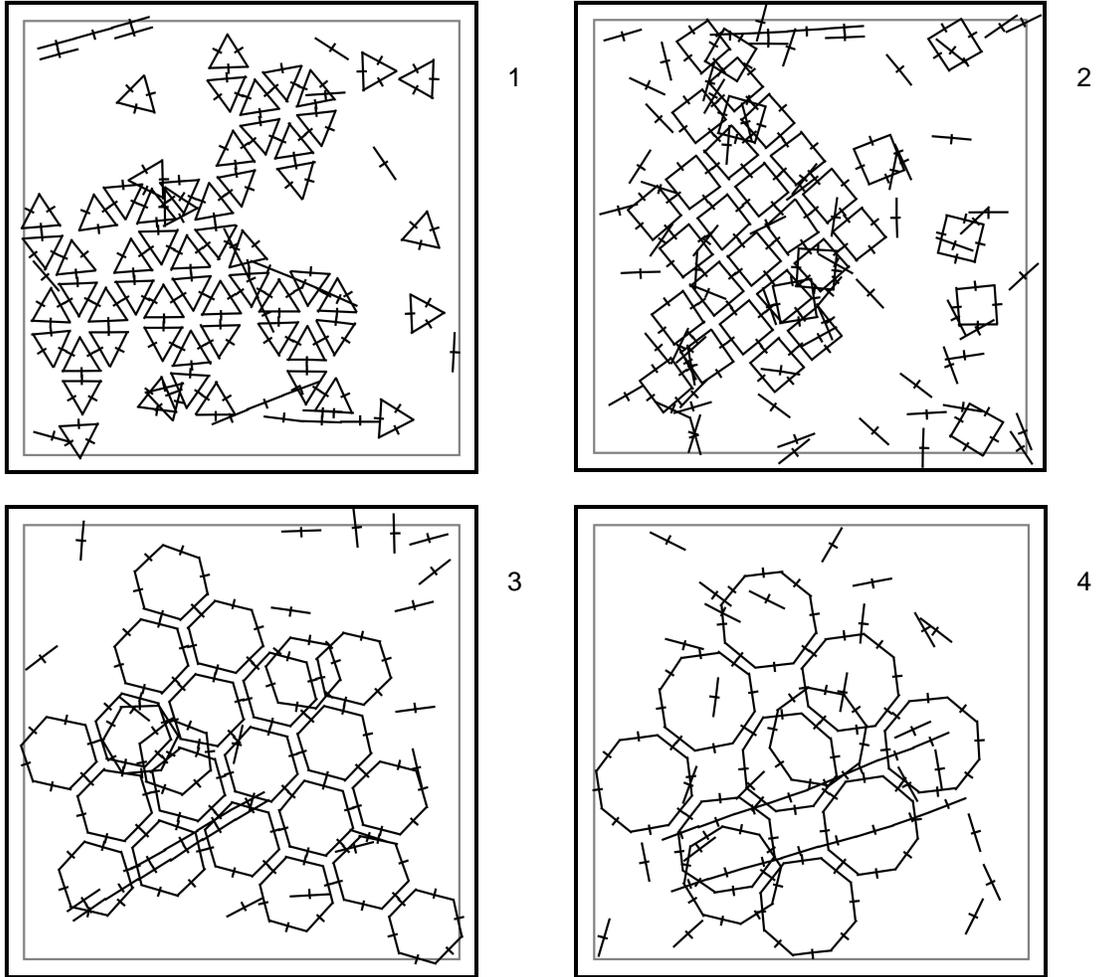
Figure 6. These screenshots illustrate the resulting meshes for each of the four supported regular polygons. The images show the final stages of four separate simulations, each using a different seed gene, as described in Section 4.2 and Table 9.
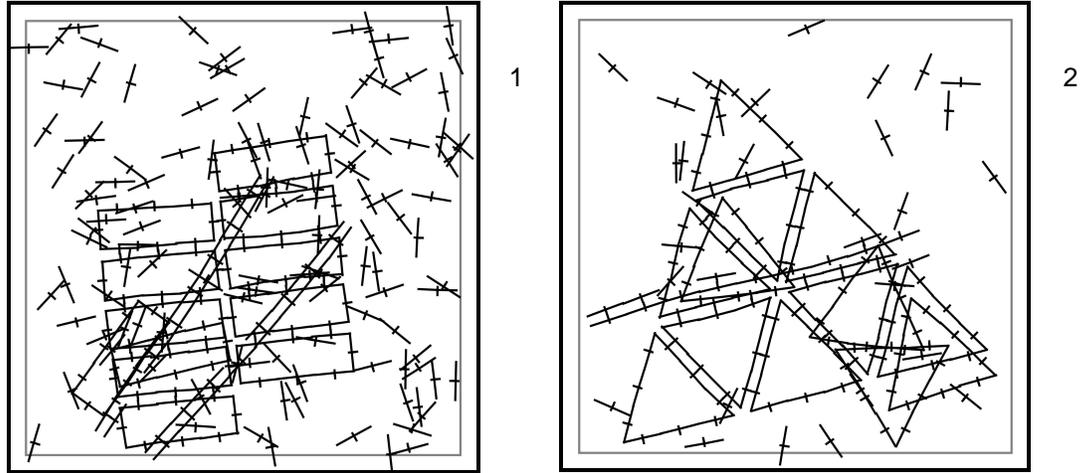
Figure 7. Image 1 shows a rectangular mesh and Image 2 is a mesh of expanded triangles. These are screenshots from the final stages of two separate simulations (see Section 4.3). They demonstrate that it is possible to vary the lengths of the sides of the polygons, by encoding the desired lengths in the seed gene.
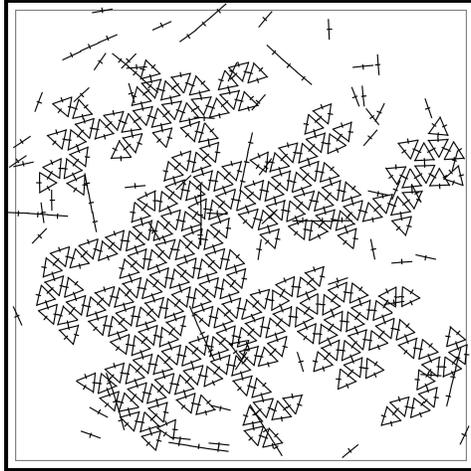
Figure 8. This screenshot shows a mesh of 234 triangles, as discussed in Section 4.4. This demonstrates that JohnnyVon can scale up to larger meshes.