Gloriya Gostyaeva

Evolving virtual ecology

Dissertation/Internship Report
Master in Multimedia Design
advised by Penousal Machado, Tiago Martins
and presented to the Departament of Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra

July 2018

· U C ·

Universidade de Coimbra

EVOLVING VIRTUAL ECOLOGY

BY GLORIYA GOSTYAEVA

THESIS

Submitted in partial fulfilment of the requirements for the degree of Master of Design and Multimedia in the Department of Informatics Engineering of the University of Coimbra, 2018
Coimbra, Portugal

Adviser: Penousal Machado
Co-Adviser: Tiago Martins

ABSTRACT

The field exploring artificial life has existed since the early ages of computer science. Because of its development, it has been possible to create numerous virtual models that have allowed the study of the behaviour of living organisms and their interactions within artificially created ecosystems. Whilst the methods employed in this field have been mostly explored by various researchers in their projects, they had not been broadly applied in the entertainment and art fields.

With this thesis, I intend to create an entertainment software which contains agents with artificial life. Beyond this, the effectiveness of machine learning as one of the key elements in an entertainment directed program is to be investigated. A question that will be explored in this thesis is how machine learning can enable the artificial creatures to respond to audio commands in addition to audio-based interaction between creatures and their environment.

KEY WORDS

Artificial life, video games, evolutionary computation, genetic algorithms, machine learning, sound recognition

SUMÁRIO

O domínio que explora a vida artificial existe desde do início da ciência da computação. O desenvolvimento deste domínio levou criaçãode vários modelos virtuais que possibilitam o estudo do comportamento de organismos vivos e suas interações dentro de ecossistemas criados artificialmente. Estes metodos tem sido explorados principalmente por investigadores do domínio, não tendo sido explorados nos campos de entretenimento e arte.

Com esta tese, pretendo criar um sistema computacional de entretenimento que contem agentes com vida artificial. Além disso, a eficácia do aprendizagem de máquina como um dos elementos-chave de um programa direcionado ao entretenimento deve ser investigada. Uma questão que será explorada nesta tese é como aprendizagem de máquina pode permitir que as criaturas artificiais respondam aos comandos de áudio, além da interação baseada em áudio entre as criaturas e seu ambiente.

PALAVRAS-CHAVE

Vida artificial, video-jogos, computação evolutiva, algoritmos geneticos, aprendizagem de máquina, reconhecimento de som

ABBREVIATIONS AND ACRONYMS

2D – Two-dimensional/ second dimension

3D - Three-dimensional/ third dimension

AI - Artificial intelligence

ANN – Artificial neural network

A-Life - Artificial life

CPU – Central processing unit

DNN – Deep neural network

DQN – Deep Q network

EA – Evolutionary algorithms

GA – Genetic algorithm

GPU – Graphics processing unit

LSTM – Long short-term memory

MSE – Mean squared error

Q – Queen

UC - University of Coimbra

UI – User interface

UML – Unified Modelling Language

W – Warrior

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

This thesis represents the work undertaken to complete the dissertation element of a master's course in Design and Multimedia at the University of Coimbra. The introduction section provides a summary of the research context and rationale, the key aims and objectives of the project as well as an outline of the thesis structure.

1.1 Motivation

The A-life field has been present since the early days of computing. Some of algorithms related to the field had existed even before it was possible to reproduce them programmatically (e.g. Conway's Game of Life). Living organisms are extremely complex and by learning their patterns, it is possible to translate them into algorithms.

The motivation of this work was primarily enforced by observing various examples of artificial life and artificial intelligence at several exhibitions and the desire of discovering possibilities that are offered by this often under researched field within the entertainment industry.

The book of Nicola Tesla, "My Inventions", served me as a key element towards understanding how learning the rules and patterns of life is an important task that can lead to many significant discoveries. According to Tesla, all humans, animals and creatures may arguably be nothing more than automata acting in accord with the data sent to their brains by various sensors. It is important to study these patterns in order to create a system that can be truly efficient and stable.

An interesting topic, related to simulated environments is the use of audio sensors in virtual creatures. It is not a broadly explored topic and creating such an environment could serve as a possible contribution to the field of AI. Creation of a piece of entertainment software with the a-life elements will be a step into this not broadly explored area, especially in the field of entertainment, which I am willing to undertake.

1.2 Area

A-Life is a field wherein researchers are studying systems related to natural life, its patterns and evolution. The study is done through creation of computational models, robots or bio-chemistry. In this thesis, the main project within the A-life field was the creation of artificial agents functioning within a virtual environment.

Machine learning is a part of the field of AI. It studies algorithms that allow various agents to learn, similarly to actual living creatures. Several algorithms and techniques from this field were explored in this project, primarily artificial neural networks (ANN), evolutionary algorithms (EA), and Q-learning.

Additional fields which were relevant to this project to a small degree are game design and user interface (UI).

1.3 Objectives

In order to consider this work successful it was necessary to accomplish the following objectives:
- Study how A-life agents can be created using a computational approach.
- Determine and study design problems within similar projects and how they can be avoided.
- Develop a virtual toy involving A-life agents as a part of its key elements.

- Construct a system within the toy environment allowing the user to interact with and teach A-life agents through their audio sensors.
- Develop a communication system between the A-Life agents of the toy and the simulated environment.

1.4 Methodology

In the first phase of the project, the study of existing systems was accomplished. During this phase, it was equally important to understand how neural networks and the evolution factor can be applied to A-life agents using a programmatic approach.

The second phase consisted of the planning of the software that is to be developed. The decisions had to be made about the style, tools and the user interaction. In addition to that, decisions about the architecture of neural networks and the evolutionary systems had to be made.

This phase also involved the development of an application and its analysis. During this phase, the conclusions about the project were made.

1.5 Expected contribution

It was expected, through a critical analysis of the existing cases in the relevant areas to find a relation between them and to bring up new research questions and answers.

From the practical side of the work, it was expected to utilise A-life agents in a unique manner within the entertainment application. It was also expected to find a new way of interaction with these agents using speech and as a result integrating it into their AI. This could not only contribute to the field of UI but also AI.

A way of evolving artificial agents within a toy environment is expected to be a possible contribution to an entertainment field such as video games, as well as the field of AI.

1.6 Document structure

The present document is structured as follows:

Chapter 2, LITERATURE REVIEW, contains a brief introduction into the history of A-life and explains its nature. It also describes some papers that particularly inspired this work and contributed to its development. By no means is it a review of the most recent and state of the art contributions to the field, neither are these projects are representative of the field. The mentioned works are merely considered relevant to this thesis.

Chapter 3, OBJECTIVES AND METHODS, defines the objectives of the thesis and the methodology employed in order to achieve them.

Chapter 4, PRELIMINARY WORK, describes preliminary work that had been done in the 1st semester and in summer in preparation for the thesis. The analysis of early experiments is also presented here.

Chapter 5, PRACTICAL WORK, describes the practical work that had been done in the 2nd semester. It describes the continuation of preliminary experiments and implementation of the final application.

Chapter 6, CONCLUSIONS, contains the conclusions made about the work.

2 LITERATURE REVIEW

“In the course of time it became perfectly evident to me that I was merely an automaton endowed with power of movement, responding to the stimuli of the sense organs and thinking and acting accordingly.“

Nicola Tesla - “My inventions”

2.1 Chapter structure

Before going into the chapter, I will discuss its sections and reasons behind their inclusion.

Section 2.1, Artificial life and humans, serves as a brief introduction to the topic, describing some historical details that served as an inspiration for this project. The section is limited to technology that is relevant to this work and served as an inspiration for it.

Section 2.2, Cellular Automata, primarily talks about John Conway’s Game of Life which is one of the life simulations from the early days of computing. The reason for its inclusion is primarily its historical importance in serving as one of the first steps towards the creation of life simulation. Even though it is not the first or the last cellular automaton that was invented, it was an important inspiration for this project.

Section 2.3, Simulations of Creatures, discusses the attempts that had been accomplished in order to create the realistic simulations and models of creatures, their evolution, life-cycle and other key elements of their existence. Some of them are not state of the art technologies of the field, but they are relevant for this particular project.

Section 2.4, Living agents in entertainment software, discusses the attempts that had been made within the field of entertainment software towards including the A-Life agents as their key element. The included examples were studied with the goal of finding the unexplored path which could possibly be taken in order to discover new opportunities in the field, as well as space for contribution.

Section 2.5, Machine learning technology, briefly talks about the definition and history of Artificial Neural Networks and other machine learning technology relevant to this project. This section exists primarily to introduce the technologies that were used in this project.

2.2 Artificial life and humans

2.2.1 Historical overview

The question of artificial life has always been one of the most intriguing topics for humanity. During the excavations of ancient civilisations, a lot of statues and paintings portraying humans, animals and body parts are discovered. From the ancient Greek myths and poetry, we learned about the fantasies of people trying to bring the statues to life. Ovid's Metamorphoses still remains as one of the most important sources of classical mythology. One of the notable poems inside the book is about Pygmalion and the statue where the sculpture had been brought to life by the Greek goddess Venus. This romantic view on crafting a living organism is one of many examples of people dreaming of engineering their own living creature.

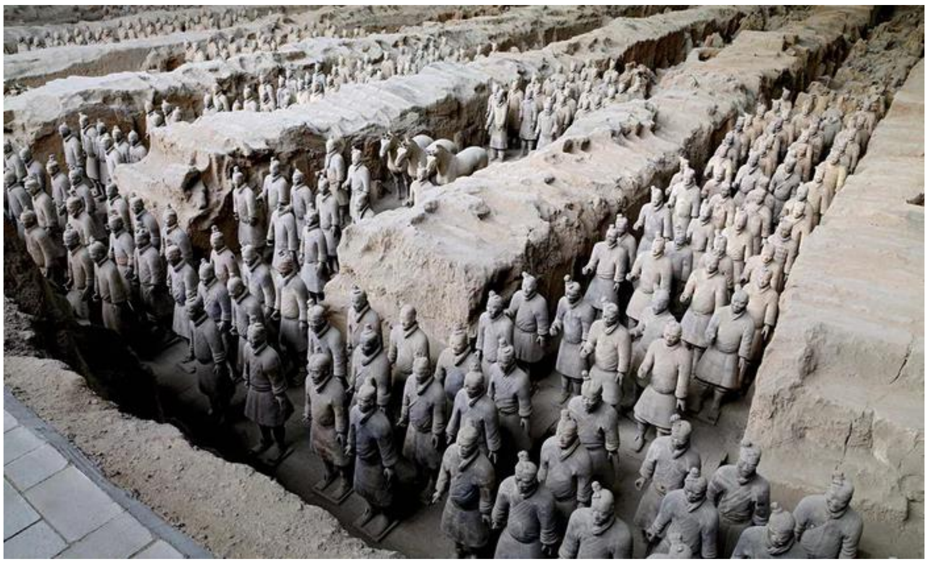


Figure 2: Photograph of a part of the Terracotta Army in China.

The Terracotta Army (Fig.1) in China was constructed with the belief that it would serve the emperor in the afterlife, it is a notable example of people creating art which they believe will come to life at some point (even if it will only animate in the afterlife).

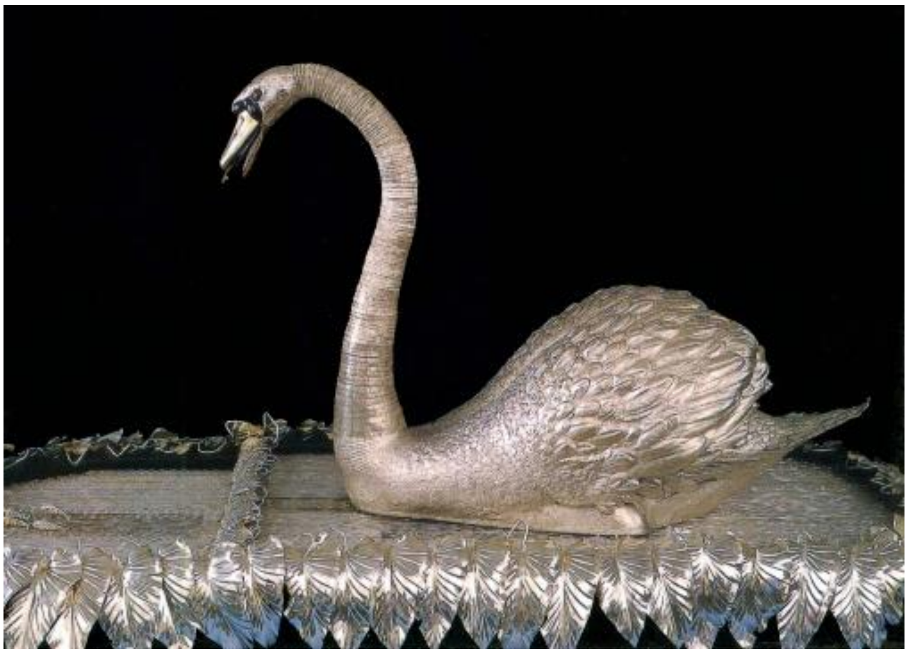


Figure 1: Silver Swan Automaton on the lake made of silver and glass.

Later on, in the era of the machines, we have examples of automata. An automaton is usually a figure made out of metal (such as brass) or wood which is then animated using mostly mechanics, clockwork mechanisms and, in some cases, a basic electricity without any computer logic. One of most notable examples of automata is the Silver Swan (Figure 2). It consists of a mechanical construction of a swan positioned atop a crystal lake with mechanical fish in it.

When the swan automaton is wound up, it comes to action. Glass rods rotate to simulate water and the mechanical fish are moving within it. The swan made of silver turns its neck and picks up one of the fish with its beak. It is a unique and beautiful mechanism made in an attempt to recreate the grace of a living animal.

Some more modern classic examples of this fantasy are the Frankenstein's monster and Maria (Figure 3) from the 1927 silent film Metropolis among many other examples.

