UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Implementation of the efficient truly two-dimensional artificial life simulation

Author:
Hermann YAVORSKYI

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences

Lviv 2021

Declaration of Authorship

I, Hermann YAVORSKYI, declare that this thesis titled, "Implementation of the efficient truly two-dimensional artificial life simulation" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"If you can't fly then run, if you can't run then walk, if you can't walk then crawl, but whatever you do you have to keep moving forward."

Martin Luther King Jr.

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences
Department of Computer Sciences

Bachelor of Science

Implementation of the efficient truly two-dimensional artificial life simulation

by Hermann YAVORSKYI

Abstract

This work aims to develop an efficient implementation of the two-dimensional artificial life simulation Fungera (Yavorskyi, 2021, GitHub). Initial Python-based implementation of the Fungera allowed to verify that the two-dimensional approach is viable to study evolutionary processes. Nevertheless, this prototype implementation is too slow to study emerging evolutionary innovations and emerging ecological relations efficiently. One-dimensional simulator Tierra, the first successful artificial life evolution simulator, required several million iterations for the emergence of parasites, hyperparasites, "social behavior", and so on. Because of its two-dimensional nature, Fungera would require orders of magnitude more time for such events leading to many weeks of simulations using Python-based implementation. So, this work aims to develop efficient implementation based on C++. It should support easy modification of the instruction sets used by the simulation. The other way this project can be considered as an experiment on two-dimensional computer architectures, inspired by the esoteric programming language Befunge.

Acknowledgements

I am grateful to my supervisor Oleg Farenyuk for showing me the world of artificial life research and leading me through this journey and to Olexandr Syzonov and Pavlo Yasinovskyi for all contributions we made as a team researching the ALife topic.

I would like to thank my classmates Yulianna Tymchenko and Andrii Koval for being on the same page during this work and showing support.

I really appreciate the impact of Olesia Tretiak on this work since she believed in me more than I did.

Contents

List of Figures

List of Tables

List of Abbreviations

RNA	**R**ibo**n**ucleic **a**cid
CPU	**C**entral **P**rocessing **U**nit
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata
RAM	**R**andom **A**ccess **M**emomry
IP	**I**nstruction **P**ointer
DNA	**D**eoxyribo**n**ucleic **a**cid
IPC	**I**nter-**p**rocess **C**ommunication
ISA	**I**nstruction **S**et **A**rchitecture
OS	**O**perating **S**ystem
NOP	**N**o **OP**eration
GUI	**G**raphical **U**ser **I**nterface
TUI	**T**erminal **U**ser **I**nterface
MVC	**M**odel-**V**iew-**C**ontroller
XML	e**X**tensible **M**arkup **L**anguage

Dedicated to all the amazing people that I met during our 4-year studies

Chapter 1

Introduction

1.1 Foreword

In an ideal world, biology should study all forms of life, while in reality, it is concentrated only on life on Earth (Ray, 1991, p.1). This fact means that there is an open niche in research of alternative forms of life. Artificial life takes this free space. Artificial life means "*life created by human rather than by nature*" (Langton, 1995). In this work, the term is used to refer to the simulations and studies related to natural life but simulated with computers. These researches aim to find reasons for the natural processes and patterns in the living organism. However, there is no possibility to study them by examining organic creatures.

1.2 Motivation

This work is very tightly related to Fungera (Poliakov, 2020), the artificial life simulation, which was a proof-of-concept for the possibility to implement simulation with the truly two-dimensional memory space. Previous artificial life simulators showed remarkable evolutionary results. For example, in the Tierra simulation, the occurrence of both punctuated equilibrium (Wikipedia contributors, 2021b) and phyletic gradualism (Wikipedia contributors, 2018) was shown (Ray, 1991, p. 15). The original Fungera simulator showed the punctuated equilibrium model of the evolution. The current results and diversity of organisms prove that the Fungera model is viable to research to achieve more evolutionary results. "Evolution of digital organisms in truly two-dimensional memory space" (Poliakov, 2020) describes the results obtained up to the simulation cycle 700000. Other simulations required significantly more time in order to achieve their results. Taking the dimensionality into consideration, significantly longer executions are required to develop new evolutionary patterns or exhaust their opportunities.

1.3 Goal

The goal of this work is to create software that allows study Fungera more efficiently. This includes implementing the simulator in C++, which is compiled programming language significantly faster than Python. The implementation needs to be extensible in order to allow modifying instruction set without great effort. The GUI (graphical user interface) will also be beneficial for real-time visualization of the system and troubleshooting in case of the extension of the instruction set.

Chapter 2

Related Works

2.1 Tierra

2.1.1 Model description

An Approach to the Synthesis of Life is a work of Tom Ray published in 1991, which aimed to show the origin of the diversity of life. The research involved engineering complex, evolvable organisms and creating conditions that will lead to the evolution process with increasing diversity and complexity. It is one of the most successful and well-known artificial life simulations. The Tierra simulator in its core is an artificial world that can be referred to as the virtual version of the RNA world of self-replicating molecules. The simulation resulted in the emergence of such interactions as parasitism, hyperparasitism, sociality, and cheating from just one rudimentary ancestral creature, which consists of code only for self-replication (Ray, 1991, pp. 2-3).

Organic life needs two primary resources — energy, mainly derived from the sun, and space to live in. Artificial life parallel for the sun is CPU, and memory is the analog for the spatial resource. There is a natural selection mechanism in such a simulation: organisms compete for CPU time and memory space and evolve strategies to do it more efficiently and exploit each other. Digital organisms are computer programs — creatures constructed entirely of machine instructions that can be referred to as analogies for amino acids. The ”genome” of a creature is the sequence of machine instructions that form the creature’s self-replicating algorithm. The prototype creature has a ”genome” of 80 machine instructions (Ray, 1991, p. 5).

2.1.2 Virtual computer

The simulation takes place on a virtual MIMD type computer. This virtual machine’s machine code is designed with evolution in mind in contrast to the real machine code, which is almost certain to result in invalid and non-functional programs because of mutation or recombination events. Each organism has its own processor. There is no true parallelism; it is emulated by allowing each creature to execute in a small-time slice in turn. Each CPU has two address registers, two numeric registers, an error flag register, a stack pointer, a stack, and an instruction pointer. The instruction set includes simple arithmetic operations, bit manipulation, moving data between registers and RAM, IP manipulation instructions. IP indicates the position in RAM where the code of the organism is currently executed. Code is executed in the fetch-decode-execute-increment-IP style. The instruction is fetched from the RAM; then, bits are decoded to determine the corresponding command. The command is executed, and IP is incremented to the next point in memory (except commands that

directly manipulate the IP, such as JMP, CALL, RET). The RAM is one-dimensional and shared between all CPUs (Ray, 1991, p. 6).

