Energy Based Entity Automata

Siniša Petrić

SigmaPi Design 2014, Revision 1.
www.sigmapi-design.com

***Abstract*—A new type of automata with complex internal structure used to present virtual entities in discrete, grid-based virtual Eco-system. Each automaton may change behavior depending on current environment condition. Furthermore, various entity types can be mixed in single grid-based environment in order to observe automata behavior in relatively complex systems. Each automaton is hierarchical structure, consisting of basic building blocks, called organs. Organs can be grouped in organ containers, which are finally grouped in automaton entity.**

***Index Terms*—cellular, automata, energy, environment, Eco-system, automaton, entity, bacteria, object.**

I. Introduction

VARIOUS automata systems are well known in mathematics, as well in computer science. From automata covering computational theory, to discrete grid-based automata, such as *cellular automata* [1] and automata used to model behavior of non-linear systems (*coupled map lattice* [2]). More or less, all automata are driven by externally defined rules. What does it mean? Well, speaking in terms of cellular automata, each cell depends on it's neighboring cell. New state of observed cell is (generally) defined as a function (rule) of current cell state and current state of it's neighboring cells. Despite this simplicity (at least by definition), cellular automata can be used to model very complex systems, however, each cell is still "stupid", i.e. it's state depends only on one rule. There is no internal cell complexity whatsoever. Cell is just a point inside the grid. Everything else (the rule) is somewhere outside.

In this article, I will try to build more complex cell structure in order to allow the cell to take some responsibility of it's behavior. Yes, some part of rule will still be outside the cell, but some part will be passed to the cell. Entity automaton state is change in *integer time* (stepwise), just like with ordinary cellular automaton.

Now, instead of term cell, I will use the term entity to give some "individuality" to our poor and brainless point inside the grid. Although this automata concept doesn't have intention to mimic living beings, some parallels with living organism ares necessary for better understanding.

Every living creature (and not only them) for their existence needs **energy**. Every living creature has some kind of **organs** (reproductive, organs for consuming food, etc.). Even single cell organism have organs, no matter if they are on molecular level. Also, every living organism has some kind of **processor** charged for processing information inside organs as well with the rest of organism.
Energy, organs and processor (process) are three main components of living organisms (at least in my perception) and with little abstraction it can be "ported" to objects in virtual world of automata. So, why not call that system organism automata? Well, term entity seemed more suitable for this level of abstraction. Organism sounds too much close to living beings and entity automata can be used for modeling something else, not only for emulating bacterial behavior or even more complex organisms.

And why energy based? Between those three main components: energy, organs and processor I vote for energy as the most important and something that's common to all things in the world. Again, I will draw a parallel with living organisms. Various organisms may have different organs and different processing structure, but energy is common to all of them. Let's take *Escherichia Coli* bacteria, for example.

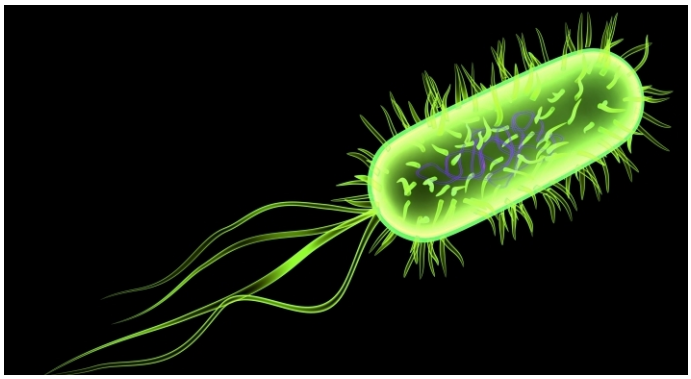


Figure 1. E. Coli

E. Coli belongs to a group of bacteria having specific organ called *flagellum*, an organ in charge for moving and rotating bacteria when searching for food[3]. Movement needs energy and bacteria doesn't have infinite amount of energy. Every movement of *flagellum* takes some amount of energy. Another organ of interests is *chemotaxis* sensor, charged for food detection. It also draws energy from bacteria. If bacteria does not find food it will die. It may go to some kind of hibernation (idle) state when energy buffer reaches "alarm" level, but hibernation state is consuming energy as well (although much less then in active state). So, when energy buffer reaches some energy threshold, it dies..as all living creatures, and decay process begins. When energy level reaches 0 value, organism does not exist any more.