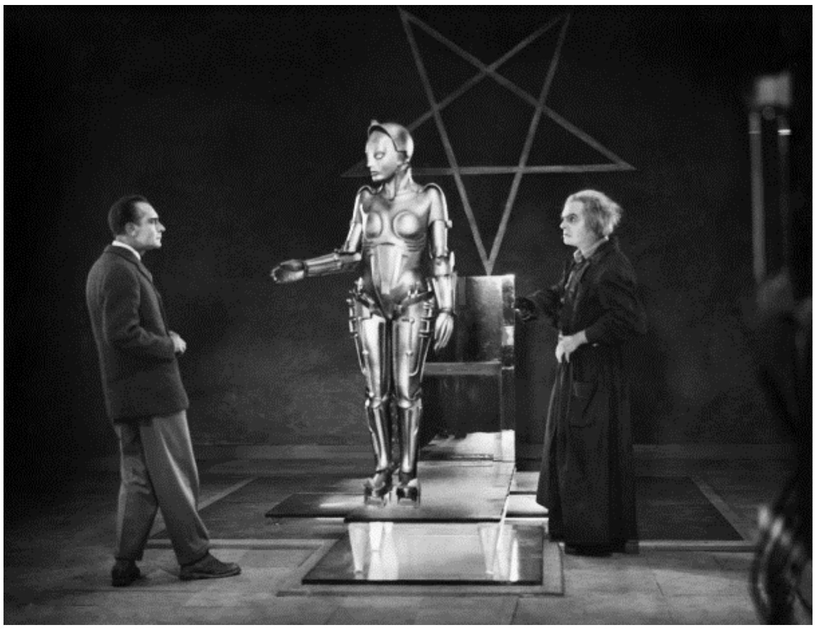


Figure 3: A scene from Metropolis showing Maria the automaton in the middle of the composition.

Why are humans interested in creating something that would come to life? Sometimes it would be a personal issue like loneliness or a desire to have a perfect servant, but the idea of creating an artificial human/creature is much more ancient and deep than that. Life is a mystery and the one who discovers its secret will be able to accomplish great things.

2.2.2 Nature in AI

In our current world, scientists, artists and engineers around the world are working within the field of AI. Many of the AI algorithms have their roots in natural phenomena. The fact that some of the patterns from nature can be translated into a programmatic algorithm had become known since the early days of computing. Some of the most notable algorithms based on nature are the artificial neural networks (ANN) and the evolutionary algorithms (EA).

ANN are the algorithms that simulate the network of neurons in the human brain (or a brain of any other creature). The function of ANN consists mostly of the responses of artificial neurons to outputs of different sensors. Each of the artificial neurons has connections with other neurons, it is a connectionist system. Each of the connections has a value associated with it (weight). Each action performed by a network is a result of calculation involving weights and inputs from sensors. Any change in weights results in different behaviour. Changes in the state of the world (reinforcement learning), or feedback from the teacher (supervised learning) can result in changes of weights, therefore altering the final behaviour. Functionality, history and types of ANN will be described in more detail in section 2.5 of this chapter.

2.2.3 Evolutionary algorithms

EA are based primarily on Darwin's theory of the evolution of species, described in his book "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life". According to his theory, the life on Earth exists the way we see it today due to natural selection with possibility of mutation.

Fitness of an individual is a factor determining its survival. Only the fittest can reproduce (and therefore their genotype stays in the universe), while unfit ones fail to do so and their genotypes get discarded. This process is called natural selection.

Mutation factor is something that allows evolution to progress. Sometimes one or more individuals among the offspring may be born with random alterations in their genotype which may cause them to be fitter than the other individuals, or, if they are less fortunate, lower degrees of fitness. If a mutated individual is fit, it will procreate and produce offspring which will inherit its characteristics. As the evolution progresses, it can create an individual with high enough fitness to solve a problem that is set before the population (be it a survival-related problem, or a mathematical problem).

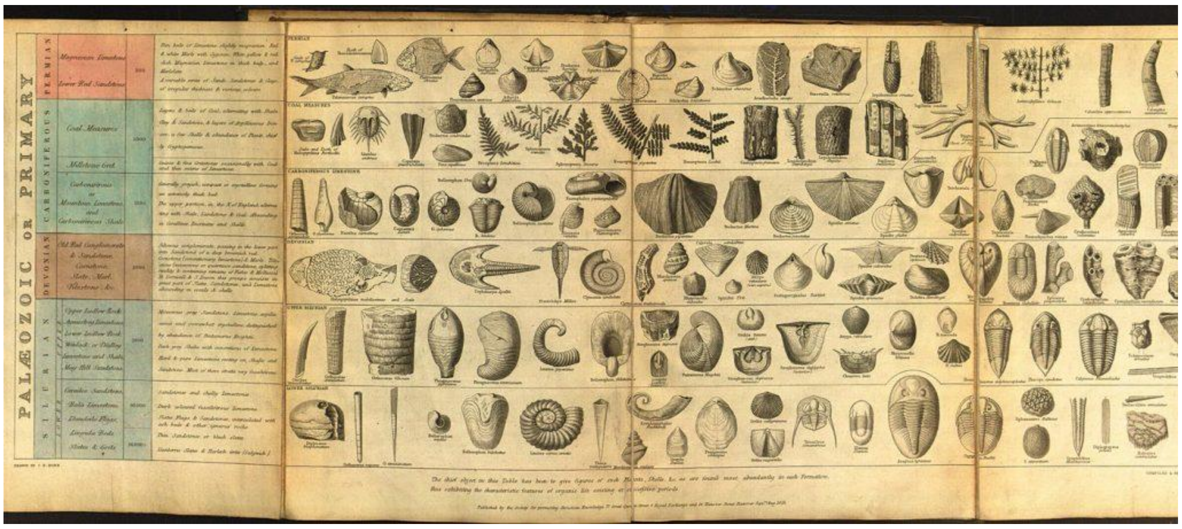


Figure 4: Tabular View of Characteristic British Fossils, Stratigraphically Arranged" (1853) from "Science Circa 1859: On the Eve of Darwin's 'Origin of Species.

Genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection and belongs to class of EA. GA are relying on bio-inspired operators such as mutation, crossover and natural selection.

Its creation began with Alan Turing in 1950 when he proposed a "learning machine" which would function in accord to the principles of evolution. Actual computer simulation of evolution began later in 1954 with the research of Nils Aall Barricelli, but his publication was not widely noticed. Later, with the works of Fraser and Burnell (1970), as well as Crosby (1973) and Bremermann (1960) computer simulation of evolution became more common. Other noteworthy early pioneers include Richard Friedberg, George Friedman and Michael Conrad.

Good examples of EA use are in space programs. When a rocket is being launched it is necessary to perform all calculations beforehand in order to maintain the program as cost efficient as possible and ensure safety. If a spacecraft needs to meet a space station on orbit, it is required to make a step further with these calculations and keep in account the position of the station relative to the position of where the spacecraft is being launched from, predicting the future positions of both objects as they are moving on their orbit. It needs to be launched at the right moment when the estimated fuel burn required to meet the target object is the smallest, apart from many other factors that need to be kept in mind. Distances, trajectories, angle of a rocket, weights, weather, day and year, etc. In order to perform a calculation using so many variables, it has proven useful to utilise GA.

Both EA and ANN are used in Boston Dynamics who are working on a robot with the goal to make it develop an ability to walk, jump, recover from a fall and do many other things related to maintaining equilibrium and developing a good walking technique. Because it would be a tedious task for any human to hard code such a complex and detailed behaviour, the calculations and adjustments in the robot's movements are done through use of ANN and EA.

Solutions to many complex mathematical problems such as of the travelling salesman or of the brachistochrone curve (Figure 5) were found using these algorithms. Systems like web search, image and speech recognition, text prediction also use these algorithms. They can be found in various computers (tablets, smartphones, laptops, PC's, etc.) throughout the world.

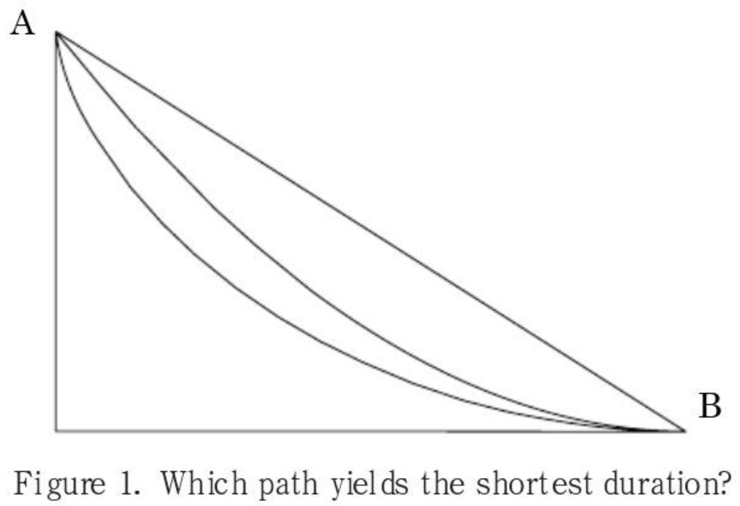


Figure 5: Brachistochrone problem depicted as a graph with two points.

2.3 Cellular Automata

2.3.1 Historical introduction

A cellular automaton is a mathematical model studied in computer science, theoretical biology and many other fields. The model consists of a grid within any finite number of dimensions of cells where each of the cells has one of the finite states (such as on and off). The state transitions are usually triggered by the status of the nearby cells or the previous status of the cell.

John von Neumann in 1940 has defined life as a creation which can reproduce itself and simulate the Turing machine. He was thinking about an engineering solution which would use electromagnetic components floating randomly in gas or liquid. But because of the state of technology of those days, it didn't prove to be realistic.

Then Stanislaw Ulam invented the cell automaton which was to simulate von Neumann's idea by applying the early computers to do so. In parallel, von Neumann attempted to construct Ulam's cellular automaton which would be alive. He succeeded but left it unfinished. Von Neumann’s cellular automaton was also very complicated. Later on, much simpler life constructions were provided by other researchers and published in papers and books. It turned out that even simpler life automatons were possible.

2.3.2. Conway’s Game of Life

John Conway had set a goal to define an interesting and unpredictable cell automaton. He wanted it to have many different configurations, some of which would die out fast, and some would last for a long time. Conway's Life game (1970) admitted a configuration which was "alive" in the sense of satisfying two of von Neumann's general axioms of life:

- Can reproduce itself
- Can simulate the Turing machine

Conway’s game of Life is operating in accord with four simple rules. The rules of Conway's automaton are the following:

1. Any live cell with fewer than two live neighbours dies as if caused by underpopulation.
2. Any live cell with two or three live neighbours’ lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

With these rules, it is possible to create several patterns which would result in interesting behaviour within the simulation. The simplest patterns were discovered before the use of computing. Among them are still and oscillating patterns that were discovered while tracking various small configurations using graph paper or physical game boards such as Go. During that research, a glider (Figure 7) and some other patterns were discovered. The glider is the simplest of the moving patterns.

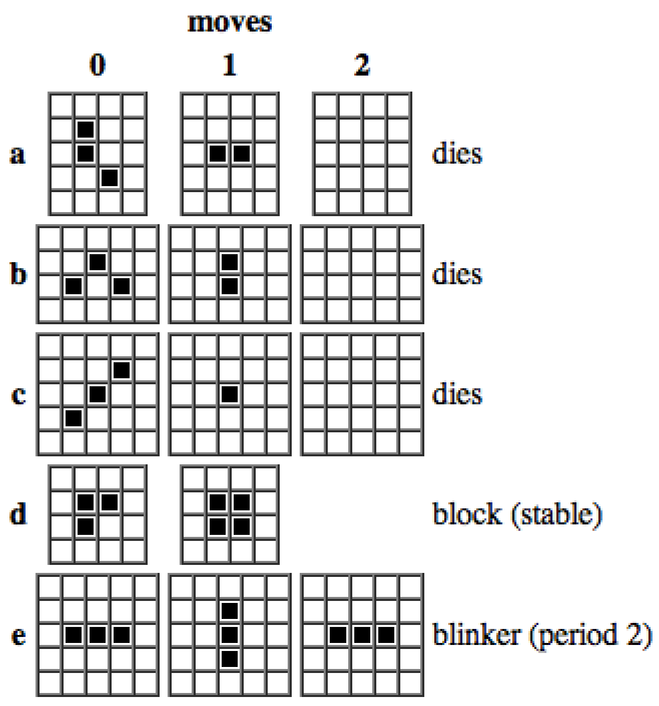
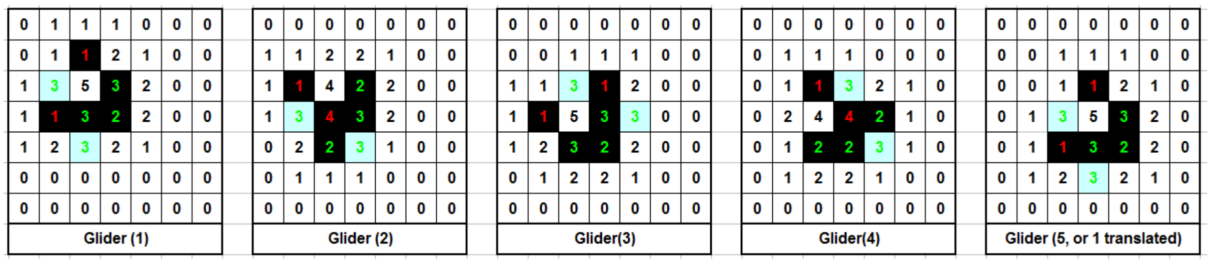


Figure 6: Some possible combinations and outcomes from Conway's game of Life.



0	1	1	1	0	0	0
0	1	1	2	1	0	0
1	3	5	3	2	0	0
1	1	3	2	2	0	0
1	2	3	2	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
Glider (1)						

0	0	0	0	0	0	0
1	1	2	2	1	0	0
1	1	4	2	2	0	0
1	3	4	3	2	0	0
0	2	2	3	1	0	0
0	1	1	1	0	0	0
0	0	0	0	0	0	0
Glider (2)						

0	0	0	0	0	0	0
0	0	1	1	1	0	0
1	1	3	1	2	0	0
1	1	5	3	3	0	0
1	2	3	2	2	0	0
0	1	2	2	1	0	0
0	0	0	0	0	0	0
Glider(3)						

0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	1	1	3	2	1	0
0	2	4	4	2	1	0
0	1	2	2	3	1	0
0	1	2	2	1	0	0
0	0	0	0	0	0	0
Glider(4)						

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	0	1	1	2	1	0
0	1	3	5	3	2	0
0	1	1	3	2	2	0
0	1	2	3	2	1	0
0	0	0	0	0	0	0
Glider (5, or 1 translated)						

Figure 7: The glider pattern in action.