2.1.3 Instruction set

Traditional CPU instruction sets are not suitable for genetic operations such as mutation. A new instruction set was developed. This virtual programming language, "Tierran", was designed to be the same order of magnitude as the genetic code. The information in DNA is encoded through 64 codons, which consist of 20 amino acids. Tierran language consists of 32 instructions which are represented by five bits. It is achieved by restricting the operands to be only registers or stack. Another crucial feature of the Tierran language is addressing by a template. The template set of the instruction NOP_0 or NOP_1 follows the JMP command. When the JMP command is executed, the CPU searches for the pattern, inverted to the given in both directions. For example, on the execution of command JMP NOP_1 NOP_1 NOP_0, the system will search for the sequence NOP_0 NOP_0 NOP_1, and in case it was found, the IP will be moved after this sequence. Other than that, instructions are similar to most "real" instruction sets (contains such standard instructions as MOV, CALL, PUSH, POP) (Ray, 1991, p. 7).

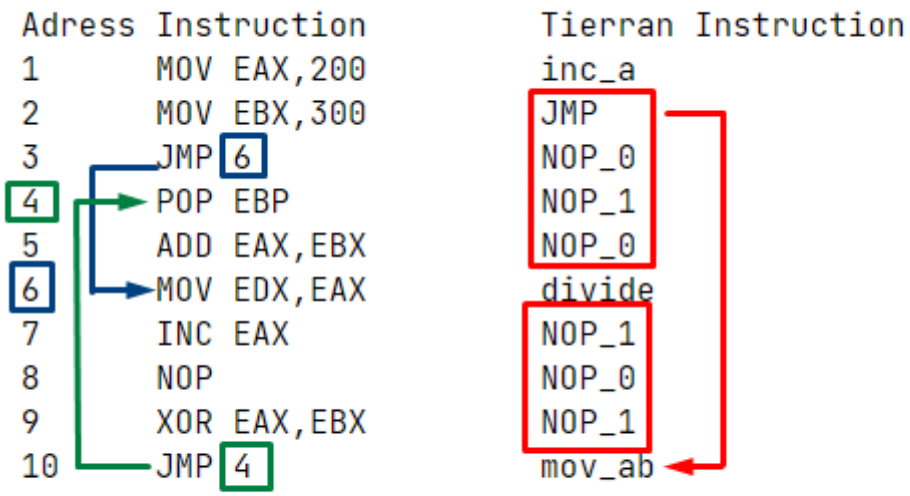

FIGURE 2.1: Normal addressing and template addressing

2.1.4 Operating system

In order to work, the virtual machine needs the operating system, which defines the mechanisms for IPC, memory, and CPU time allocation. The Tierran operating system establishes the environment. In combination with ISA, it also describes the topology of possible interactions between creatures. OS operates a block of RAM, which is referred to as "soup". Each organism has its block of memory in the soup. The OS provides cellularity so that memory allocations protect creatures as a "semi-permeable membrane." Each organism has exclusive write permissions in its block

of memory. Read and execute permissions are not restricted from other organisms that are not owners of the memory block. Each creature can have the write privileges in at most two memory blocks: its memory block and the block of the child. Memory block for the offspring is received through the execution of memory allocation instruction. When the organism "divides", the creature's child becomes an independent organism. It obtains its registers, IP, stack, and CPU. The parental organism, in its turn, loses the write permission to the child's memory block (Ray, 1991, pp. 8-9).

Multitasking allows organisms to "live" in the memory simultaneously. It is implemented through a "slicer queue" -— a queue of all creatures (each having its virtual CPU). These virtual CPUs receive slices of real CPU time in turns. Since the time slices are small relative to the generation time of the creatures, the author calls it approximate parallelism. There is no true parallelism in such an approach. The "slice power" — a variable that defines whether the "slicer queue" is size neutral or if it favors the organisms of large or small size can be configured. If power is greater than 1, large creatures get more CPU cycles per slice. If it is less than one, then tiny organisms get more CPU cycles (Ray, 1991, p. 9).

In a fixed-sized memory soup with self-replicating creatures in it, there is a problem with overpopulation. This problem is solved by implementing mortality. All organisms, when born, are put into the "reaper" queue. When the memory allocated for simulation fills to a set percentage, it begins to "kill" the creatures from the top of the reaper. When the organism is killed, it is removed from the "reaper" and the "slicer queue," its memory block is deallocated. However, the organism's "dead" code remains in the soup. Organisms in the queue are sorted by the number of errors they made during the execution of machine instructions. There are two relatively challenging to execute machine instructions. Successful execution of these instructions moves the organism down in the queue as long as it has not more errors than the organism below. The reaper queue eliminates organisms with broken or seriously damaged algorithms and frees the memory for new organisms. In general, the older creature gets, the higher is the probability of death (Ray, 1991, p. 10).

Evolution is a change in the genomes of the creatures over successive generations. Tierra simulator implements it in three aspects. At some set background rate, bits are randomly selected from the entire soup and flipped, preventing organisms from immortality as each organism will eventually mutate to death. The mutations may also happen during the replication of organisms: the bits are randomly flipped at some rate during copy instruction execution. The rate of mutations during replication is higher than random bit flips. Both mutation mechanisms are set to happen in variable intervals to prevent possible periodic effects. In addition to the mutation mechanism, the behavior of the Tierran instruction set is probabilistic by its nature. Most of the instructions can be executed with a flaw: the result can be off by 1. For example, an increment command can add 0 or 2 instead of 1, or a bit-flipping instruction flips the next higher bit or no bit. All these possible mutations in the genomes of organisms cause new operational genotypes to evolve through time and provide open-ended evolution (Ray, 1991, pp. 10-11).

An automated genebank manager is implemented to watch the creation of new genotypes that appear due to Tierra's evolutionary mechanisms. When new genotypes replicate twice with both children being genetically identical, they are given a unique name and saved to disk. With each genotype, some additional data is stored, e.g., the name of its ancestor, which allows building a family tree, time and date of origin, number of errors generated in the first and second reproduction, and other information. (Ray, 1991, p. 11).