Other bacterias do not have same organs as *E. Coli*, but energy buffer concept is the same. No energy, no living.

So this is the main reason for automata name: **E**nergy **B**ased **E**ntity **A**utomata, or **EBEA**. Furthermore, acronym **EBEA** (or **ebea**) will be frequently used in this article.

II. Definitions

FORMALIZATION of automata concept may be frustrating (and boring) but necessary. I will try to give simple and consistent definition (as much as possible) without going too deep in formalization. We will first start with organ definition. Organ is defined as a tuple consisting of three elements. Yes, they are: energy, organ and processor (process), but defining organ in such way would yield to self-referencing and because organ is some kind of basic, *atomic* structure, instead of organ we will use *terminal* element. It can be NULL (in terms of computer programming), it can be empty set (in terms of set algebra), or simply terminal symbol (in terms of formal language theory). We could also define *terminal organ*, prior to defining (normal) organ, but I leave that to mathematical purist. So, here is something that look like serious mathematics:

$$\omega = (E,\ T,\ P);\ T = \emptyset \tag{1}$$

where $\omega$is an organ, our basic building block. Each organ has it's own energy system, denoted as **E**, a terminal element **T** and it's process **P** (we could use $\omega_t$instead of letter **T** to be more consistent with the next definitions, but **T** will hold the water). If this organ present some kind of moving organ (flagellum) it uses energy from organ related energy system (buffer) and process **P** actually moves complete entity somewhere. Simple, isn't it? However, some organs have one or more common attributes, like sensors or movers (motor organs). Such organs, beside it's own energy buffer, may share common energy buffer and have common process as well. Those organs may be grouped together and we came to another definition, *organ container*:

$$\Omega = (E,\ \Omega_i|\omega_j,\ P) \tag{2}$$

Now what the hack is that? Organ container Ω may contain another organ container(s) as well as basic organ(s). Lets assume we have an *organ container* consisting of three basic organs and another *organs container (sub container)* . Expanding previous formula, we will get:

$$\begin{aligned}\Omega \quad = \quad & (E,(E,\omega_1,P),(E,\omega_2,P),\\ & (E,\omega_3,P),(E,\Omega_1,P),P)\end{aligned} \tag{3}$$

If organ container $\Omega_1$contains two basic organs, we finally get something like this:

$$\begin{aligned}\Omega \quad = \quad & (E,(E,\omega_1,P),(E,\omega_2,P),\\ & (E,\omega_3,P),(E,(E,\omega_4,P),\\ & (E,\omega_5,P),P),P)\end{aligned} \tag{4}$$

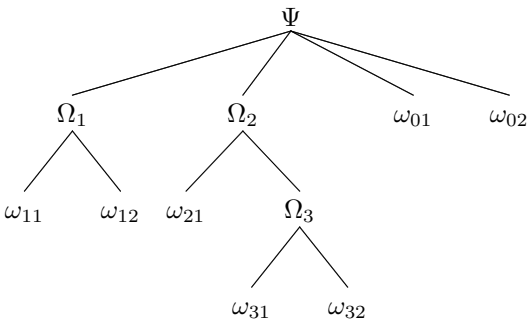
Of course, values for **E** and **P** in tuples are not equal, they just represents energy and process system. To distinguish those values, we can add some indices to them, and also rearrange indices of omegas:

$$\begin{aligned}\Omega \quad = \quad & (E_0,(E_{01},\omega_{01},P_{01}),(E_{02},\omega_{02},P_{02}),\\ & (E_{03},\omega_{03},P_{03}),(E_1,(E_{11},\omega_{11},P_{11}),\\ & (E_{12},\omega_{12},P_{12}),P_1),P_0)\end{aligned} \tag{5}$$