Later with the development of computing and translation of the game of life into a digital form, more complex patterns came to the surface. Figuring out the patterns was a difficult mathematical problem and there were many researchers trying to come up with patterns that would prove or disprove a theory.

Even though Conway's game of life contains the word "game" in its title, it cannot really be played. However, it is an interesting mathematical model that can be analysed and studied.

The combination of the simple rules and fascinating results make this model worth studying when working with any kind of artificial life.

Figure 8: Conway's Game of Life.

Conway's game of life proves that the A-Life which is not very complex and has a set of simple rules is capable of generating unique and unexpected results.

2.4 Simulations of creatures

2.4.1 Virtual creatures of Karl Sims

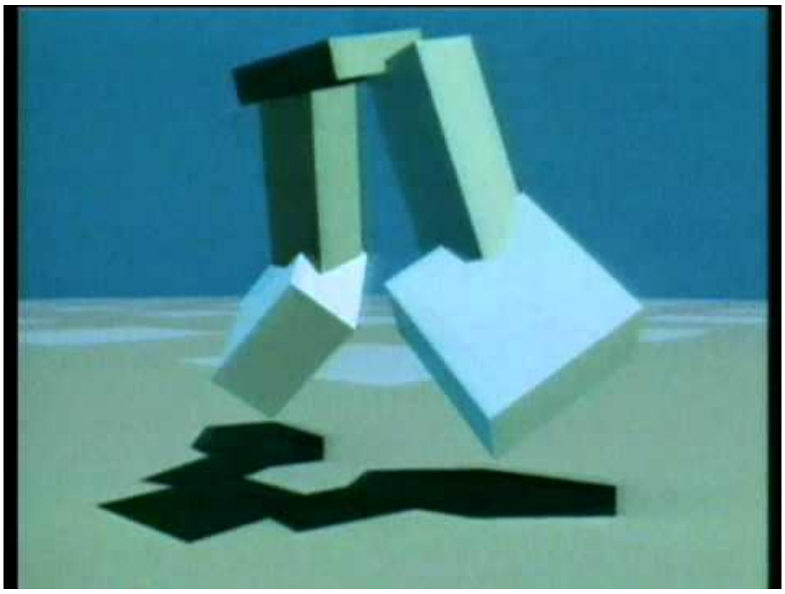
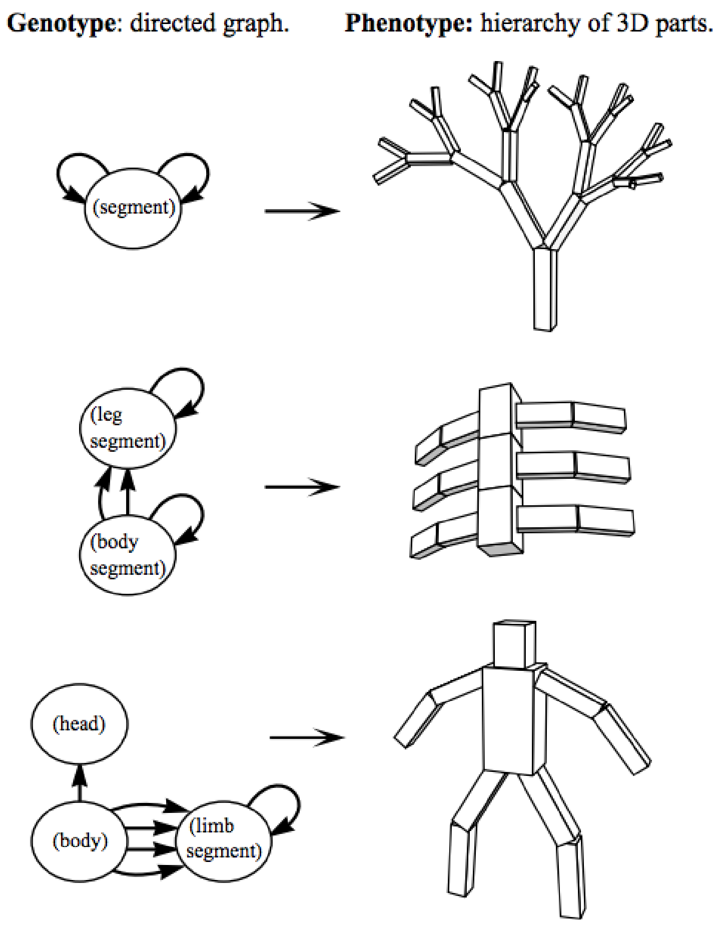


Figure 9: A creature of Karl Sims that evolved sinusoidal motion in water.

The research of Karl Sims plays an important role in the development of this project. In his paper, "Evolving virtual creatures", he had created a simulation of living organisms that can evolve their bodies in order to facilitate the achievement of a set goal. His program generates structures of different creatures out of polygons within a physics engine. They can develop various limbs with different types of joints that would enable them to move. The type of joint or limb is predetermined by a genotype using GA. ANN of each individual is also a product of many generations of genetic crossover which allows the program to produce creatures of high complexity.

“Evolving Virtual Creatures” is a paper written by Karl Sims in 1994 about the evolution of artificial creatures. It describes the use of a combination of techniques involving the use of both ANN and the EA in order to evolve virtual creatures.

This paper is important not only because it explores a unique technique of evolving the digital organisms but also because it offers a unique perspective on the field of artificial life.



Genotype: directed graph.
Phenotype: hierarchy of 3D parts.
(segment)
(leg segment)
(body segment)
(head)
(body)
(limb segment)

Figure 10: Use of L-systems to generate limbs as shown by Karl Sims in "Evolving Virtual Creatures".

While the creatures of Karl Sims are not completely intelligent, they are capable of solving various problems through their evolution and learning. They can evolve limbs and learn how to operate them, an ability that can be particularly useful in the field of robotics. It also offers a technique to generate 3D models that will then adapt to their surroundings and perform a specific behaviour.

The creatures themselves are 3D objects generated of multiple polygons within a simulated universe. Each one of them operates in accord with a combination of ANN and GA techniques. The creatures form their bodies using L-systems as a basis.

In the first phase of the program, creatures are evolving. Their evolution works in accord with Darwin's law where unfit ones are eliminated and only the fit ones get to reproduce with an element of mutation present.

The fitness of the creatures is determined by how quickly/effectively they can move through space. So the creatures that reach a particular point faster have higher fitness. The effectiveness of their movement is determined by their evolved limbs and control over them.

Interesting examples of accomplished variety could be the water creatures that were evolving either fins or a snake-like body with sinusoidal motion, ground creatures developing leg-like structures, or rolling behaviour developed through the use of a tail.

The project of Karl Sims is undoubtedly important within the field of A-life. It shows the capability of virtual creatures to evolve over time and develop physical characteristics along with the ability to control those new features using EA techniques.

The movement and control over body parts was accomplished with the aid of ANN which could evolve new nodes using GA. This simulates the evolution of a nervous system. Resulting generated complexity was high, but on the good side, a person working on it didn't need to understand it as long as the results were seen. According to Karl Sims: "A control system that someday actually generates "intelligent" behaviour might tend to be a complex mess be-yond our understanding."

While the creatures can evolve new appendages and accomplish a simple task it is still unclear from the work how tasks of a higher difficulty could be accomplished using this system. Each of the Karl Sims's creatures evolves with a determination to accomplish one simple task to a better degree than other individuals, however, the efficiency of the system functioning with more tasks is not shown.

The program can dynamically gain in complexity by using GA which results in creatures evolving new capabilities. While the system gives the creatures a freedom to evolve, it sacrifices control over itself. An aesthetic user selection is possible if a user wants more control, however, in the real situation it would be a tedious task. It is also eliminating the purpose of system in the first place which consists of creatures evolving by themselves, using their own means.

De Garis has evolved weight values for neural networks, Ngo and Marks have performed generic algorithms on simultaneous pairs and van de Panne and Fiume have optimised sensor-actuator networks. Each of these methods has resulted in successful locomotion of two-dimensional stick figures. Here, the entire creatures are designed using the algorithms, unlike in the projects described above the nervous system also evolves and grows in complexity.

Karl Sims had succeeded in creating a system that generates complex and interesting creatures without cumbersome user input, design knowledge or algorithms. The system is able to simulate the evolution of species to a high degree. It had also proven to be able to evolve appendages and their use, with resulting behaviours similar to the ones in Nature.

There is unlimited space of creatures to be explored. The extreme variety increases even further by introducing a user input in the form of aesthetic selection.
In addition to other pros of the system, complex results produced by it are the reason for applying an approach of Karl Sims to more projects. As computers become too powerful, the human might no longer be able to manually create a complexity that would utilise the full potential of a computer, so the generated system used in the project of Karl Sims is a possible solution to the problem.

The downsides of this project are mainly related to very small user control. The creatures are not really customisable and almost everything is left in the hands of simulated evolution. Another negative is that the resulting creatures are too basic. The only feature that is being evolved is locomotion and no other factors are taken into account. According to the paper, Karl Sims suggests that it could be expected in the future to afford creation of more complex structures - like scales, fur, a rigid skeleton covered in flexible skin, etc.

The environment is very simple. The simulated world is either a plane or a body of water. There are no obstacles, predators or other elements that could interfere with evolution. This lack of complexity in the world results in a small level of realism in the model.

Karl Sims has a similar paper called “Evolving 3D Morphology and Behaviour by Competition”. In this work, he uses the same type of creatures from his previous paper. The main thing that makes this project different is that the creatures in it are evolved through competition.

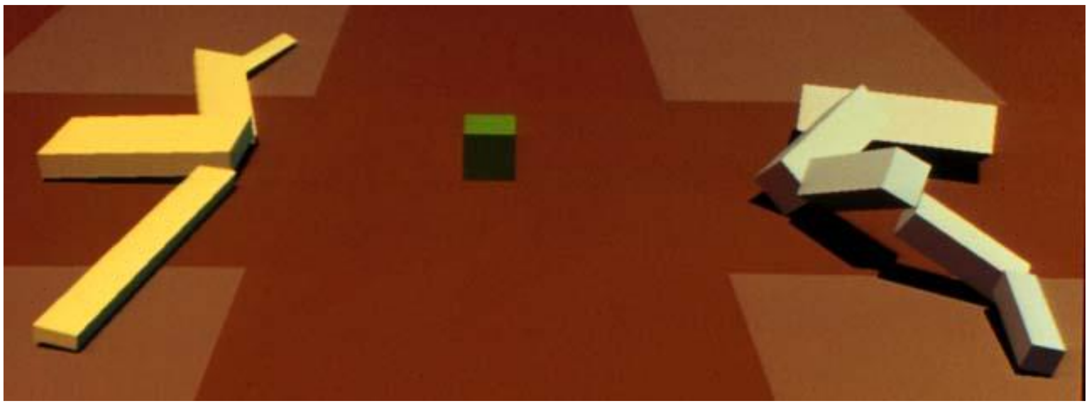


Figure 11: Competing creatures of Karl Sims in his application. Note the goal (cube) positioned in the middle.

There are several types of competition available - all vs all, tournament, random, etc. In each competition a creature is pitted against another creature, selection of which determined by a competition type. They compete for a goal (for example, reaching a cube) and as a result of competition, the fittest individual is determined.

The fitness is defined by the distance between the centre of the creature's mass and a 3D object (cube) for which it is competing. The object has a mass and physics applied to it. In figure 11 two creatures competing for the possession of a cube are shown.

For a time there was a problem where taller creatures could easily get hold of a cube by falling over it. It was solved by placing a creature further back or relaxing the bodies of the creatures immediately after placing them.

As a result, the creatures evolve not only the locomotion techniques in order to get to the object but also appendages with grabbing capabilities which can even move the object in order to decrease the distance between the object and an opponent.

2.4.2. Polyworld

Some of the downsides found in the work of Karl Sims were later solved in another project written by Larry Yaeger with the goal to evolve artificial intelligence through natural selection with GA. The project was started in 1994 and it is still ongoing.

Polyworld (Figure 12) is an artificial ecosystem inhabited by the virtual creatures where they get to adapt to the environment rather than to achieve a simple goal.

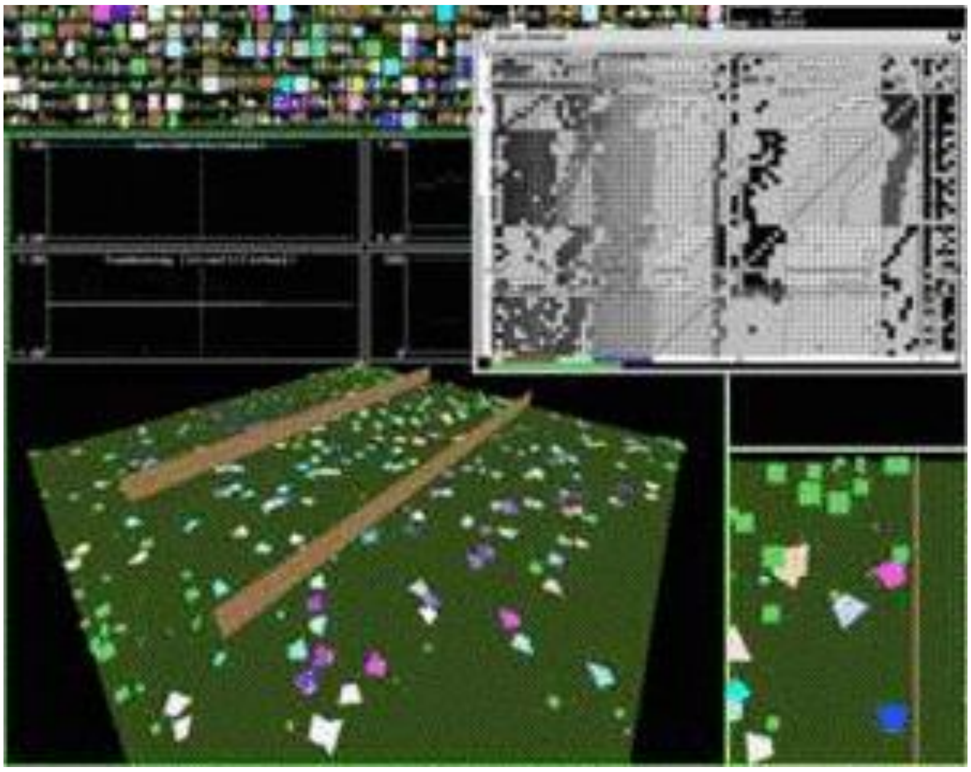


Figure 12: Screenshot from Polyworld showing the world populated with creatures, a grid of neural network weights (right) and fields of vision of each creature (upper left).