2.1.5 Results

Mutations can not change the creature's size directly. However, they change instructions that form the organism. These changes in template patterns (used in such instructions as JMP) may change how the organism examines itself, potentially changing its descendant size or behavior. As a result, such mutation changed the low order bit in instruction NOP_0, converting it into NOP_1, changing the template from indicating copy procedure to the end of the organism. These changes caused the emergence of parasites. For example, popular Tierra genotype 0045aaa is not self-replicating. However, due to the ability to read and execute instructions of different organisms, it can use the initial ancestor to perform the ancestor's copy procedure forming parasitic relationships. Other behavioral patterns that appear in Tierra simulation (Ray, 1991, pp. 13-14):

- Immunity to parasites — some of the size 79 genotypes demonstrated to be resistant for parasites. Parasites of several genotypes are still able to exploit their code for their goals. However, the reaper queue will quickly eliminate parasites' children from the soup.
- Other parasites can still successfully reproduce using immune to parasite hosts, therefore developing circumvention of immunity to parasites.
- Hyper-parasites are creatures that can use parasites for reproduction. This kind of organism manipulates the instruction pointer, so after the copy procedure, it executes not RET instructions but JMP to the proper address of the reproduction loop, effectively stealing the parasite's IP. Hyper-parasites can form groups.
- Groups of hyper-parasites are invaded by cheaters: hyper-hyper-parasites. They steal the IP from hyper-parasites when it is passed from one member of the group to another.

In terms of macro-evolution, the results are dependent on the "slice power". When selecting from the small organism, the simulation shows interesting communities of tiny parasites. Selection for significant cases usually leads to the continuous increase of the creatures' size. Long runs with size-neutral selection illustrate the pattern of interchanging periods of stability with periods of rapid evolutionary changes. The run that lasted 2.56 billion instructions contained 1180 size classes of organisms where each size class can consist of several distinct genotypes (Ray, 1991, pp. 15-16).

2.2 Avida

Avida is an artificial life simulation inspired by Tierra. Chris Adami developed it because of the need for a larger and faster system designed for evolution towards complexity (Adami, 1994, p. 1). The software distributed with Avida includes the following applications (Ofria, 2003, p. 11):

- Avida core, which provides virtual hardware and services parallel to Tiera's OS
- Analysis and statistics tools, with test environment which allows studying organisms outside the environment

- GUI, which allows interaction with previous parts

Avida proposes a memory model in the shape of a torus, the surface of which is the grid (Adami, 1994, p. 1).

Although memory is described as an M x N grid, the model is pseudo-two-dimensional. Organisms are not two-dimensional. Each organism is allocated a sweep of the torus. This sweep is circular. Therefore when the last instruction is executed, execution continues from the beginning of the creature. Despite being circular, it has a defined starting point necessary for the self-replication process. The organisms are isolated in their stripes of memory and can be affected only by the command which splits the new organism from its ancestor. In this case, the new organism can replace the old one (Ofria, 2003, p. 11).

The core principle of the self-replicating organism is similar to the Tierra: its initial state can be described by the string of symbols (instructions). This string is the genome of the organism. However, the virtual hardware is different. The memory has already been described. Avida's virtual CPU has two stacks. Only one can be active. However, there is an instruction to switch active stack. The CPU also has four labeled heads. These heads are pointers to the positions in memory. One of the heads is the instruction pointer. The other two are "write" and "read" heads. They are used in the self-replication process. The first one indicates the position in the memory to which the instruction at the read head position is being written. All heads may be placed only inside the memory allocated for this organism. In this way, an organism in Avida has restricted write, read and execute permissions, in contrast to Tierra, where only write permissions are restricted. Since the organism is not able to write in memory which is not part of him when memory for the child is allocated, organism size is increased (Ofria, 2003, p. 12).

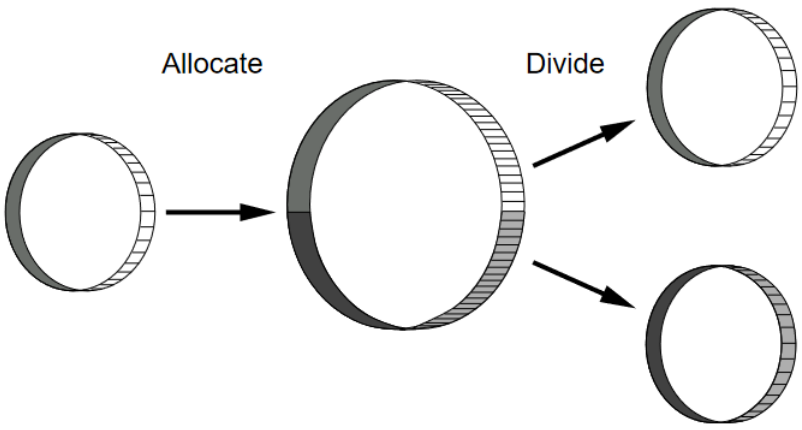

FIGURE 2.2: The h-alloc command extends the memory, so that the program of the child organism can be stored. Later, on h-divide, the program is split into two parts, one of which turns into the child organism (Ofria, 2003, p. 74)

The last head is the flow control head which is used for jumps and loops. It is worth mentioning that this head also is used to implement addressing by the template, the mechanism similar to the one in Tierra (the difference is that Avida has 3 NOP instructions) (Ofria, 2003, p. 14).

2.3 Amoeba

This work aimed to build a system that generates a self-replicating ancestor from a randomly generated code sequence, which is the parallel for prebiotic soup (Wikipedia contributors, 2021a). The first version of Amoeba was using a reduced set of instructions with only 16 machine operations compared to Avida's 24 and Tierra's 32. Amoeba-I is not Turing-complete. The second version uses an instruction set of size 32. Moreover, in this version, two stacks for each virtual CPU were added. The memory is two-dimensional similar to Avida's memory model (Pargellis, 2001, pp. 1-2), though in the same meaning as in Avida – organisms are one-dimensional.

Amoeba defines the "codon" as a pattern that encodes specific machine instructions. Therefore, each organism consists of codon-operation pairs. Amoeba uses 64 of such codons in order to encode 32 machine operations randomly. This allows addressing any operation of the creature in a new way, similar to the template matching. The CPUs are dynamically assigned to a particular lattice site. When the child is created using the MALL instruction, the virtual CPU is created on a lattice part. The codon-operation pairs are copied to this virtual CPU's operation stack (Pargellis, 2001, p. 4).