which now look more consistent, but still cumbersome. As you may notice, above organ container has it's own energy system and process system (processor) that are higher in hierarchy then same elements in basic organs. This can come handy if you want to create more complex entities. Organ container process P can be used to make decision which organ(s) will be active and when, depending on environmental situation. Also, two or more organ container can be grouped in new, higher organ container and so on and on...until we reach the master, i.e. entity:

$$\Omega_{entity} = (E,\ \Omega_i|\omega_j,\ P) \tag{6}$$

Entity is actually top-level, supreme *organ container*, with it's own energy system and process, used to distribute information to and from low-level *organ container* and organs. To distinct entity $\Omega_{entity}$from it's "subdued" *organ container* we will use another letter...let's say Ψ. Entity can be represented as general tree, which is more convenient then expanding tuples, as we did in expression (5). Here is an example:



Now, this representation is obviously more readable: our entity consists of two organ container and two basic organs. Furthermore, organ container Ω_1 consists of two basic organs and $\Omega_2$consists of one basic organ and one sub container, which again consists of two basic organs. From this representation, one constraint regarding process P, arise as natural:

Communication signal (as part of process **P**), can be sent from some organ (node), only to parent node (entity or organ container). Signal received by some organ is always sent from parent node.

Why? A simple reason: with this constraint, we will avoid communication paths to form cyclic graphs. We have a nice acyclic, tree structure...why to complicate things to much. We can formalize this statement, but I think it is not necessary

at this moment. **Energy flow** inside entity also obeys the same rule. Energy can be sent or received in the same way as communication signal. What's left? Well, we need to expand terms like energy system and process. Energy system **E** of some entity element, let's say an organ, consists of maximum energy level (energy buffer size), current energy level, energy consumption per time step, alarm value and...some other stuff that comes to your mind, which are related to energy. For now, let's keep it simple:

$$E = (B_e,\ e,\ \delta e,\ \xi e,\ A_e, D_e) \tag{7}$$

where $B_e$is energy buffer size (maximum level), e is the current energy level, δe is energy consumption per time step when organ is in active state, ξ_e is energy consumption in hibernated (idle) state and $A_e$is alarm value. Alarm value is minimum acceptable energy level, When current energy level reaches alarm value, organ (or organ container) may send a signal to higher organ container (entity) to notify it of current crisis. Entity can either send energy from reserves (or borrow it from other organs) or trigger organ (organ container) to go to hibernate state. If nothing is possible, energy consumption goes on and on until entity dies. Entity dies when value of idle state threshold D_e is reached. Entity sill has some energy, but it's actually dead. It can be only used as a food for scavenger entities. Energy consumption continues, but entity is in decay stage. When energy level 0 is reached, entity does not exists any more. Entity overall current energy level can be expressed as a sum of all energy levels of it's constituting elements:

$$e(\Psi) = \sum_{i=1}^{n(\Psi)} e_i \tag{8}$$

where $n(\Psi)$ is number of nodes in entity, including the root node, and e_i is current energy value of each node. This value becomes important when one entity is actually food for another entity. When eating process is finished, it represents the amount of consumed energy. And finally...process (processor). Process consists of communication system and organ related action:

$$P = (Cs,\ \alpha) \tag{9}$$

Communication system is always bi-directional (duplex) and organ related action is for example: move complete entity to another point (if organ is mover), or eat the food (if organ is eater), or scan environment (if organ is sensor), etc, etc...which leads us to organ types.

III. Organ types

ORGANS can be defined (and programmed) according to our needs, however few specialized types impose themselves as natural:

1) *Mover*: an organ charged for moving complete entity from some point **A** to some point **B**.
2) *Eater*: an organ charged for consuming food and converting it to energy. After food is converted to energy, Eater organ's process, sends signal to upper level to take and distribute energy across the system.
3) *Reproductor*: a reproductive organ. It's main task is to produce another entity of the sane kind.
4) *Sensor*: sensing organ. Scans environment searching either for food, or for friends and foes. Sensor organ's process, generally keeps a list of food (possibly in preferred order) and mandatory, a list of friends and foes.