Polyworld is a 3D simulation of a pocket-sized world. It consists of a plane and the creatures inhabiting it along with many other elements such as obstacles and plants. Creatures can die, reproduce, eat, communicate and kill other creatures. Each one of the creatures has an energy pool which can be depleted by performing an energy taxing activity such as reproduction. Food restores the energy. When the energy decreases beyond the minimum, the creature dies.

Each of the creatures is equipped with a number of sensors and uses ANN. The GA is used so an offspring of a creature can inherit properties of a parent.

A prominent feature in the Polyworld is that the creatures can communicate with each other at a certain level. The communication is established through a visual change of colour which can theoretically result in various signalling patterns. While in practice this feature is, in most of the cases, rudimentary, it finds its use in certain cases such as expression of violence or reproduction.

This project went a step further from the virtual creatures of Karl Sims. Creatures in the Polyworld can develop complex life cycles and they don’t have a determined fitness function - the best ones simply survive like in real world. The environment, energy, life, sources of nutrition and new interactions between the creatures offer a broader view of a model of the evolution of species. It is possible to study the behaviours of interaction between the creatures.

A downside of this project is a lack of realism of the creatures. Scenarios presented in the Polyworld are highly stylised to correspond to the model and possibilities of computation. Creatures cannot evolve complex structures like fur, scales, feathers or a skeleton.

Another downside is that the world itself is lacking detail. Whilst it contains obstacles, food and other elements, it still lacks a lot of features which would be crucial to the evolution of creatures. Such features could be opposing species or world elements such as wind, temperature, humidity.

Understandably, evolving communication is not a major preoccupation of this work, even though a way of signalling was added for the creatures. Some level of communication had been established between the agents nevertheless. The evolution of “speech”, in this case, was provoked by the necessity to report reproduction intention, which seems necessary because it is established between two creatures. The violent mood communicated by the change of colour was a necessary measure for the species to achieve survival.

2.4.3 Flocks, Herds and Schools

While both the research of Karl Sims and Polyworld are playing an equally important part in this study, it is important to mention a paper that made me think more globally about the topic and imagine my creatures in a group rather than as individuals.

When we observe a school of fish or a swarm of insects they can almost be perceived as a single organism. The “Flocks, herds and schools” is a title of the paper from 1987 by Craig Reynolds which served as an eye-opener to me when it comes to thinking about artificially evolved creatures. Even though it was published many years ago, the technique described in the paper is still relevant and is widely used within the animation industry. It is also one of the major sources of inspiration for this thesis.

The paper was written in order to develop an animation technique for film in order to be able to produce a realistic animation of a moving flock. Due to how animation works, in order to produce a realistic animation of a flock, an artist would be forced to draw each bird and animate it separately in order to achieve a good result. It would be an extremely tedious task and a realistic result would only scale with the animating skills of an artist. In order to automate this process, an algorithm was needed.

Craig Reynolds came up with an algorithm when he was observing birds moving in a flock. A flock is essentially a survival mechanism that emerged in species that move through 3D space, especially in low visibility scenarios. Like fish that move through murky water obviously have no notion of what is around them, therefore they are forced by conditions to form schools.

Low visibility is something that makes schooling essential for survival because within a group like that an individual does not need to have a good visibility. In order to notice the food, obstacles, and predators - it just needs to follow other individuals in a flock. Fish scales that are reflecting light help them to be noticed by other members of a school. Therefore the algorithm was designed after a natural phenomenon of creatures that form schools and flocks.

However, variations of this algorithm are considered to be more applicable to fish that form large schools in murky water with low visibility.

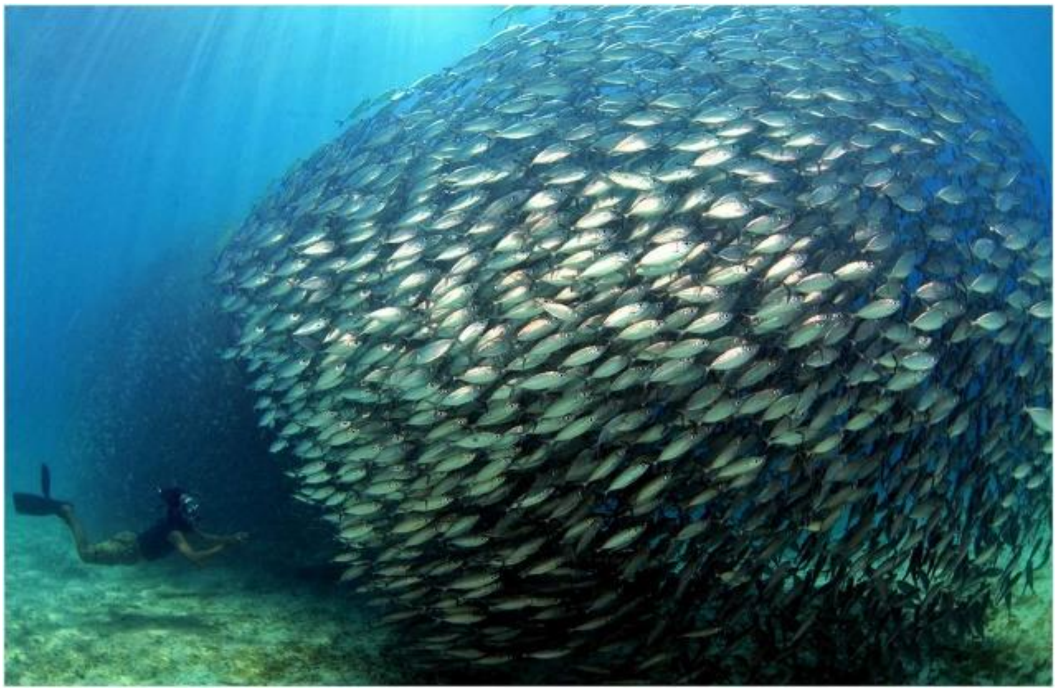


Figure 14: Schooling fish forming what seems to be a whole new organism.

What would be the rules that an organism within the flock would follow? It is fairly obvious to assume that it would be important for each fish to follow the general mass of a school because becoming isolated in a dark water means becoming a good target for a predator. Equally important here would be catching up with the average speed of a school. And finally, the most important rule would be avoiding collision with other fish and other types of obstacles.

Therefore the three rules of flocking are Alignment, Cohesion and Separation. Using these simple rules, Craig Reynolds was able to design a flocking algorithm.

Within the artificial flock of Craig Reynolds, each individual is called “boid” (bird-oid). Each boid is a little computer in itself and is also a centre of its own coordinate system. The vision range of a boid is something that defines the size of its coordinate system.

What makes this algorithm realistic is a geometrical flight function of each boid. They exist in a more or less realistic simulation of the physics system and each boid can pitch, yaw and roll, emulating a flying body. There is the possibility of a stall which happens when a body loses the velocity during the flight and also there is flight inertia - a flying object cannot stop. When starting to program a flocking algorithm it’s advisable to start implementing the geometrical flight for each boid because with it helps achieving realistic results.

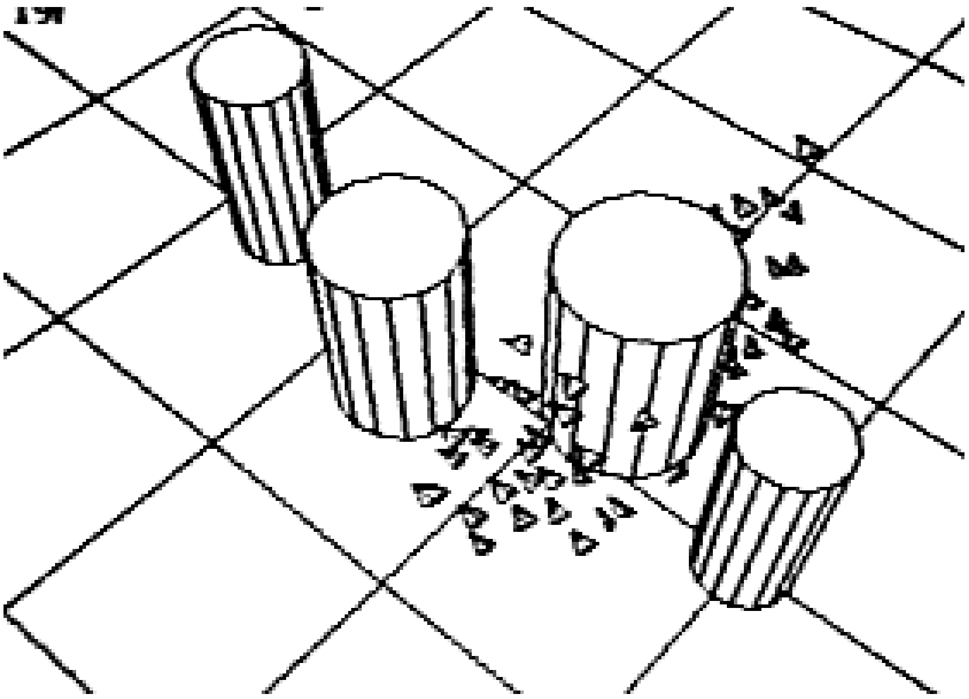


Figure 15: Flocking of Reynolds in a custom environment.

A difficult part of writing this algorithm is combining the three rules of flocking in the right way, so a boid would know which rule is the priority in the current situation. (Collision avoidance rule, of course, would be a priority in all of the situations). Additionally, a boid should “know” which rules can be ignored or performed to a lesser degree without significant loss.

The algorithm of Craig Reynolds is extremely widely used and in some physics engines it is a default feature. It can be used in cinema, video games and as a minor special effect in any other digitally produced arts.

The flocking algorithms are compatible with the possibility of more complex environments with include predators and more complex obstacles. They can also be adjusted easily to work with different species of flocking, swarming or schooling creatures. Broad capabilities together with simplicity of this algorithm make it an important tool to reference in this project.

However, the flocking algorithm has some downsides. The major one (at least at the time when the original paper was written) is the processing speed. While nowadays it's not such a big issue, some problems can arise when using a very complex flocking algorithm with a large number of boids.

Another downside is a lack of detail in movement of each individual. The paper doesn’t include information about factors like wing movements, head rotation, etc. The only factors that are being calculated are general direction, speed and angle of movement. Additionally, the algorithm is rather applied to fish schools with low visibility factor rather than bird flocks or fly swarms (even though it may still be adjusted to be used in those examples).

As I figured out later, the algorithm can still be very fast in real-time rendering if the actors are simple planes with animated texture and rendering is still done on CPU level. The down-side of this would be the fact that at a closer inspection one can see the lack of detail of the boids, even though the overall image would still be realistic. A similar technique was observed in various computer games that have rivers and oceans with fish (World of Warcraft), the designers use this when real-time rendering is required.

2.4.4 Neural studies

Artificial creatures are not only created with the goal of producing animation or studying the natural process of the evolution of species. There are several A-life projects aimed at understanding the biology of an animal by creating a computational copy of its brain and nervous system. One such project is Neurokernel.

According to their own definition, The Neurokernel Project aims to build an open software platform for the emulation of an entire brain of the fruit fly Drosophila melanogaster on multiple GPUs. The development started in 2011 and it is still ongoing.

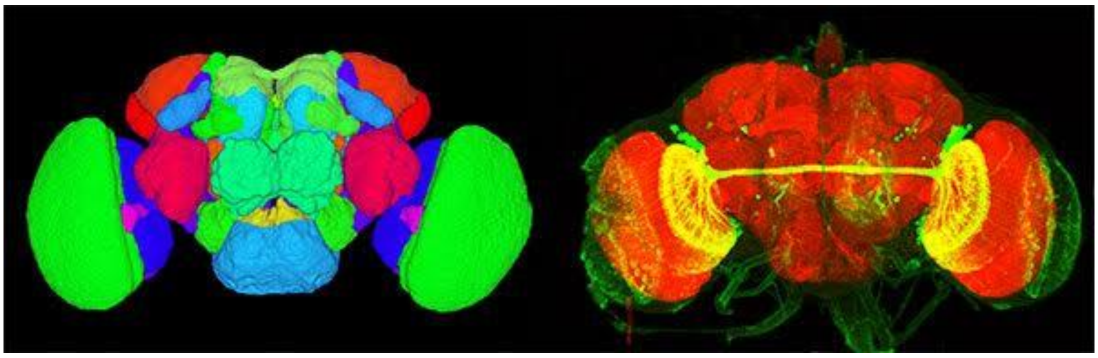


Figure 16: Visualisation of a fly brain from Neurokernel project.

The development of this project is conducted among a great number of researchers where each creates a separate model which is then being connected and integrated with the rest of the models using their Neurokernel software.

According to Neurokernel, animal behaviour is governed by the activity of interconnected brain circuits. Comprehensive brain wiring maps are thus needed in order to formulate hypotheses about information flow and also to guide genetic manipulations aimed at understanding how genes and circuits orchestrate complex behaviours. A successful determination of how a brain’s highly complex structure implements specific functions requires its decomposition into functional modules whose input-output relationships can be individually analysed and whose interactions can be explained in terms of the groups of synaptic connections that exist between them.

The importance of this project can be justified by how significant it would be for medicine and many other fields to be able to finally replicate a working model of a human brain. The human brain is extremely complex, the computers of our days are unable to process such high complexity and creating such a model manually would be a task far too complicated. Therefore starting with a brain of a small animal with a more basic brain structure is the first step towards creating a model of a human brain.