The first version of Amoeba was producing only inefficient organisms that could reproduce only once before dying. Such organisms are not the aim of this research. They were called protobiotic cells. However, the diversity of the organisms that emerged from the second version is more comprehensive. A probiotic cell evolved into a self-replicating biotic organism in 3472 generations. This result shows the possibility of simulating artificial life without a predefined ancestor (Pargellis, 2001, p. 8).

2.4 Fungera

The main goal of Fungera was to increase the complexity and diversity of organisms. Tierra produces a wide diversity of organisms at the beginning of the simulation. However, the diversity of the creatures reaches the plateau. Several reasons were examined as suspects for causing such behavior. Several approaches were used to solve the issues. However, there were no significant differences in evolutionary processes. The problem seems to be in the one-dimensional nature of the artificial world. Other reason – it's inherent unlocality: the effort to access the memory is not growing with the distance. Avida and Amoeba tried to solve this problem by implementing pseudo-two-dimensional models of RAM. However, these models use multiple one-dimensional memory spaces where instructions are isolated along the lines. In order to proceed with research in this direction, Mykhailo Poliakov under the guidance of the Oleg Farenyuk developed Fungera — a truly two-dimensional artificial life simulation (Poliakov, 2020, p. 10).

2.4.1 Befunge & Fungera instruction set

In order to create a truly two-dimensional organism, a two-dimensional instruction set is required. Most two-dimensional programming languages are considered esoteric because bare-metal machines are designed with one-dimensional RAM and CPU (registers, stack, etc.). In order to create the instruction set, the modification of Befunge Programming Language was used — Befunge-98. In this language, the IP (Instruction pointer) can move in four directions, and four corresponding operators change the direction of the IP. There is also the instruction that changes any character in the grid to another character. This instruction makes Befunge even more beneficial to use since it provides the base for self-replicating code. This ability is already very similar to the Tierran alternative. The Fungera instruction set is created by combining the two-dimensional nature of the Befunge-98 and Tierran instructions designed with open-ended evolution in mind (Poliakov, 2020, p. 11).

Code	Sym	Ops	Description	Type
[0, 0]	.	0	Template constructor	Template
[0, 1]	:	0	Template constructor	Template
[1, 0]	a	0	Register modifier	Register
[1, 1]	b	0	Register modifier	Register
[1, 2]	c	0	Register modifier	Register
[1, 3]	d	0	Register modifier	Register
[2, 0]	^	0	Direction modifier (up)	Direction
[2, 1]	v	0	Direction modifier (down)	Direction
[2, 2]	>	0	Direction modifier (right)	Direction
[2, 3]	<	0	Direction modifier (left)	Direction
[3, 0]	x	0	Operation modifier	Operation
[3, 1]	y	0	Operation modifier	Operation
[4, 0]	&	2+	Find template, put its address in register	Matching
[5, 0]	?	4	If not zero	Conditional
[6, 0]	0	1	Put [0, 0] vector into the register	Arithmetic
[6, 1]	1	1	Put [1, 1] vector into the register	Arithmetic
[6, 2]	-	2	Decrement value in register	Arithmetic
[6, 3]	+	2	Increment value in register	Arithmetic
[6, 4]	~	3	Subtract registers and store result in register	Arithmetic
[6, 5]	*	3	Add registers and store result in register	Arithmetic
[7, 0]	W	2	Write instruction from register to address	Replication
[7, 1]	L	2	Load instruction from address to register	Replication
[7, 2]	@	2	Allocate child memory of size	Replication
[7, 3]	$	0	Split child organism	Replication
[8, 0]	S	1	Push value from register into the stack	Stack
[8, 1]	P	1	Pop value of register into the stack	Stack

TABLE 2.1: Initial Fungera instruction set, (Poliakov, 2020, p. 12)

Fungera's instruction set introduces one new concept that is not present in Tierran or Befunge -— modifiers. For example, Tierran instruction SUB_AC (subtract register RC from register RA and put the result into RA) consists of Fungera's instruction ~ (subtract) followed by the three register modifiers: `~aca`. Modifiers by themselves are machine instructions as well. However, they are NOP instructions. Implementation of the instruction set in such a way is more straightforward because

there is no need to define every operation that could be performed in the instruction set since they can be made by combining simpler instructions with modifiers. Moreover, it makes the machine code more readable and familiar to the users of modern PLs. These aspects make the reversed set of the instructions equivalent to the original, e.g., +acb is the same as bca+ if the first case IP moves from the left to right and in the second case vice versa (Poliakov, 2020, p. 12).

2.4.2 Organism structure

Organisms are Tierra-like too except for being two-dimensional. Therefore, we have a new element in the CPU — ”delta”, which is the direction in which the IP is incremented. IP can be moved up, down, left, and right through memory. Another difference is that all registers are general-purpose registers. There are also concepts of the main memory and the child memory blocks. First is the address of the organism, the second – address of the memory where the organism is reproducing. Since the organisms are two-dimensional creatures, all elements of CPU are vectors of size 2 (Poliakov, 2020, p. 11).

CPU element	Description
RA	General-purpose register A
RB	General-purpose register B
RC	General-purpose register C
RD	General-purpose register D
IP	Instruction pointer
Delta	IP direction
Stack	Stack of configurable size (default 8)
Main memory block	Allocated memory block for organism itself
Child memory block	Optional memory block for child allocation

TABLE 2.2: CPU structure in Fungera (Poliakov, 2020, p. 11)

2.4.3 Memory

The memory in the Fungera simulator is truly a two-dimensional RAM grid with a configurable size. Each sell can hold one instruction, which makes it again very similar to Tierra’s RAM. All CPUs share it. The address of each cell is a two-element vector. The grid is not circular in contrast to Avida. Tierra’s, Avida’s, Amoeba’s, and Fungera’s memory models are visualized in the Fig. 2.3. Another difference from the Tierra is that organism has to write permissions only when it has allocated memory for the child. However, the write permission is not tied to the child’s memory block or its own. The organism has write privileges in each cell of the RAM. Mutations occur only by changing random instruction to random one from the instruction set with a configurable rate (Poliakov, 2020, p. 13).