Do wee need more then one specialized organ (let's say mover) per organ container/entity? This is quite an interesting question. Drawing a parallel with *caterpillar* we may construct a bunch of same mover organs (legs) and group them in some organ container. But, how we shell handle them in program implementation? Will our entity move faster with so many legs? Well, we can do that, but it would complicate our task. We can say: OK, entity with more then one mover organ can traverse bigger distance per one step and more mover organs, bigger the distance. Nice, but now what about *cheetah*? Cheetah has only four legs, much less then caterpillar, but moves much faster. So, this problem is not easy to solve,and for sake of simplicity, in this paper (implementation), only one specialized organ per organ container/entity will be used.

Now, what about organs order of creation? Entity, depending on environmental situation, must decide if it will move, scan the environment or reproduce. Organs creation order, tells entity which organ's process has precedence. If we have created entity with three organs in the following order: *mover*, *reproductor* and *eater*, entity will first try to move, then will try to reproduce and after that will try to eat. Of course, we can add some value to each organ, indicating precedence probability, but I leave that for next version of EBEA implementation.

IV. Observer

SO FAR we were playing with definitions and descriptions of entity structure and, as proposed in introduction, we have "passed" a part of automata rules to entities via processes **P**. Entity processes dictate entity behavior inside environment. However, at this moment we can not pass complete responsibility to entities. So, we came to observer's point of view. Observer, or more precisely environment designer is responsible for creating initial environment (grid size, entities seeding) and of course, imposing some rules that can not be handled by entities themselves.

As previously stated, cellular automata (CA) works by changing state of observed cell **X**, depending on it's current

state and state of surrounding (neighboring) cells (according to some rule). New state of the cell **X** is not taken into account when this cell becomes neighboring cell of newly observing cell **Y**. New cells state is transferred to new "environment", when complete grid is scanned and all cells states evaluated. Similar to applying some convolution filter to image. I like to say that CA is static-evolving system. From cells viewpoint, it's static and from observer's viewpoint, system is perpetually evolving, i.e. system is dynamic.

EBEA works quite different and is somewhere between CA and *particle system* [4], especially when entity movement (or reproduction) occurs. When entity **X** is observed, it's new state immediately influences the state of entity **Y**. Note: in some of my previous papers, I have treated CA in exactly the same way...which is not the way CA operates. However, I can always use term quasi-CA, or non-strict-CA. So, EBEA is natural successor of "quasi-CA" and it can be viewed as combination of CA and *particle system*..to some extent . Enough apology, let's get back to business.

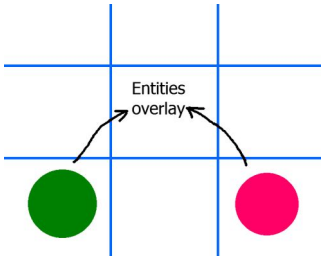


Figure 2. Overlay problem

When entity moves to the free cell inside grid, it occupies that cell and releases the previously occupied position. If we do not "refresh" our environment with new situation, another entity may occupy the same cell. However, in regular CAs, the way we are scanning the grid does not effect the new automata state in no way.

With EBEA, the things are different. We do not scan complete grid. Entities (entity pointers) are usually stored in some array and if we loop through array to detect new entity position, entities with smaller array indices are privileged, as they have precedence of moving and occupying cell. We can avoid such biasing, using array shuffle, but still same problems persists. Besides shuffling, we can use several methods to get desired results:

1) We can introduce hierarchy to entities of the same species and order array accordingly. Entity higher in hierarchy moves first.
2) We can allow occupying the same cell by more than one entity and then treat that situation as *meta-state* or perform entities collision, like with *particle system*. We can allow that two entities occupy the same cell, if one of the entities is food to another. This, however involves *move and eat* mode of operation, i.e. food consumption is performed when entity **X** overlays entity **Y**, which is in the food list of entity **X**.
3) Again, we can allow occupying the same cell by more the one entity ("subcell precision") and say: this is claustrophobic, "suffocating" situation which will trigger some entity process **P** to resolve it. Either by driving away competition (another entity) in some kind of entities battle or by decreasing entities energy level, due to the lack of space (suffocation).