While Neurokernel is an important project that is open for public and contributors from outside of the main research group, and it can, later on, serve as a guideline for designing a brain for constructs with AI, it is still in the process of being designed and the effectiveness of the final result is not yet guaranteed.

As was mentioned earlier, there are many projects that aim to create a working computational model of a creature, therefore here is another project that shares the same goal: OpenWorm. As the description on their site states: “OpenWorm aims to build the first comprehensive computational model of the Caenorhabditis elegans (C. elegans), a microscopic roundworm. With only a thousand cells, it solves basic problems such as feeding, mate-finding and predator avoidance. Despite being extremely well studied in biology, this organism still eludes a deep, principled understanding of its biology. “

It’s an open source project collecting contributors from around the world to build a model of a roundworm. Even though the roundworms are believed to be simple organisms, the brain of such a small creature is still very complex.

The OpenWorm project is similar to Neurokernel, it is not being built with the purpose of entertainment. Its main goal is to research the biology of a creature using a computational model. It is a relevant example of how artificial creatures can contribute to our society. Both Neurokernel and OpenWorm are mentioned here as more scientific examples showing how accurate the computers can be at reproducing a copy of a biological neural network of a living being and are a glimpse into the capabilities of modern computers.

2.5 Living agents in entertainment software

2.5.1 Black & White

Black & White is a “God simulator” game from 2001. The game was developed under the direction of Peter Molyneaux. Previously he was famously working on such games as Populous and Dungeon Keeper. Both of the games are top-down strategies. Populous is an earlier god simulator and Dungeon Keeper is a game where the player takes on the role of an overlord of a dungeon full of monsters, where the main objective is to protect the treasures from questing heroes.

While in Black & White the player takes on the role of a god, one of the key features of this game is a presence of an avatar - a giant creature that can be trained by the player. The creature can learn the simple concepts of good and evil and it can develop certain traits depending on the player's actions. For example, the player can choose to punish the creature after it eats a human, or to reward it instead. Depending on this choice, the creature will change its behaviour.

The main objective of the game is to increase the belief of the in-game villagers in you as their god through using the creature and some special abilities.

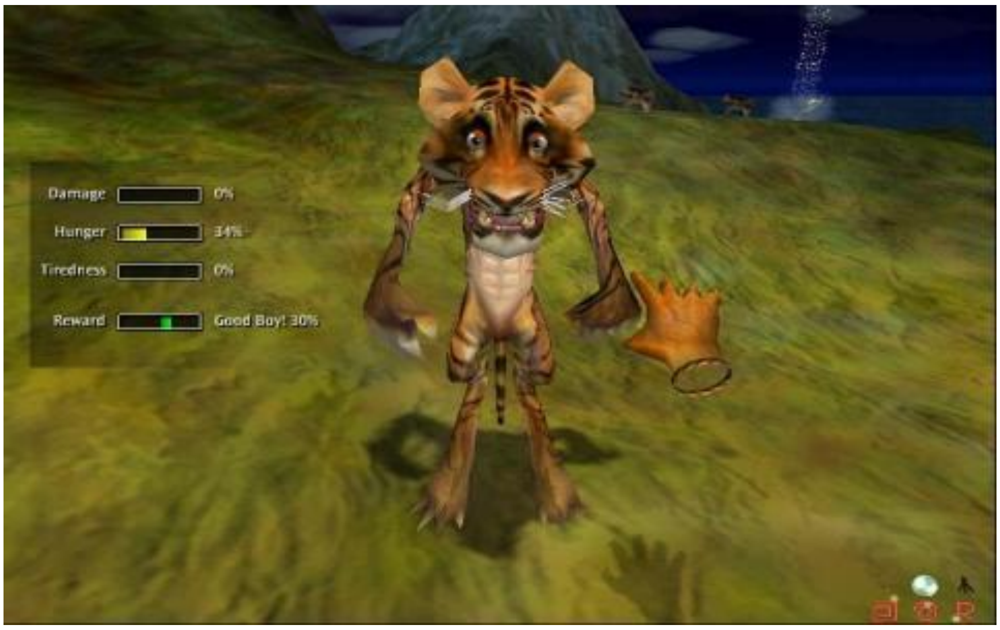


Figure 17: Screenshot showing a creature from Black and White. The hand of a user can be seen interacting with it.

The villagers have AI that was in no way innovative at the time when the game was released. It was the creature that was programmed in a sophisticated and innovative manner. The developers of the game intended to make the creature as alive and human-like as possible so the players would connect with it. The creature starts as a child, not knowing anything about the surrounding world and later grows up and learns things about the world. The chief AI developer for the game, Richard Evans, provided the gaming website www.gameai.com with some simple documentation of the game design.

What makes the AI of Black and White so powerful is the mixture of different approaches to representing the intelligence. This idea was taken from Marvin Minsky's years of research about AI at Massachusetts Institute of Technology. In particular, three representations were used. Symbolic attribute-value pairs are used to represent a creature's belief in an individual object. An example of this type of representation is:

```
The strength of obstructions to walking:
object.man-made.fence->1.0
object.natural.body-of-water.shallow-river->0.5
object.natural.rock->0.1
```

This method is used alongside rules-based AI (situational calculus) to give creatures their basic intelligence about objects. This scripted-AI is the most popular form of artificial intelligence found in games today. Decision trees represent the agent's beliefs about general types of objects. Finally, neural networks of perceptrons represent desires.

For example, a creature knows from birth that eating will satisfy its hunger. That is a natural instinct. But, it does not yet know what type of object satisfies its hunger best. It may see a fence and instantly walk over and take a bite out of it. It will then realise that the fence did not satisfy its hunger well and tasted horrible in the process. The creature will alter its internal food decision tree in order to keep it from eating fences in the future.

Black & White is an important example because it is an attempt at introducing a learning creature into a video game. The AI of the game was also a big breakthrough at the time when the game was released.

In this example, we have a video game and a creature that can be trained. However, the creature in this game cannot really be considered as a living organism as it does not exist within an ecosystem, interact in a natural way with the ambience or have any features associated with the realisation of the natural life cycle. The hybrid nature of AI limits the living computation possibilities in this example with the purpose of introducing the element of fun and control to the user.

2.5.2 Nintendogs

Nintendogs is a game released by Nintendo in 2005. This game is virtual pet simulator for the Nintendo DS platform. The user can adopt a virtual pet and take care of it. One of the features of interest in this project is the voice recognition feature.

Figure 18: Pet from Nintendogs.

While the AI of the pets is not introducing anything new into the field, the voice recognition system is an interesting addition to the game. When the game starts, a user is asked to introduce a name for the pet. After a name is introduced a user will be repeatedly asked to say the name into a microphone. Through an analysis of multiple examples, the system will be able to recognise the voice of a user pronouncing the name of a pet. The mechanism involves ANN that learns to recognise a particular command.

While it takes a lot of time to make the program finally learn the name of a pet, it is a quite rewarding experience to call a virtual pet and see it respond to its name. It is important to mention this game because it utilises ANN working together with voice commands for one of its core features.

The negatives of this game are that pets lack complexity. While they have a very realistic and advanced animation, their behaviour is basic and does not introduce any novelty into the field of AI or A-Life.

The voice recognition feature takes too many attempts for a pet to finally learn a name. I would call it realistic, but because it happens on a separate screen with only occasional testing on the pet itself, it loses realism.

In 2011 a new improved version was released for Nintendo 3DS under the title “Nintendogs + Cats”. This game features new improved voice recognition together with facial recognition. The names of pets and tricks don’t need to sound exactly the same during the play anymore. Another function that was added was facial recognition for the camera, so the pets can now recognise their owner. Apart from improved learning functions the game is still very similar to the original.

2.5.3 Creatures

Perhaps the closest example to the objective of this project is Creatures. It is an A-life computer program series developed in the mid-1990s by English computer scientist Steve Grand, whilst working for the Cambridge video games developer Millennium Interactive.

Rather than a game, it is considered a program or a toy. It features a 2D world with creatures called Norns which can be taught by a player.

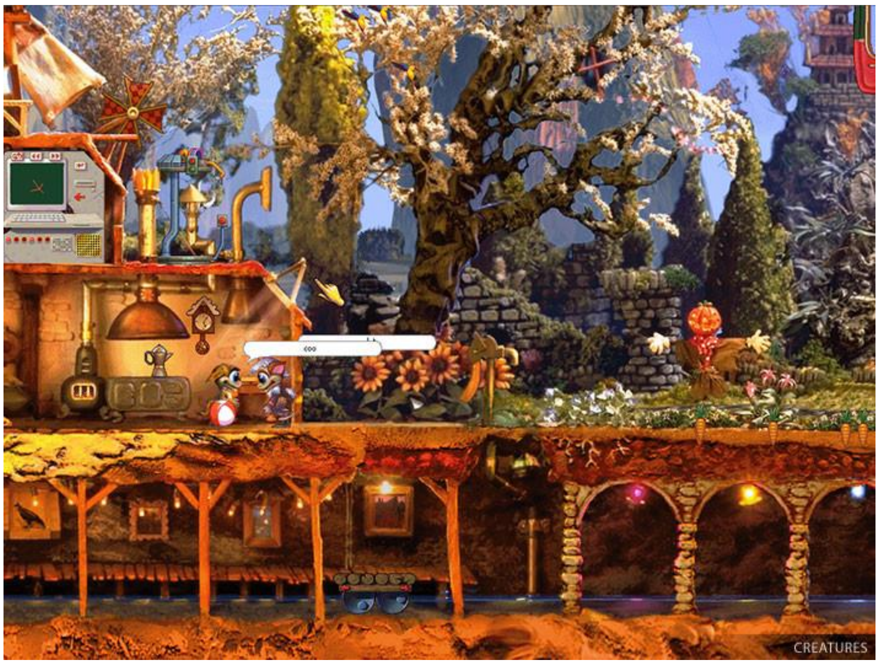


Figure 19: Screenshot from Creatures program, showing the world inhabited by Norns. Two Norns can be seen in the structure on the left. Their speech is presented in text bubbles above their heads.

A player assumes the role of something similar to a god and controls a hand which can interact with things within the world. In the beginning, a player gets a number of eggs from which the creatures can hatch. A new-born Norn has no knowledge of the world but it is curious about it. It is the task of a player to teach it by using reinforcement learning techniques. A player can type the commands and then can slap or tickle a Norn. Slap gives negative feedback in the form of a feeling that a Norn is doing something wrong, while tickling gives a good feeling of having done something good. Apart from these commands, a player can teach words to a Norn and also make it form a bond with the player's hand (cursor) by giving it a good impression of it.

Figure 20: One of the characters from Creatures, a Norn.

Success of this toy was arguably due to effective incorporation of A-life creatures. Unlike in similar games, creatures here react to changes in the world brought by a player and interact with a player's hand dynamically. The creatures also possess a biological life cycle with several stages such as adolescence, adult-hood and a senescence phase when the creature finally dies. The creatures can procreate, the offspring inheriting various features of the parents which introduces more variety into the game. A learning approach in a new generation might be different due to inherited features.

The Creatures series have brought innovation into the field of A-life partly because of the underlying model that generates such interesting behaviours.

AI in Creatures is close to being an accurate biological structure. It is not completely realistic and is adjusted in order to fit into the game. This becomes obvious in:

1. The way a creature perceives its world.
2. The way creatures inherit characteristics from their parent.
3. The way a creature learns new things.

As mentioned by Steve Grand in “Creatures: Entertainment Software Agents with Artificial Life”: “The network architecture was designed to be biologically plausible, and computable from the ‘bottom-up’, with very few top-down constructs.”

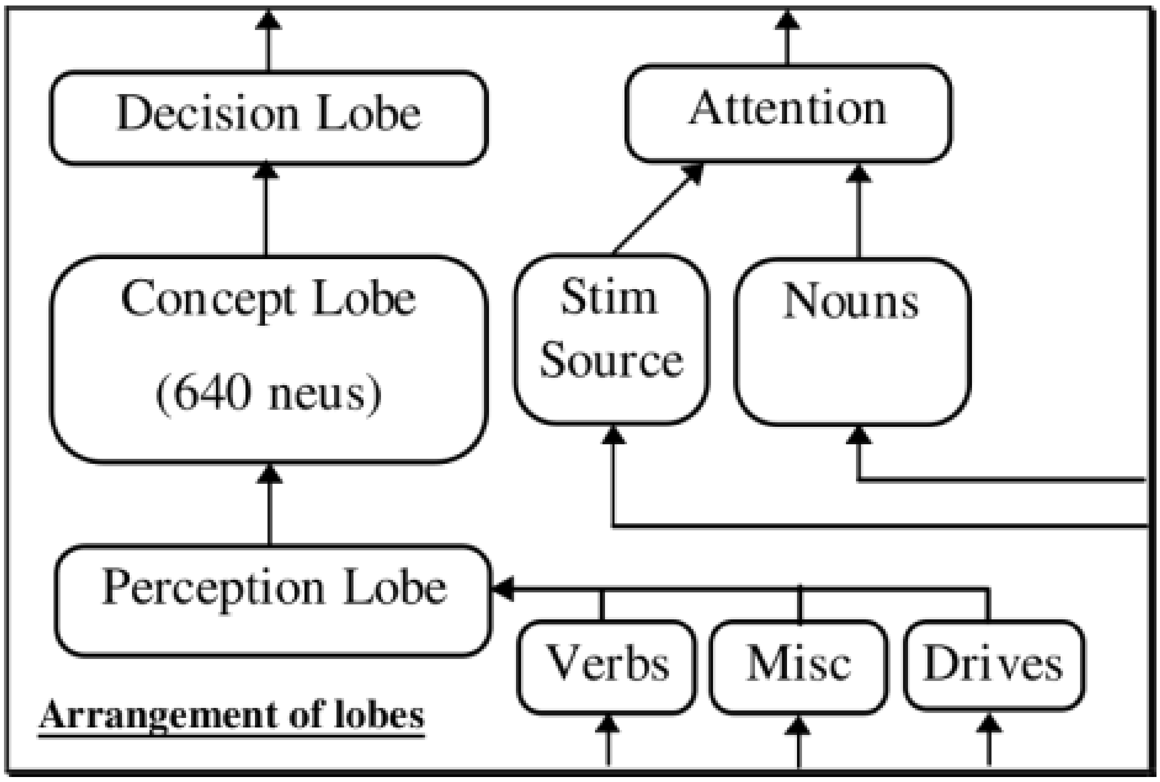


Figure 21: Brain structure of the Creatures of Steve Grand.