Each organism in the simulation is added to the Queue. There is only one queue as opposed to Tierras’s ”reaper” and ”slicer queue”. Mortality is achieved in three ways:

- organisms dies if it made more than a configured amount of errors;
- the Queue removes configured percentage of creatures with most errors when the memory is filled to a configured extent;

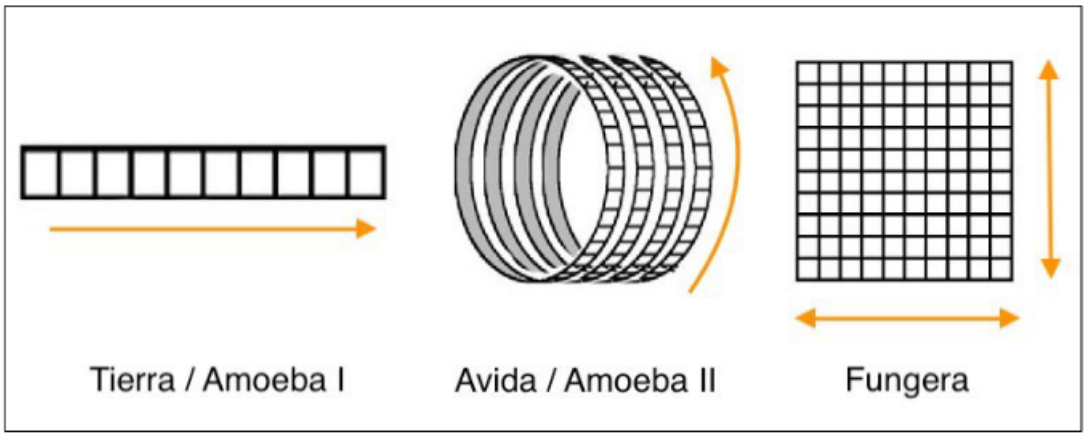

FIGURE 2.3: Tierra, Avida, Amoeba vs. Fungera memory design (Poliakov, 2020, p. 13)

- the infertile organisms are removed after a configured amount of cycles.

This queue also is responsible for multitasking in order to simulate parallelism. However, as for now, all CPUs receive the same CPU time slice despite their size: enough to execute just one machine instruction (Poliakov, 2020, p. 13).

2.4.4 Results

More than 170 size classes emerged in the large-scale simulation with memory size [5000, 5000]. However, only a few of them are capable of further reproduction and evolution. Different behavioral patterns in different size classes occur as well as it was in the Tierra simulator. For example, the offsprings emerged, which can reproduce only to the right and downwards. At the same time, this organism's reproduction cycle was shorter (Poliakov, 2020, p. 17).

Results in micro- and macro-evolution prove that Fungera is a step in the new direction which may lead to valuable results and conclusions. However, it is stated in the work that the implementation of the simulation needs improvement: Fungera should be reimplemented using compiled, efficiency-oriented language like C++; simulator itself and GUI should be separate objects, which will allow running the simulation faster without the visualization; there should be a work done to reach some true parallelism (Poliakov, 2020, p. 25).

Chapter 3

Implementation

The work of Mykhailo Poliakov proves that the Fungera instruction set and a virtual computer model are viable to research and can lead to meaningful evolutionary results. In the described simulation runs, a significant number of size classes emerged. Some of them are even capable of reproduction. Several successors showed different reproduction strategies; some, such as microvesicles, have an interesting behavioral difference, which is close to parasitism. All these conclusions are based on the results of the original paper, which describes the results of simulation reaching cycle 700000 (Poliakov, 2020, p. 25). In comparison, some runs of Tierra reach 2.56 billion executed instructions (Ray, 1991, p.16). It is evident that for further analysis, there is a need for a simulator that runs significantly faster to diversify the number of runs with different configurations and various initial conditions. Also, research of the Tierra-like artificial life requires ability to easily change the instruction sets.

My work provides a more efficient and flexible implementation of the Fungera simulation in C++.

In order for the implementation of the Fungera simulation in C++ to be efficient, the existing issues in the original Python implementation need to be addressed:

- TUI implementation is embedded into the Fungera abstractions such as RAM, organism, and the simulation class itself.
- Simulation can not run without TUI.

In order to eliminate such flaws from the current implementation (Yavorskyi, 2021), GUI was designed to be a separate essence from the very beginning and use simulation as the source of data. This decision results in the minimization of the simulation deceleration caused by the GUI support. Simulation only notifies the GUI that the data has changed, which allows executing the simulation without the GUI with little to no drawbacks brought by the GUI feature. This part is crucial because the graphical interfaces are not needed in the long run when the data for further analysis is collected. Basic logging is enough to know the status of the execution. Interfaces are needed mainly for debugging purposes: to check if the simulation runs correctly, examine the behavior of a specific creature, and visualize the simulation when modifying the instruction set.

The GUI was built using *Qt* — a cross-platform framework implemented in C++ for application development, building robust and extensible GUI for C++ applications. The framework is designed using model/view architecture to manage the relationship between data and visual representation. This architecture is a modification of the MVC design pattern, where view and controller objects are combined. Such a solution allows separating the data representation from the way it is stored. Therefore, the connection between the simulation and GUI is minimized to the notifications that contain updated information.

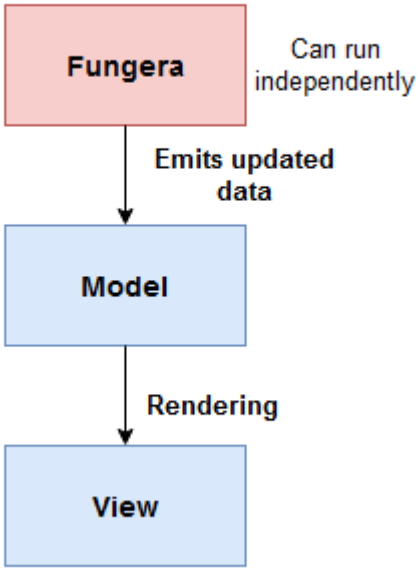

FIGURE 3.1: Connection between simulation and GUI

The GUI can be divided into three sections (Figure 3.2). The first one is showing the basic status of the simulation:

- current cycle
- number of alive organisms
- number of times queue was killing the organisms

This section also provides information about the selected organism, such as:

- values in the stack and registers
- the position of the IP
- IP direction
- number of errors made by this organism
- etc

The second part is the visualization of RAM. This part of the GUI fixes some issues present in the reference implementation TUI, such as unstable IP of the selected organism visualization. That allows more convenient step-by-step debugging. New features were implemented, such as highlighting organisms' borders and visualizing memory cell numbers.