I will leave all possibilities opened for discussion and further works in this area, as it may yield to interesting and complex situations. Current implementation simply does array shuffle and does not allow that single cell is occupied by more then one entity. The same situation can occur when entity's reproductive organ is triggered. Also, in current implementation, *eater* consumes food only from neighboring entities. There is no "food entity overlay"...maybe in next implementation, which will be presented in second revision of this article.

So, what is the responsibility of observer? I use the term observer and not designer (or creator), because every observer affects the world he or she observes. Yes, it happens on quantum level, but I like to use this concept everywhere I can. So I used it on EBEA:

1) Observer creates a 2D space (grid) of some particular size.
2) Designs and creates entities and seeds them in space.
3) Keeps track of every entity position (x, y) inside the grid.
4) Resolves conflicts inside the grid (multiple entity in single cell)
5) Presents visually every step in environmental change.

And that's it...more or less. In next section I will give a brief (very brief) explanation of implementation, which is still in rudimentary phase, but some ideas are behind EBEA can be well visualized.

V. Implementation

IMPLEMENTATIONof EBEA is done in C++, but it can be implemented in any other Object Oriented Programming (OOP) language. OOP is natural choice because implementation starts with class called *base organ*. All organs are derived from this base class; eater, mover, reproductor, base organ container and entity. The idea is to be able to dynamically create various entities with various organs, energy systems and processes. For now, implementation is in rudimentary phase and I have experimented only with simple entities, consisting only of simple organs (without organ container).
Also, I have created some predefined, specialized entities consisting only of one, base organ: *mana* and *obstacle*:

- *Mana* is entity consisting of only one organ, base organ. It does not move and is always in idle state, with very low energy consumption. Mana is general food and it can be used to emulate *agar-agar*.
- *Obstacle* is also base organ entity. I does not move, it's not a food and it does not consume energy. It stay in one place forever.

Here is an example of environment situation with one entity type, consisting of two organs: *mover* and *reproductor*. Reproductor is simple asexual reproductive organ that works on *binary fission* bases. It splits entity in two. One entity stays in place, while other is dislocated in randomly chosen free cell. Here is a code snippet used to create entity with two organs:

```
// create entity
entity = new entityOrgan(1000.0,1000.0,10.0,1.1,0.1,
                ORGAN_CONTAINER, entity_id,2,iPos);
// add mover organ
entControl->addOrgan(entity,1000.0,1000.0,10.0,
                0.01,0.1,ORGAN_MOVER);
// add reproduction organ
entControl->addOrgan(entity,1000.0,1000.0,100.0,
                0.1,10.0,ORGAN_REPRO);
```

First line creates new entity type. Second line adds mover organ and third one adds reproductive organ. As previously stated: int this implementation version, organs adding order is important and actually tells entity that moving is more significant then reproduction (at least in this example). Floating point values describe energy system: buffer size, initial energy value, active state energy consumption, idle state threshold and idle state consumption.

As you may notice, alarm value is not implemented and because there is no *mana* in environment, entities will only move and reproduce, but will die fast...in less then 1000 iterations.

White pixels represent live entities, red pixels[1] represents dead entities, while green pixels are entities traces (in process of moving and reproducing). Intensity of green pixels falls-off with each iteration. Each image in this example is actually environment snapshot took every 100 iterations.

VI. Conclusion

ENERGY based entity automata concept is still in its infancy and lot of work has to be done in order to cover some "real-world" problems. Current implementation allows only playing with few entities and mouse-driven seeding method (spray). My first intention with EBEA was to produce fancy brush strokes for artistic purposes, but idea evolved a bit by time. With little more efforts, current EBEA implementation can be used to simulate bacterial movement and reproduction on agar plate, and that is the situation where *mana* enters the playground. Beside implementation, a lot of work must be done on theoretical part of EBEA. For instance,

[1]Unfortunately, images were saved as lossy JPEGs and red pixels are not visible.