The game simulates sounds, touch and sight for each Norn. Sounds attenuate over distance and are muffled by any objects between a creature and a sound source. The result is a Boolean value used as an input for a neuron in the network.

ANN is something that is completely responsible for controlling the creatures. As shown in Figure 20 it is built in a modular way. It helps to deal with common unpredictability problems of neural networks and helps to make the training more accurate. As mentioned in "Creatures: Entertainment Software agents with Artificial Life": "Each creature's brain is a heterogeneous neural network, sub-divided into objects called 'lobes' […] Cells in each lobe form connections to one or more of the cells in up to two other source lobes to perform the various functions and sub-functions of the net."

The creatures are also capable of generating emergent behaviour by interacting with various objects present in the game, additionally, ANN is connected to a body which acts in an approximate biological way:

- The brain's decisions are subject to hormones in the body.
- The body may be affected by toxins ingested while eating.
- Norns have a metabolism that may be affected by bacteria in food.

As mentioned before, Creatures has an evolution system which adds a variety to players who play for multiple generations. Because the evolution can produce several changes in a genotype and some of them can be so extreme that they can break a creature, the developers implemented a post-mutation check.

Quote from "Creatures: Entertainment Software agents with Artificial Life" : "To prevent an excessive failure rate due to reproduction errors in critical genes, each gene is preceded by a header which specifies which operations (omission, duplication and mutation) may be performed on it."

It is important to note that even though behavioural traits are affected by evolution, mutation and learning, they can be subtle. Arguably the most important changes from a player's perspective are the visual ones.

Quote from “Creatures: Entertainment Software agents with Artificial Life”: “Creatures are bipedal, but minor morphological details such as colouring and hair type are genetically specified. […] The life-span of each creature is genetically influenced: if a creature manages to survive to old age (measured in game-hours) then senescence genes may become active, killing the creature.”

In general, this is a toy/program that incorporates A-Life agents successfully, taking such a conclusion from to its popularity.

After playtesting this toy, it was evident that the interface was not intuitive, as the action was happening in several windows at the same time and interactions were not clear at the very start. The creatures, however, seemed to be truly alive, acting in a realistic manner and learning words and commands typed by a user.

What was lacking, however, was mostly due to the processing power available at the time. The creatures could have much higher complexity in their behaviour, having more variables for their expression, emotions, animation and needs.

While the creatures could identify various objects using speech, they could not rely completely on sounds and most of the interactions were happening using sight. Speech itself was also represented by typed commands which the creatures learned to replicate and respond to. With this thesis, I intend to make sound play a much more significant role in the life of artificial creatures.

2.6 Machine learning technology

2.6.1 Introduction

ANN is a computational model inspired by the network of neurons that can be observed within the brains of various animals inhabiting our planet. Neurons are cells that transmit information within a nervous system, generating the response to stimuli from the environment. The number of neurons in different parts of the brain determines the neural function and behaviour of an individual. While the human nervous system contains around 86 billion neurons, where roughly 16 billion are in the cerebral cortex, many simpler organisms like sea squirt or roundworm possess less than 1000 neurons.

When a machine is required to perform a human function, such as recognise images, speech, patterns, etc., a possible way of accomplishing it is by creating an ANN that will perform the task. While ANN are not used in neuroscience studies because they are an oversimplified analogy to axons in a biological brain and don't function in the same way as their biological counterparts. Biological neural networks are difficult to model in detail because they are composed of large numbers of nonlinear elements and have a wide range of time constraints. There is no evidence that some neural computing algorithms such as backpropagation represent actual brain mechanisms of learning, therefore most neural network architectures are an example of "neural inspired" modelling, not modelling of actual brain structures.

2.6.2 Brief history

The simplest history of neural networks would start with three items: McCulloch and Pitts (1943), Hebb (1949) and Rosenblatt (1958). These publications introduced the first model of neural networks as "computing machines", the basic model of network self-organisation, and the perceptron model of "learning with teacher" respectively. To fill the story, we would have to review the neural network models of vision, memory, motor control, and self-organisation studied in the 1960s and 1970s by Amari, Anderson, Cooper, Cowan, Fukushima, Grossberg, Kohonen, von der Malsburg, and Widrow. However in order to keep this brief, I will move to a more practical approach.

Figure 22 : Photograph of Mark 1 Perceptron machine of Rosenblatt, as seen at MIT exhibition.

2.6.3 Perceptron

The most basic model of a neural network is perceptron. While in 1958 this word was used by Rosenblatt to describe his physical “Mark I Perceptron” neural network machine (figure 22), it is now used in literature to describe a very simple neural network consisting of one layer and having a binary input/output, as the perceptron learning rule was invented by Rosenblatt in 1950.

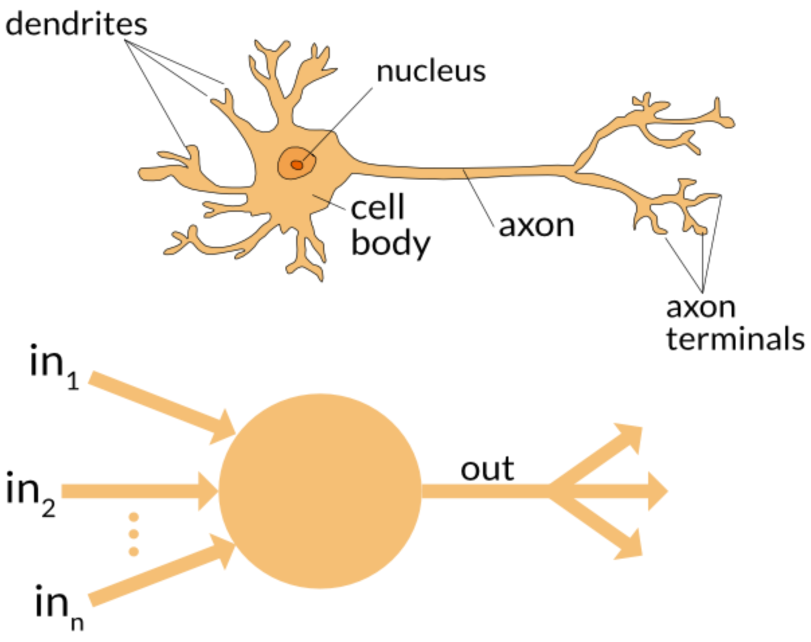
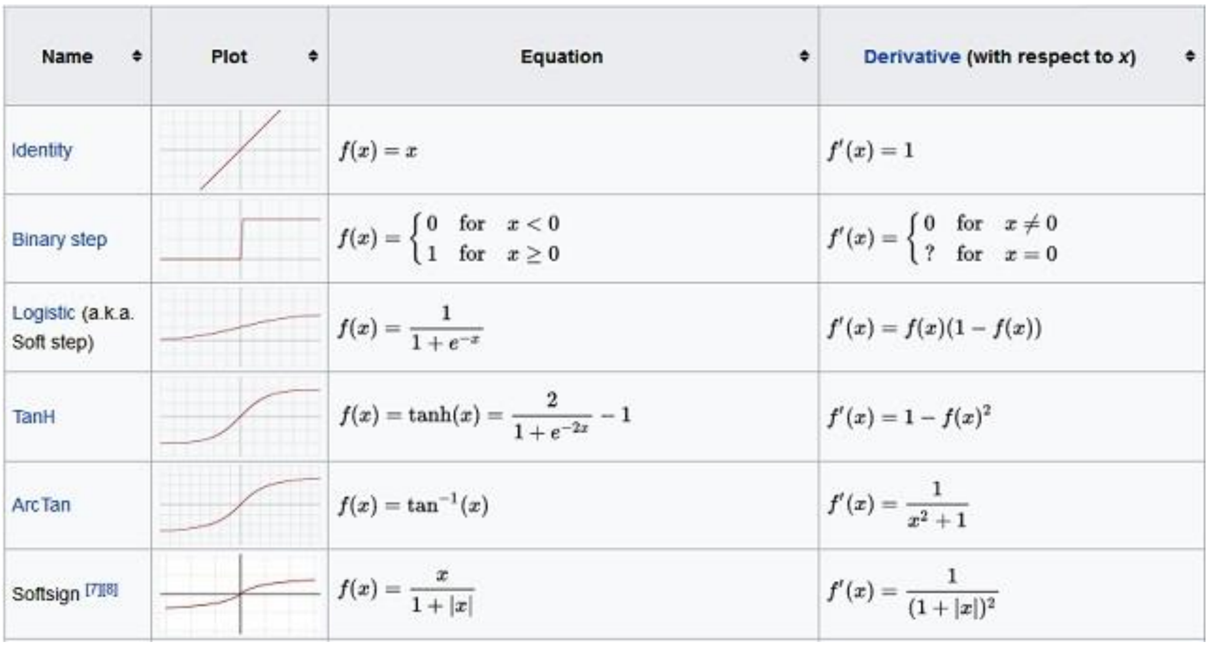


Figure 23 : Schemes of biological axon (up) and perceptron (bottom).

Similar to a biological axon, perceptron receives inputs from the environment, processes them and generates an output. The training of perceptron is accomplished through updating weights, comparing each guess of the network to the right answer. As can be seen in Figure 23, in1, in2 and in3 on the perceptron scheme are the inputs. Arrows pointing from them to the body of perceptron are connections which have weights attributed to them. And there is also an output function which returns the final result.

The process that allows the calculation of an output is called feed forward. In a simple perceptron, the inputs are multiplied by weights corresponding to them and then summed together to calculate a final number. In order to transform an output number into a binary or a more compact output, it is necessary to apply an activation function. In Table 1, different activation functions are shown.

Table 1: Activation functions.



Name	Plot	Equation	Derivative (with respect to x)
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x \rvert}$	$f'(x) = \frac{1}{(1+\lvert x \rvert)^2}$

After the output is received, in supervised learning (learning with a teacher), the weights are adjusted according to a desired (correct) result given by a teacher. In a basic perceptron this function is simple:

$$w_j(t+1) = w_j(t) + n(d-y)x$$

$where$:
$w = the\ weight$
$d = desired\ output$
$t = iteration\ number$
$n = gain\ or\ step\ size, \qquad where\ 0.0 < n < 1.0$

Perceptron is a neural network that can learn according to small sets of inputs, however when the input is bigger and more than one output is required, the neural network of higher complexity is needed. Such networks are known as Deep Networks (DNN) and they had been used in this project in order to achieve desired results.

2.6.4 Deep Neural Networks

DNN are composed of multiple layers. Each layer consists of multiple perceptrons. In all of the layers, inputs are outputs of perceptrons from previous layers, except the first layer, where the inputs are the state of the world or any other variables that serve as an input for the entire network. There can be any number of layers and any number of nodes (perceptrons) in each of them, however, a network is only considered deep when it has one or more hidden layers. Hidden layers are additional layers between the input and output layers (Figure 24).

Because of the bigger number of weights and layers in DNN, a higher complexity backpropagation algorithm is required. Backpropagation is the function where a desired output is propagated backwards through a neural network and the weights are adjusted to fit the desired result.

For backpropagation in this project I mostly used Mean Square Error (MSE) function, as it is the most common error cost expression that is used for neural networks:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y^i - \widehat{Y_\iota})^2$$

$where$:

$Y = vector\ of\ observed\ values\ of\ the\ variable\ being\ predicted$

$\hat{Y} = vector\ of\ n\ predictions\ generated\ from\ sample\ of\ n\ data\ points\ in\ all\ var - s$

This function allows calculating the error for each of the nodes and creates a gradient descent, which means the layers further from the output layer will have smaller error values associated with them.

Derivation calculations for the output layer in the code were different from the ones for hidden layers. The reason for that is because the output layers are where the error cost expression is first calculated and hidden layers are where the backpropagation of the error cost expression is taking place.

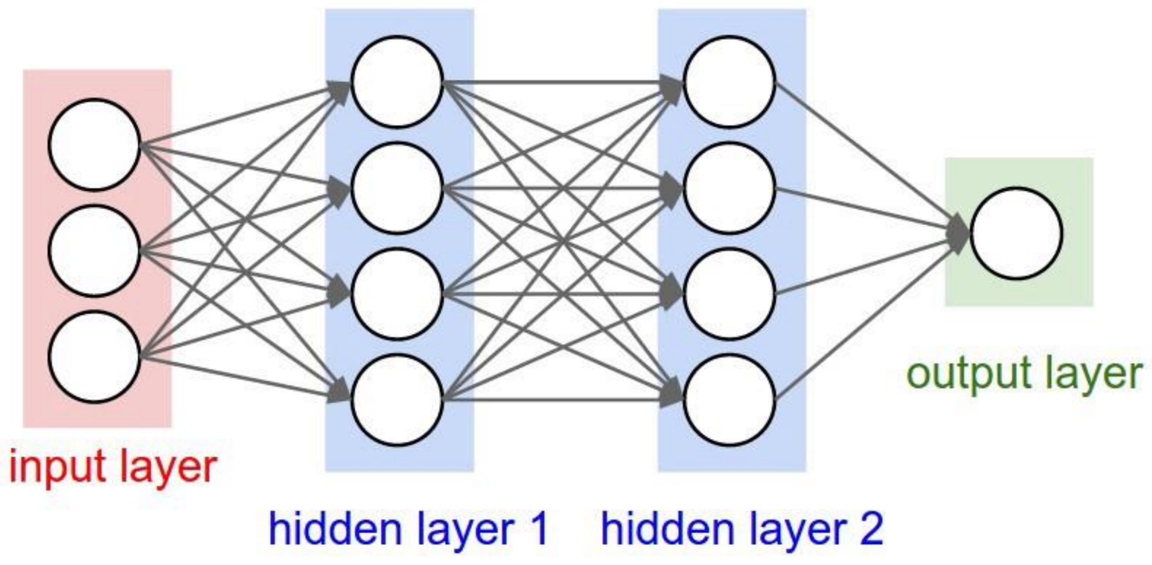


Figure 24: Schematic representation of a DNN.