The third section is control elements, which allow manipulating the simulation and implement the troubleshooting feature. The simulation can be paused with the Pause button. The Cycle button allows executing exactly one simulation cycle. The Next and Prev buttons change the selected organism, and there is also an option to choose the exact organism id with a drop-down menu. The Cycle selected button executes exactly one cycle for the selected organism without executing the code of any other organisms. To advance the simulation forward for the desired number of

simulation cycles, use the line edit and the Advance button. A user can choose the instruction. When the selected organism executes this instruction, the simulation will be paused. Last but not least, the simulation can be paused when any of the organisms' IP reaches configured memory cell.

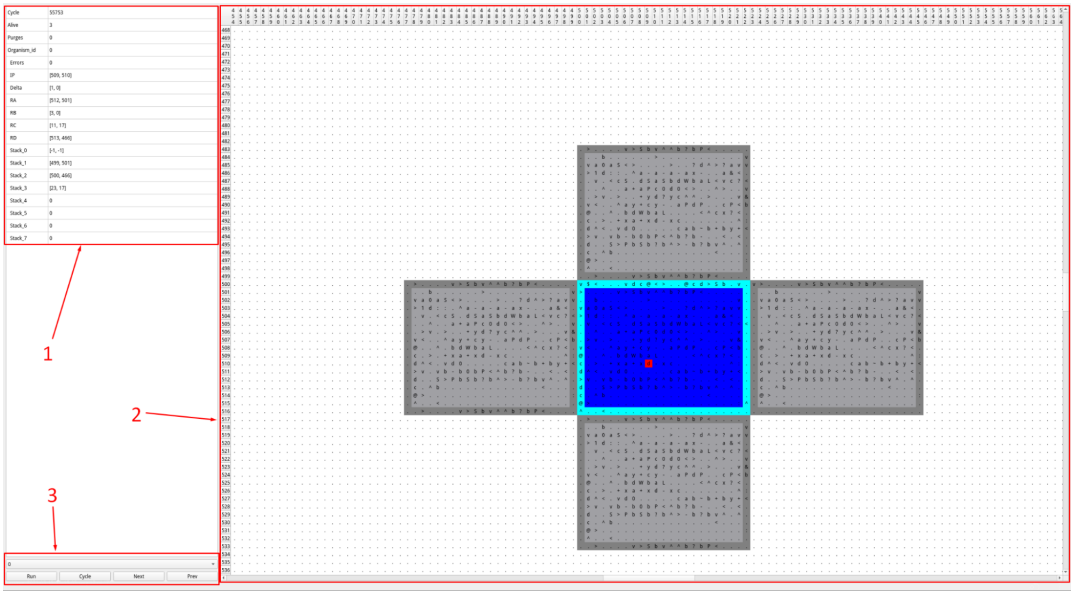

FIGURE 3.2: GUI for Fungera simulator (1 - Simulation status and selected organism's properties; 2 - RAM visualization with the selected organism, its borders, and IP highlighted; 3 - Control elements)

In order to save the information about the state of the simulation for the restarts of the simulation or further analysis and obtain information about existing genotypes and other aspects that can be valuable, snapshots are saved at a configurable rate. The implemented solution is using the *Boost Serialization library*. This library allows serializing C++ and saving them on disk. The object which represents a serialized C++ object is referred to as "archive". Among the supported archives formats, this implementation uses two:

- portable text archives - used to save snapshots
- XML archives that are easier to use for further analysis

XML archives are portable as well. However, they are not suitable to save many snapshots due to the size. The application has the functionality to convert snapshots in text archives into XML archives.

The description of the objects that form the simulator should be started with Fungera class, the central object, which manages all parts. It is responsible for the execution, creates, and owns Memory and Queue instances. Fungera object is also the main interface of communication with the whole simulation. Memory object represents RAM. It holds memory cells, each cell containing machine instruction and a flag that indicates if some organism occupies the cell. Memory also provides an interface for allocation/deallocation of the memory block, searching for a free block of the requested size. Queue object is the owner of organisms. It is responsible for providing real CPU time for all creatures. It has the interface to eliminate the configured percentage of organisms when RAM reaches a set ratio of occupied to free cells. Organism object holds Fungera's organism structure along with the implementation of the instruction set. The relation between objects is described in the figure:

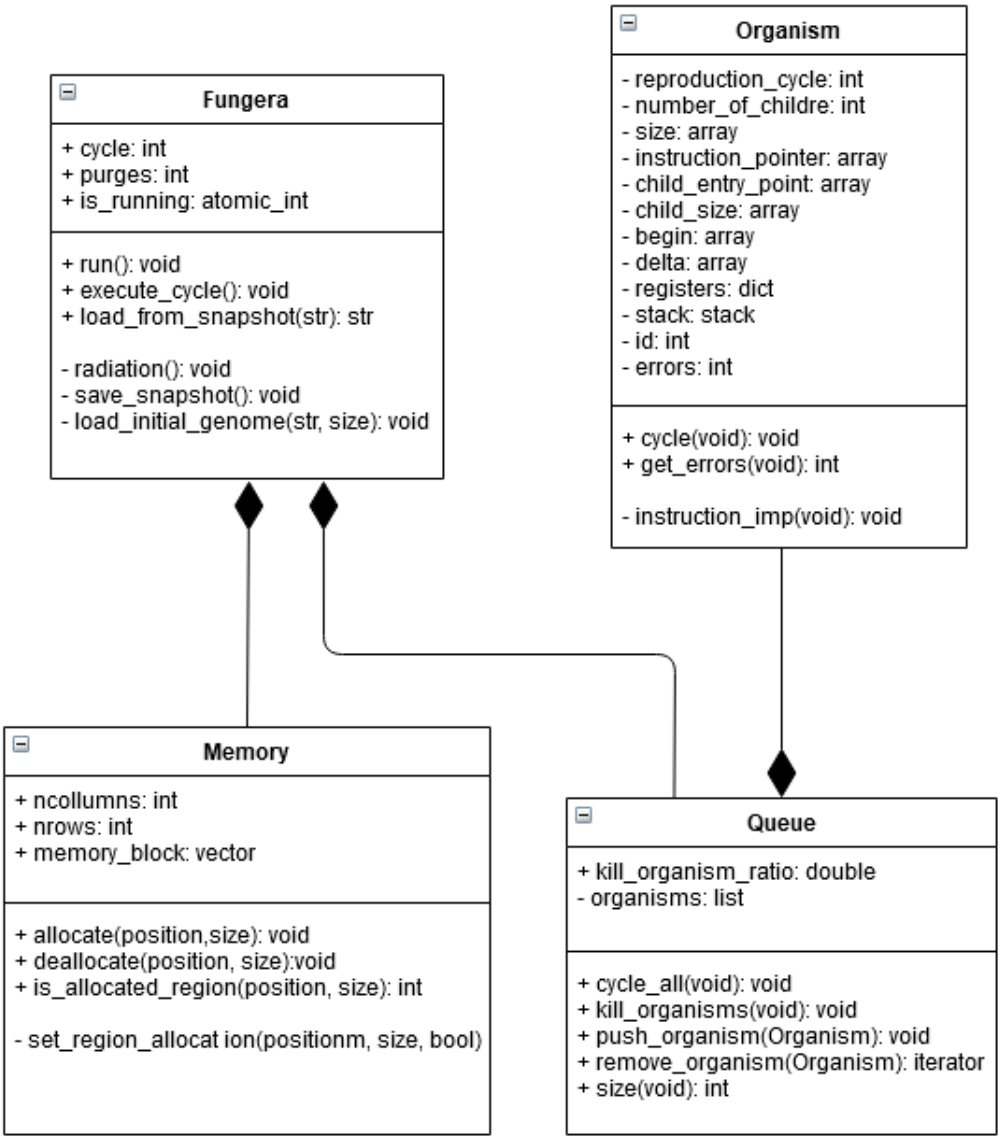

FIGURE 3.3: Basic architecture of Fungera simulator

The mutation mechanism was extended in the current implementation of the Fungera. In order to make changes in organisms more flexible, the ideas from Tierra's mutations were adopted into Fungera. The instruction set is still deterministic. However, the number of mutations and the probability of mutation can be configured. Moreover, it is possible to configure mutations during the copying process. In order to run Fungera in its original configuration, the number of mutations from radiation and its probability of it should be set to 1, and the probability of the mutation during copy instruction should be set to 0.

Regarding the current implementation and reference Python implementation performance, the time was measured of each simulator, reaching 1000, 10000, and 100000 cycles. All the executions were done on the Intel i7-7500U and 16 GB of DDR4 RAM with a virtual memory size of 1000 by 1000 cells. Since the TUI code of the Python implementation is embedded into the simulator and affects the performance, it is fair to compare all of the test runs:

- Reference implementation with TUI: 0.70 seconds, 6.93 seconds, 357.50 seconds
- Reference implementation with minimized TUI: 0.032 seconds, 0.235 seconds, 13.447 seconds
- Current C++ implementation with GUI: 0.179 seconds, 1.597 seconds, 15.141 seconds
- Current C++ implementation without GUI: 0.0017 seconds, 0.0043 seconds, 0.0417 seconds

The test runs showed that the current implementation with GUI debugger enabled has almost the same performance as the reference implementation with a minimized GUI. The simulator's performance while using GUI is not crucial since its purpose is the ability to troubleshoot and visualize the system. The consistency of the GUI with the state of the simulation at any point in time is the real important aspect here. At each step of the simulation, all the organism status and memory visualization should be updated. This aspect, along with new features introduced by GUI, makes the runs with interface enabled slower. Therefore, the regular simulation runs should be executed with GUI disabled.

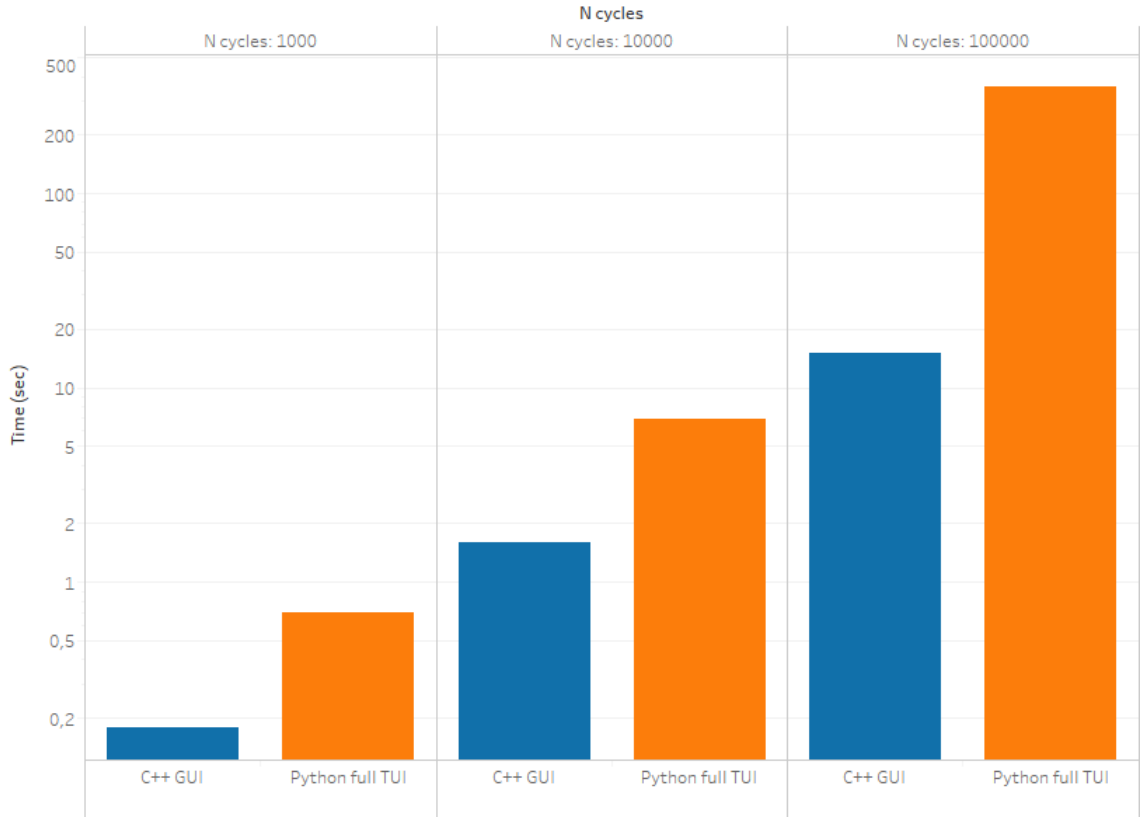

FIGURE 3.4: Performance comparison of the reference Python implementation and current C++ implementation with visualization enabled