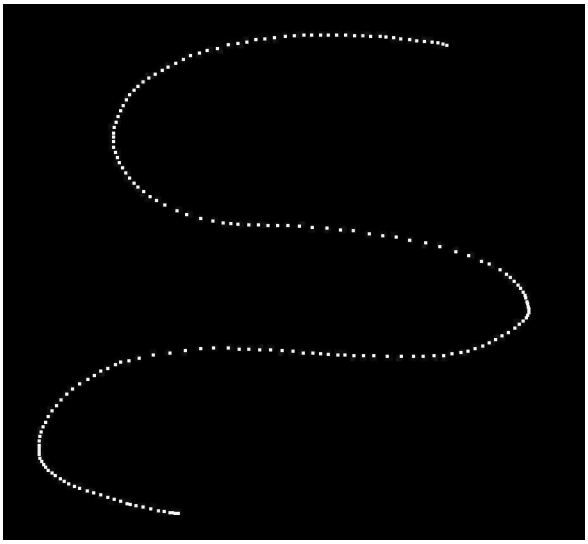


Figure 3. Seeding entities

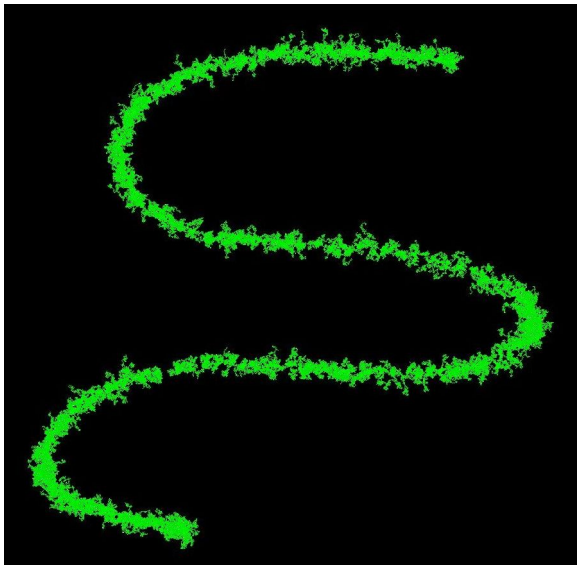


Figure 4. Snapshot 1 - 100 iterations

EBEA can be described with *formal language* production rules. Something like this:

$$S \rightarrow (E,\ A,\ P)$$

$$A \rightarrow (E,\ A|T,\ P) \mid A,\ A$$

$$T \rightarrow \varepsilon$$

where E, A, P are elements of respective subsets of **ebea** alphabet, and so on. Production rules are defined *ad hoc*, and they do not look so good from *formal language* point of view, but again, they will hold the water. That's all folks...until new implementation (program) version (with new examples and fancy images) and of course, article revision number 2.