It is necessary to mention that the backpropagation function is not always necessary. Instead of backpropagation is it possible to use EA (in particular, GA) in order to adjust the weights of the network. GA allows avoiding the use of backpropagation, but it doesn't provide learning for a unique individual, rather for the entire population. More details on the history and functionality of GA is available in subsection 2.2.3.

With this technique, the ANN would be evolved, rather than taught. A large population of individuals is created, their weights in networks are randomly distributed. After that the program runs a test or competition where it determines which of the individuals have higher fitness and are therefore allowed to reproduce. Fitness can be determined using a function, or can be attributed by a user (aesthetic selection). It is also possible to determine fitness by survival, where individuals die in cases of failure and only survivors reach reproduction.

These methods allow selecting the fittest individual out of the entire population. However, for the evolution to take place it is necessary to include the factor of mutation. With it, the offspring of individuals, instead of always having a copy (or crossover result) of their parents ANN, is going to sometimes be born with some weights altered according to mutation algorithms. This allows exploration and testing of different variants of networks that could be more efficient than the current fittest network.

2.6.5 Q-Learning

Q-Learning is another important technology that is necessary to discuss as it is directly related to Deep Q-Learning which is relevant to this project. Q-Learning is a type of reinforcement machine learning and it is table based. It was first detailed in a Cambridge PhD thesis by Christopher Watkins in 1989. It is inspired by the conditioning methods applied to animals where the desirable behaviours are rewarded and undesirable are punished (through negative reward).

A Q-Learning agent has a state. The state consists of observations that agents can make. The state could be the output of the sensors or various data that an agent received about the world. An agent can select an action which will then transition it to another state. Each action has a quality, based on its potential to result in a reward. The goal of Q-Learning is to determine the correct action to take given the current state. It is known as policy.

The state-action pairs are saved into a table and attributed a quality. While initially an agent will be performing random actions, it will be improving the quality attribution to state-action pairs and soon the correct policy will be discovered. It is done via the use of the Bellman equation:

$$Q(S_t, A) = \propto [R_{t+1} + \gamma \max Q(S_{t+1}, A)]$$

$where:$

$Q(S_t, A) = The\ quality\ of\ each\ action\ given\ the\ current\ state$

$\propto = Learning\ rate$

$R_{t+1} = Reward\ received\ after\ taking\ action$

$\gamma = Discount\ factor$

$maxQ(S_{t+1}, A) = The\ highest\ quality\ given\ the\ next\ state$

Learning rate is a value from 0 to 1 that determines how much the current Q-value is overwritten. Discount factor is also a value from 0 to 1 and it controls how much the future Q-value prediction is to be trusted. When the state is terminal (final), the equation is:

$$Q(S_t, A) = \propto * R_{t+1}$$

The problem of Q-Learning is the complexity. The Q-table needs to contain many entries even for the smallest examples. Even for a simple tic-tac-toe game, accounting all the possible game combinations and symmetries there are almost 27000 entries. Any application with any real complexity needs ANN to approximate the behaviour of a Q-table.

2.6.6 Deep Q Learning

In 2013 DeepMind released “Playing Atari With Deep Reinforcement Learning” and in 2015 “Human-Level Control Through Deep Reinforcement Learning”. This introduces us to using Q-Learning in conjunction with ANN, which results in Deep Q-Learning (DQN).

To train an ANN of a Q-Table it is required to have two feed-forward passes and one back propagation pass. On a first feed-forward pass we get an estimate of the quality of each action, then the highest quality action is selected from Q-Table and it is applied.

Then the next state is observed and passed through the ANN. Then the largest Q-Value in the output is used in the Bellman equation to calculate a target value. Finally, MSE between the first feed-forward pass and the target is calculated and backpropagated though the ANN. In figure 25, the flowchart of the DQN cycle can be seen.

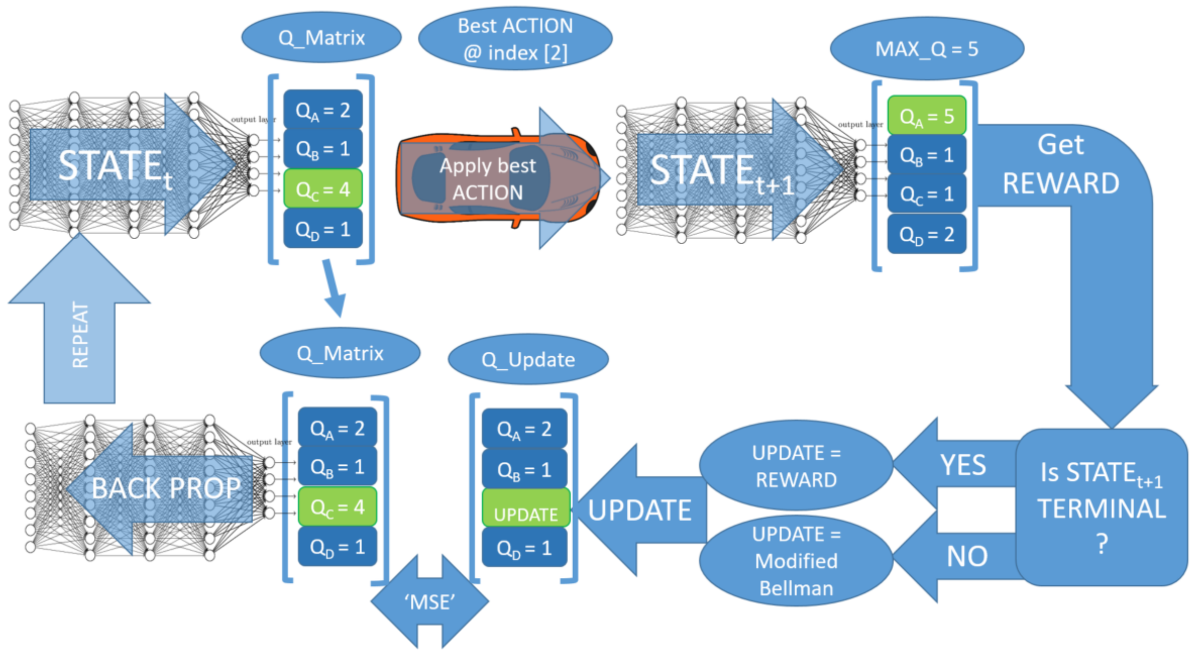


Figure 25: Flowchart showing a DQN cycle.

Another necessary component of DQN is the experience replays. Experience replays are similar to batch optimisation in supervised learning. The buffer is constantly updated with past experiences and in each loop the training is done on a randomly selected batch from a buffer.

The initial data in a buffer can be random, but it is possible to fill it with data from a human controller. The method that is mostly used in this project is accumulation of random data before batches start being selected from a buffer.

The advantages of DQN are mostly related to its ability to learn from its own experience collecting rewards from the environment. It is a technique that has been explored in self-driving cars and neural networks that learn to complete computer games.
In a sterile laboratory environment of a computer game they are highly effective. However, even though DQN can function in complex environments with lots of variables, their functionality and usefulness in the real world, outside of a simulated environment, is still a topic of ongoing research. However, as it is a fairly new technology, it is possible to see significant improvements in the future.

2.7 Other relevant material

2.7.1 Notable behaviours in Nature

Nature is rich with the variety of life. Each species, be it plants, animals, insects or some more simple life forms have behaviours that can influence computer science and many other research fields apart from ones that study Nature.

When it comes to communication, different creatures can report information to each other through the use of gesture, sound, touch, smell or colour.

Social insects's hives represent one huge body composed out of smaller entities. They are very efficient in foraging. In order to forage successfully these complex systems require a way of communicating to each other the locations of food.

Bees have a specific "dance" to indicate the distance and other data related to the location of food. Ants, however, leave scent trails – pheromone paths between a food source and the nest. The more workers return through the pass, the more the scent is reinforced and the more attractive it is to unoccupied ants.

Figure 26: Ants following a pheromone trail.

The scent naturally decays over time, dissipating into the atmosphere until it ceases to exist and ants are no longer recruited to visit the former food site. Some consequences of this method of path indication are that:

- Ants will choose the shorter of two paths of unequal length if the paths are presented simultaneously.
- After ants have chosen a path they are unable to switch to a new path, even if the new path is shorter.
- Ants will all choose one path in preference to half of them travelling the second path, even if the paths are of equal length.

These rules can be easily translated into computer logic. An example of that is the Ant Colony Optimisation algorithm.

Creatures only develop communication regarding something that requires reporting, something that is vital to their existence as a single individual or a group.

2.7.2 Robust – first computing

Robust - first computing is a way of computing where the programs are being designed as living organisms rather than deterministic machines. There are many controversial opinions about such systems, but with the development of computers, they possess more adaptability and security necessary to handle bigger challenges.

Serial determinism is a name of a traditional approach to computing. Machines are designed to perform series of functions step-by-step (serial) and the results of each step are absolutely determined at a state where the step began (deterministic).

Machines using serial determinism are remarkably easy to deal with and they have had a huge success in the marketplace, but they have several downsides. They scale poorly, have poor security and are unable to adapt to unpredictable changes in the system. There are other ways to compute, which probably haven’t received enough attention, and they are the techniques used in robust-first computing.

Viewed as a kind of computer, it is notable how different a living organism is compared to a serial deterministic machine. Deterministic machines are 100% completely repeatable – from the same inputs will come the exact same outputs — while living organisms rarely do anything the exact same way twice. Deterministic machines will crash, seize-up, or otherwise misbehave when anything goes wrong inside them; living organisms, by contrast, can suffer grievous injury and yet survive, handle the immediate situation, and get away long enough to heal up and live on.

Robust computing is an example of how the living systems can have higher survivability than Deterministic machines. Robust agents as a part of an entertainment software can have a unique response to an unpredictable input from a player while deterministic ones may fail to respond at all or provide one of the default and expected responses.

3 OBJECTIVES AND METHODS

3.1 Part One

The research was conducted in two different parts. The first part was a comprehensive analysis of the literature to determine the key factors relating to A-Life agents. The second part was planning the structure of the software and its development.

3.1.1 Keyword Search

Identifying literature for analysis began with suggestions from the advisers and further continued with the keyword search using combinations of search terms shown in table 2. Searches were repeated in a number of scientific databases and the web.

Table 2 : Search terms.

A-Life	Artificial creatures	A-Life agents
Computer learning	Neural networks	Perceptrons
Evolution	Evolving creatures	Evolutionary algorithms
Computer games AI	A-Life games	Life simulation

3.1.2 In-Reference Search

The in-reference part of the search consisted in scanning through the reference lists and reports from the already identified literature. The abstracts of any papers of potential relevance were read, and if appropriate, were selected for inclusion.

3.1.3 Literature Selection

Each publication identified in section 3.1 was studied with the purpose of finding the material related to the implementation of A-Life systems in computing.

3.1.4 Play Testing

Some of the found examples of A-Life were computer games or programs that required testing in order to understand their functionality. At this stage all the necessary testing was done and the observations were documented.

3.2 Part Two

The objective of part two was the development of an entertainment software featuring A-life agents. The software needed to include the A-life agents as one of its key elements, allowing the user to observe their behaviour.

Further on, the interaction between A-life agents and the user had to be created, making it possible to manipulate the agents, make decisions about their life cycle and evolution. A certain level of control over the agents needed to be established.

The final objective was to create a possibility for the agents to evolve speech at a certain level, be it through the interaction with the player or with other agents.

3.2.1 Planning the application

Taking into account the existing cases, the “brainstorming” technique was applied in order to create an outline of the design of the application. The key features and design elements were identified and analysed. Several early decisions about the software architecture were made. The choice of the learning and evolutionary algorithms was done relying on the conducted research.

3.2.2 Tool analysis

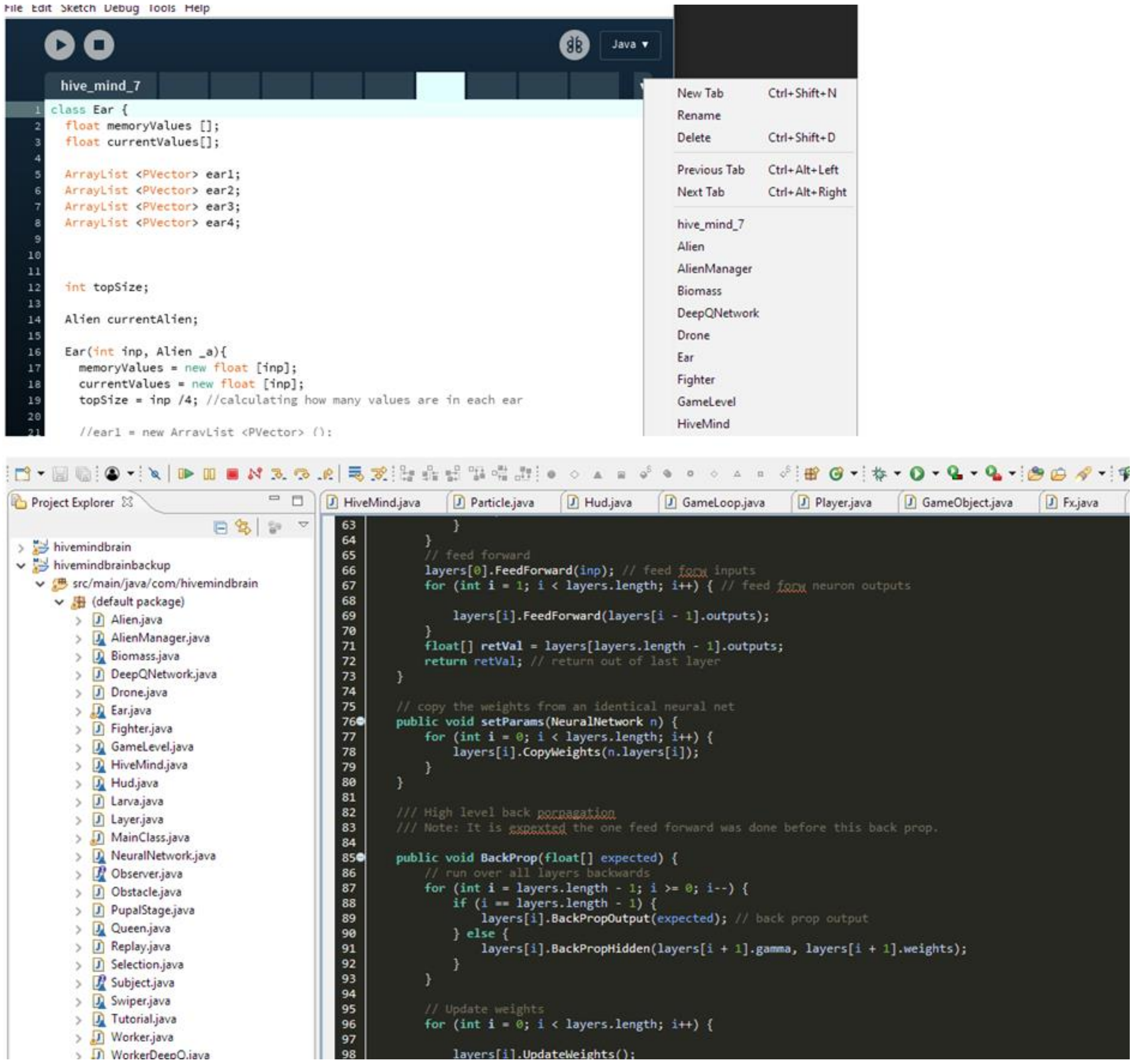
Taking into account the design of the application and considering the choice of the algorithms, it was necessary to choose a tool for implementation. The list of potential tools was created as seen in table 3.