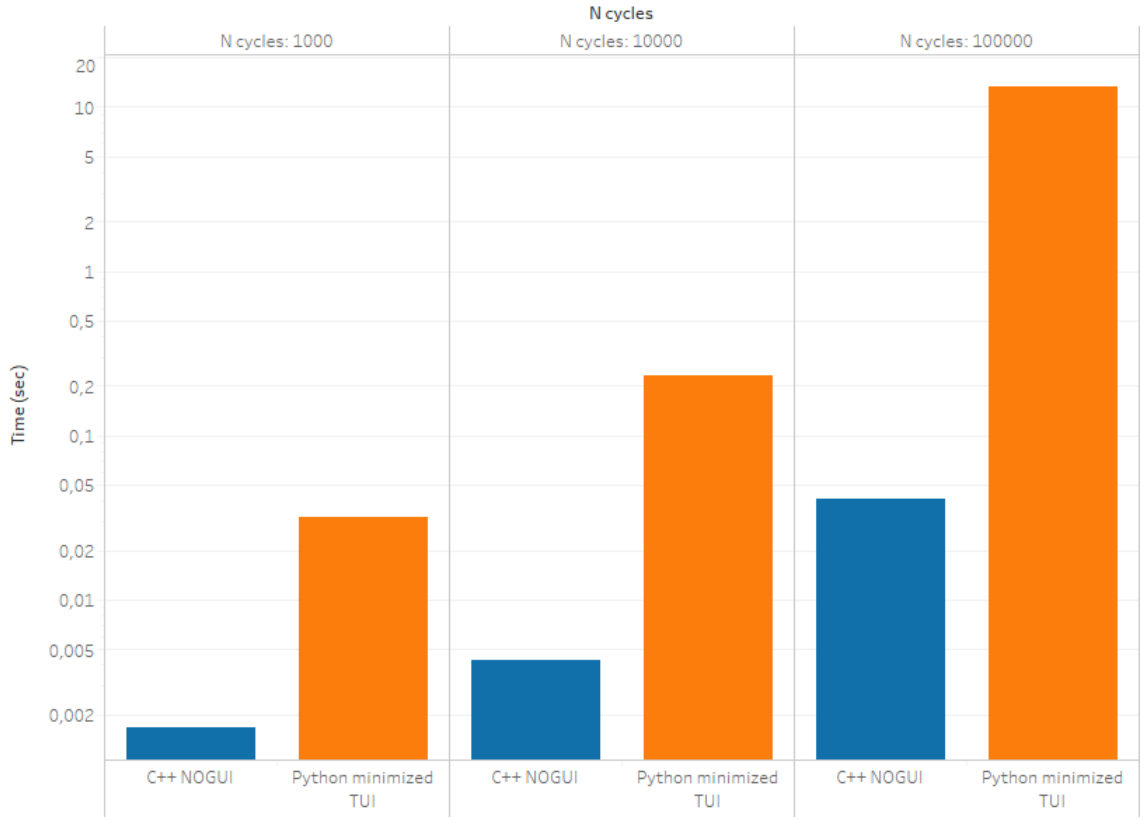

FIGURE 3.5: Performance comparison of the reference Python implementation and current C++ implementation with visualization disabled(minimized)

Chapter 4

Conclusion & Future works

4.1 Conclusion

The result of this work is the implementation of the truly two-dimensional artificial life simulator Fungera developed in C++. This implementation opens possibilities for further analysis of this artificial world by accumulating data from various executions with different configurations and starting conditions. Implemented GUI brings the possibility to examine newly emerged genotypes and their behavioral patterns step-by-step. The instruction set of the simulation is easy to modify and integrate new instructions as long as they work with the organism and memory structures defined by the Fungera. The GUI plays the role of debugger for such modifications as new instructions can be tested in order to work as expected by the direct visualization of memory status and the status of the organisms' key parameters.

4.2 Future woks

While this simulator is faster than the initial Python implementation, true parallelism, which could significantly improve the performance, was outside of the scope of this work. All instructions that do not interact with the RAM or queue can be executed in parallel in the current design. Before extending the solution with true parallelism, instructions that require synchronization when accessing shared resources such as memory and queue of organisms should be modified to prevent data races. Nevertheless, the current implementation can be extended to implement this feature in the future.

Another feature that might be useful is extending the GUI with more controls on the simulation. Suppose this application will be used to modify the instruction set. In that case, the possibility of executing the code of only a selected organism might make it easier to troubleshoot newly added instructions. The opportunity to run the simulation up to the requested cycle and to make the force snapshot on the current cycle from the visual interface are features that will help as well.

Bibliography

Adami, C. (1994). In: URL: https://arxiv.org/pdf/adap-org/9405003.pdf.

Langton, Christopher G. (July 1995). *Artificial Life: An Overview*. The MIT Press. ISBN: 9780262277921. DOI: 10.7551/mitpress/1427.001.0001. URL: https://doi.org/10.7551/mitpress/1427.001.0001.

Ofria, C. (2003). In: URL: http://www.cs.cas.cz/~petra/EA/AvidaIntro-ALife.pdf.

Pargellis, A. N. (Jan. 2001). "Digital Life Behavior in the Amoeba World". In: *Artificial Life* 7.1, pp. 63–75. ISSN: 1064-5462. DOI: 10.1162/106454601300328025. eprint: https://direct.mit.edu/artl/article-pdf/7/1/63/1661839/106454601300328025.pdf. URL: https://doi.org/10.1162/106454601300328025.

Poliakov, M. (2020). "Evolution of digital organisms in truly two-dimensional memory space". In:

Ray, T. (1991). "Evolution , Ecology and Optimization of Digital Organisms". In: URL: https://www.cc.gatech.edu/~turk/bio_sim/articles/tierra_thomas_ray.pdf.

Wikipedia contributors (2018). *Phyletic gradualism — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-May-2021]. URL: https://en.wikipedia.org/w/index.php?title=Phyletic_gradualism&oldid=845764212.

— (2021a). *Primordial soup — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-May-2021]. URL: https://en.wikipedia.org/w/index.php?title=Primordial_soup&oldid=1020031403.

— (2021b). *Punctuated equilibrium — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-May-2021]. URL: https://en.wikipedia.org/w/index.php?title=Punctuated_equilibrium&oldid=1015575621.

Yavorskyi, Hermann (2021). *Fungera efficient implementation in C++*. URL: https://github.com/wardady/fungera_hp.