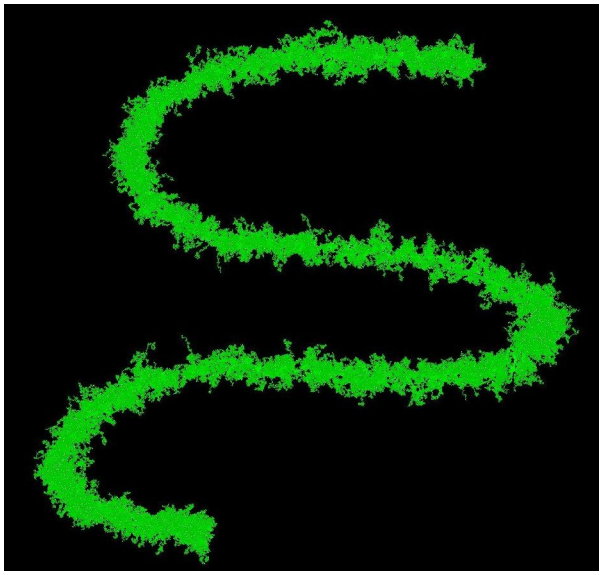


Figure 5. Snapshot 2 - 200 iterations

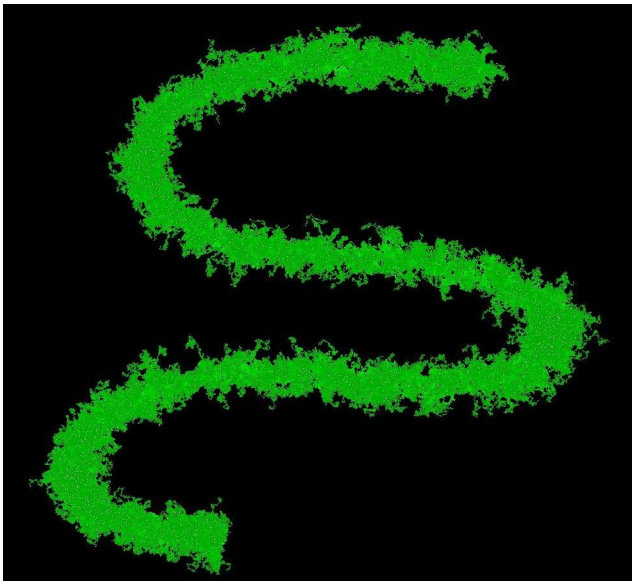


Figure 6. Snapshot 3 - 300 iterations

REFERENCES

[1] E.F.Codd. Cellular automata. *Academic Press*, 1968.

[2] K. Kanenko (Ed.). Theory and applications of coupled map lattices. *Wiley, New York*, 1993.

[3] Richard M Berry. Bacterial flagella: Flagellar motor. *The Randall Institute, King's College London, London, UK*.

[4] William T. Reeves. Paricle systems - a technique for modeling a class of fuzzy objects. *Computer Graphics*, 17, 1983.

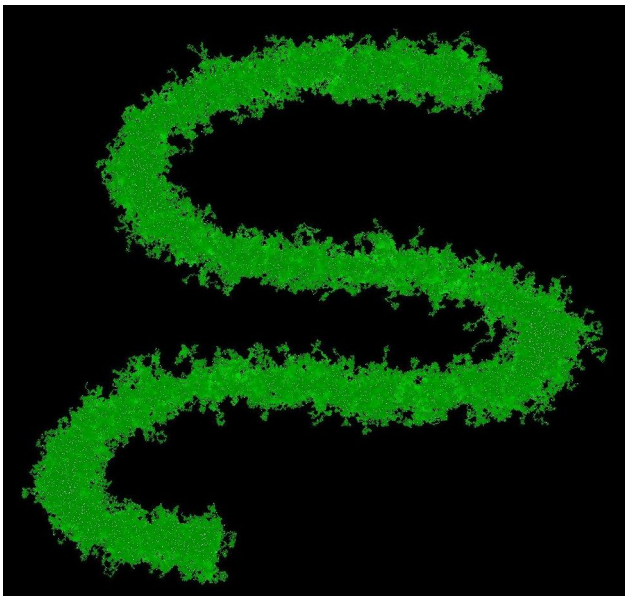


Figure 7. Snapshot 4 - 400 iterations

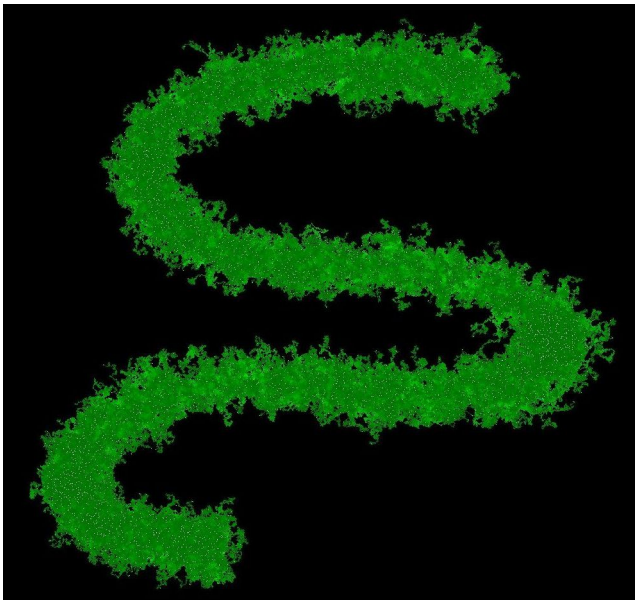


Figure 8. Snapshot 5 - 500 iterations

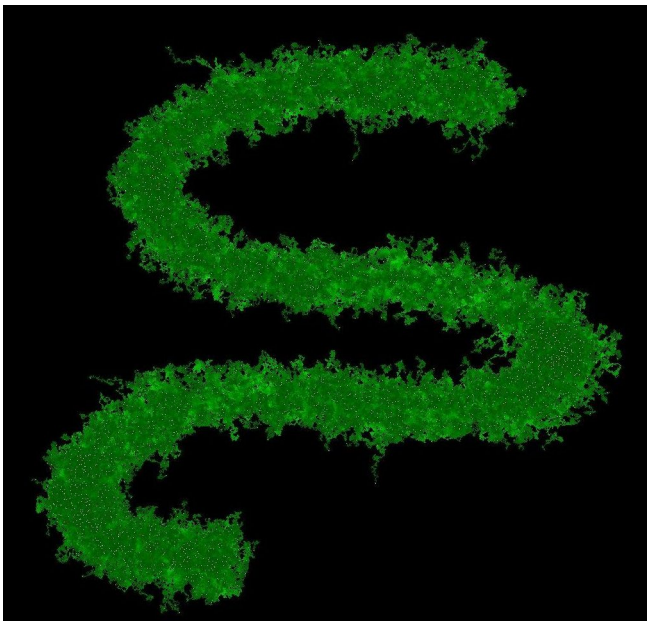


Figure 9. Snapshot 6 - 600 iterations

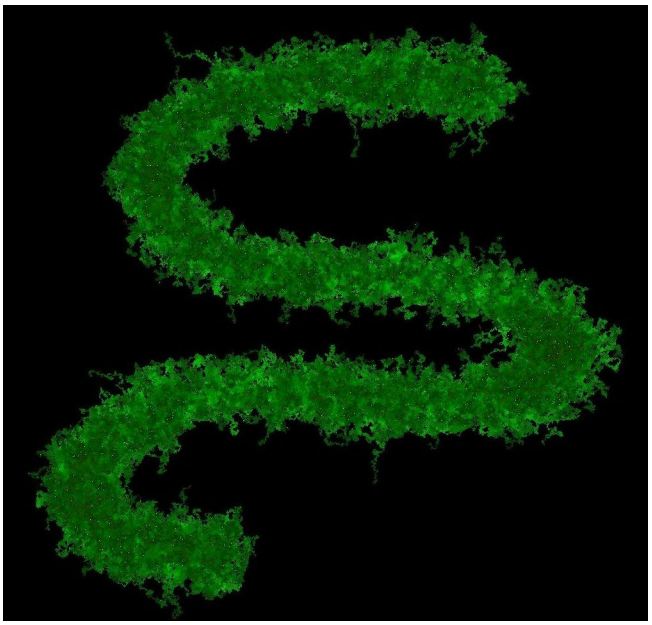


Figure 10. Snapshot 7 - 700 iterations

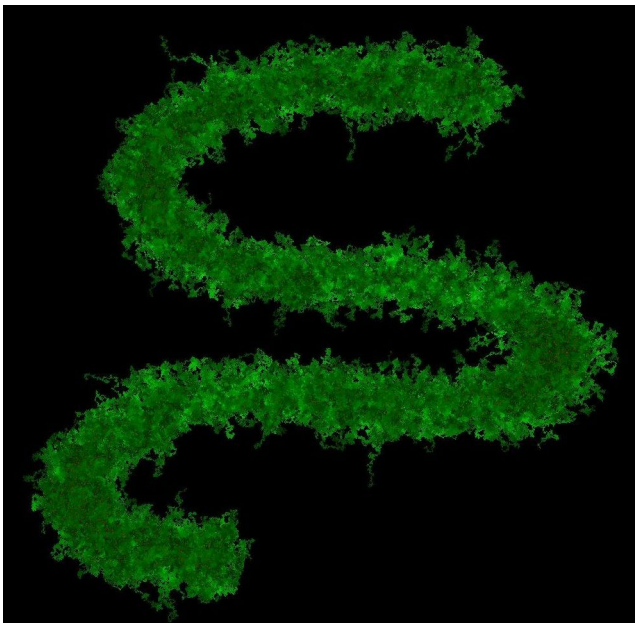


Figure 11. Snapshot 8 - 800 iterations