The tools were analysed according to the following criteria: viability when working with sound and simplicity of working with graphical elements with geometrical movement.

After analysis of given tools, it was decided to stick to the tool that I was the most familiar with in order to avoid unnecessary issues that may arise due to lack of experience with a certain platform. Processing is the tool taught in the UC and is the one I have the most experience with, besides it has a large online community which can offer support.

During the development of the project, the environment had to be switched from the native Processing software to Eclipse. The main reason behind that was the lack of Maven support in Processing. It is possible to avoid this problem by manually adding all .jar library files to a Processing directory, however, with large libraries like nd4j or dl4j it is an extremely tedious process that can cause errors that would be difficult to track, it also prevent the library from updating itself. An additional reason for the change was the fact that Eclipse offers more tools for working with Java code and simplifies managing multiple classes, while the Processing application does not contain those functions. Processing library was still used, but within Eclipse environment.

A visual comparison of Processing 3.3 and Eclipse Oxygen can be seen in Figure 27. Processing (above) has lots of tabs that cannot be closed and in a large project the names of the tabs can no longer be displayed. In order to find a required tab it is necessary to open a menu and select a class from it. Eclipse (below) is superior in terms of offering an option of closing unneeded tabs, tree based file organisation and opening a new tab without requiring opening any additional menus.



```
class Ear {
  float memoryValues [];
  float currentValues[];

  ArrayList <PVector> ear1;
  ArrayList <PVector> ear2;
  ArrayList <PVector> ear3;
  ArrayList <PVector> ear4;



  int topSize;

  Alien currentAlien;

  Ear(int inp, Alien _a){
    memoryValues = new float [inp];
    currentValues = new float [inp];
    topSize = inp /4; //calculating how many values are in each ear

    //ear1 = new ArrayList <PVector> ();
```

```
                }
            }
            // feed forward
            layers[0].FeedForward(inp); // feed forw inputs
            for (int i = 1; i < layers.length; i++) { // feed forw neuron outputs

                layers[i].FeedForward(layers[i - 1].outputs);
            }
            float[] retVal = layers[layers.length - 1].outputs;
            return retVal; // return out of last layer
        }

        // copy the weights from an identical neural net
        public void setParams(NeuralNetwork n) {
            for (int i = 0; i < layers.length; i++) {
                layers[i].CopyWeights(n.layers[i]);
            }
        }

        /// High level back porpagation
        /// Note: It is expexted the one feed forward was done before this back prop.

        public void BackProp(float[] expected) {
            // run over all layers backwards
            for (int i = layers.length - 1; i >= 0; i--) {
                if (i == layers.length - 1) {
                    layers[i].BackPropOutput(expected); // back prop output
                } else {
                    layers[i].BackPropHidden(layers[i + 1].gamma, layers[i + 1].weights);
                }
            }

            // Update weights
            for (int i = 0; i < layers.length; i++) {

                layers[i].UpdateWeights();
```

Figure 27: Eclipse (below) and Processing (above) interface comparison.

Table 3: Platforms.

Software name/ Library	Programming Language	Observations
NetBeans / Eclipse	Java	While Java without using any specific tools to facilitate the development can be tedious to program, it offers a high level of control and freedom.
Microsoft Visual Studio	C++	While C++ without using any specific tools to facilitate the development can be tedious to program, it offers a high level of control and freedom. Faster than Java.
Processing	Java (Processing)	As Processing is a tool taught in the UC, I may hope for a lot of help and support in the development. Large online community offers additional support.
Microsoft Visual Studio/ Cinder	C++ (Cinder)	A faster counterpart of Processing. Large community and lots of libraries.
Microsoft Visual Studio/ OpenFrameworks	C++ (Open Frameworks)	Similar to Cinder, but larger community.
Unreal Engine	C++	Offers the high-quality rendering and physics support. Allows focussing on the main features rather than “reinvent the wheel” by programming the geometrical movement and other details.
Unity	C#	Allows focussing on the main features rather than “reinvent the wheel” by programming the geometrical movement and other details.

3.2.3 Additional Literature Analysis

In order to implement the practical part of this project, it was necessary to collect data related to the implementation of some functionalities of this project. First, the primary functions that had to be implemented were set. Secondly, the key words associated to each of the functions were searched in various scientific databases and search engines. The key words and functions are listed in Table 4.

Table 4: Key words used for search related to functions.

Functions	Key words
To understand audio input from the user and reacting to it	Sound recognition, audio prediction, speech recognition, speech prediction, Audio spectrum, distinguishing audio input, Processing sound libraries, minim
To teach agents to respond to audio input using user interaction	Reinforcement learning, Q-Learning, Deep Q-Learning, Reward based machine learning, learning based on experience, learning based on user input
Interaction between the agents using sounds	Deep Q-Learning, multiple agents, multiple neural networks, sensors for neural networks, NL4j, Reinforcement learning, fast deep learning
Interaction with simulated environment using sounds	Evolutionary algorithms, Generic algorithm, Deep Neural Network, mutation, fitness function

Additional study was performed with help of the book “The Handbook of Brain Theory and Neural Networks” by Michael A. Arbib. The beginning of the book contains a brief introduction to the history of neural networks and neural studies, as well as pointers to the articles and authors that contributed to machine learning. There are also details related to the implementation of neural networks and some useful algorithms. The rest of the book is a compilation of articles by different authors that contributed to different areas of machine learning. This book helped with the literature search, related to technical aspects of this project such as types and implementation of ANN.

3.2.4 Practical work

Practical work was done in accord with the literature analysis of subsection 3.2.3. While being originally implemented in accord with the plan, described in section 4.2, work had to deviate from it in order to achieve its goals in the most straightforward fashion. Chapter 5 contains a detailed chronological description of the work process.

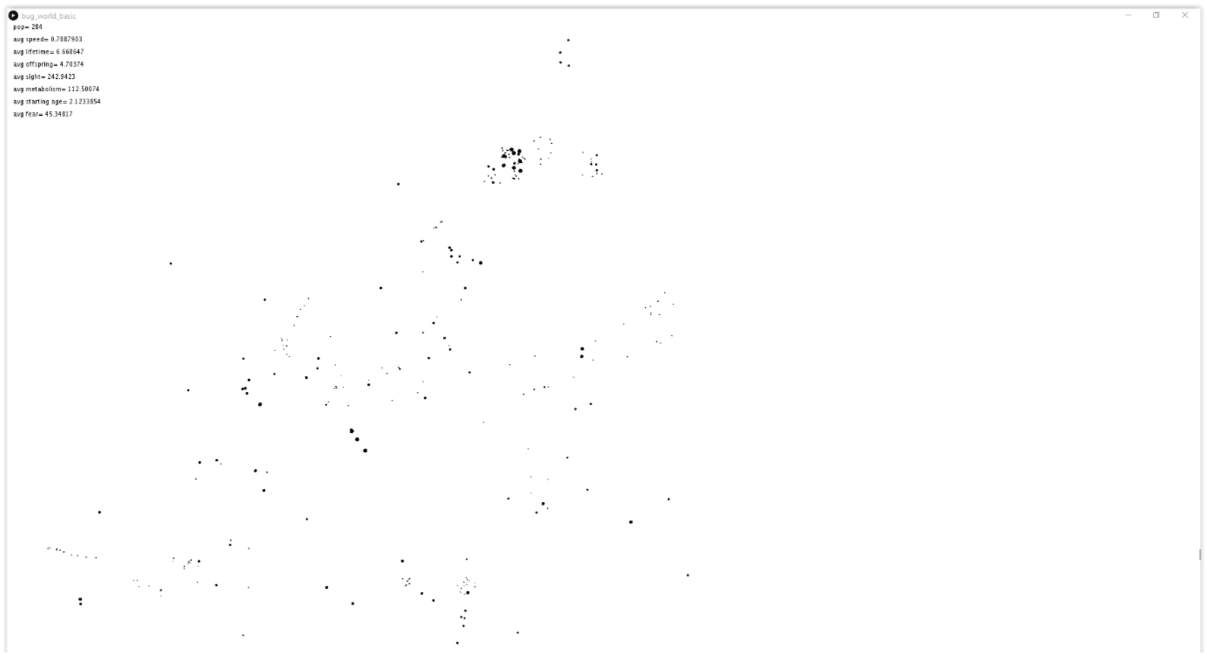
During the practical work, discovered algorithms were applied to the project. Practical work started with some minor experiments with the purpose of familiarisation with the algorithms and techniques. Later, the main project was started.

Due to various discoveries during the development, the final project had to be different from the one described during the preliminary work in section 4.2. The main reason for it being that interaction described in section 4.2 could not provide any entertainment-directed mechanics.

4 PRELIMINARY WORK

4.1 Bug World

Bug world was part of the preliminary work undertaken. It consisted of a virtual ecosystem developed in preparation for this project using Processing.



```
pop= 284
avg speed= 0.7887903
avg lifetime= 6.668647
avg offspring= 4.70374
avg sight= 242.9423
avg metabolism= 112.50074
avg starting age= 2.1233854
avg fear= 45.34817
```

Figure 28: A screenshot from the Bug World.

4.1.1 Demonstration

The following link contains the demonstration video of the Bug World application: https://vimeo.com/252213430

4.1.2 Description

In this simulation, a user can observe a simple ecosystem of cannibalistic creatures - bugs. The bugs are represented as a number of black dots that start their life cycle at a very small size. They move pointlessly controlled by the need for feeding and their fear.

The world in which they exist is a 2D toroidal or infinite world (it is possible to switch between the two). It does not contain any other elements apart from bugs and can be observed from the top.

This model is very far from the realistic biology, but realism was not my goal. With this project, I aimed to create a virtual ecosystem with fictional rules for the creatures that do not reference any real world insects, even though they are called bugs. I wanted to experiment with the capabilities of evolutionary algorithms and test how the behaviour of very simple creatures would be influenced by evolution. What inspired me to do so, was the book Biological Bits by Alan Dorin.

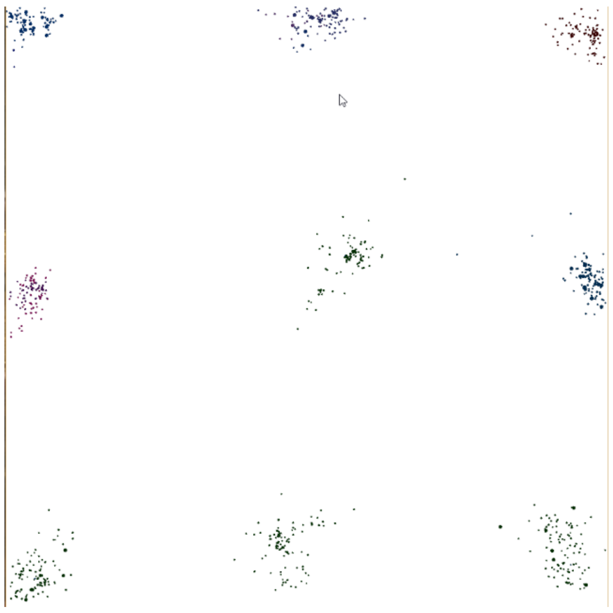


Figure 29: Bug World after evolving for 4 hours. Note the different coloured species.

Each bug has a need to feed. The hunger meter of each bug is defined by its metabolism gene. Depending on the metabolism rating each agent would eventually get hungry and decide to chase another. As the world does not contain any other agents apart from bugs, they feed on each other. Each feeding increases the size of a bug. When they reach a certain size they reproduce and die simultaneously. They can die of hunger fairly quickly and need to chase their prey and feed very often.

Each bug has a number of statistics that can be inherited by their offspring - initial size (size at which they start when they are born), metabolism (how quickly they get hungry and die of hunger), speed (how fast they move), lifetime (how much they need to eat before they reproduce and die), offspring number, fear (how likely they are to run away from the bigger bugs), vision (how far they can see other bugs), colour (a cosmetic feature to observe the mutation). Each one of these statistics can mutate into a new-born individual. Mutations can be favourable (bigger size) or not so (lower speed).

Each of the statistics is essential for the survival of a bug:

- Bigger initial size grants an advantage in hunting other bugs of the same age as a bug can only consume a prey if it is smaller than itself, this will ensure the survival of the bug during its young age.
- Changes in the metabolism rating can slow down the hunger meter which will result in a bug having more time for hunting before dying of hunger.
- High speed is essential for those who want to run away from predators or be good hunters.
- Longer lifetime is actually bad and means that the bug might need to consume more bugs in order to finally reproduce which can undermine the reproduction chance.
- Higher offspring number helps the survival of the species as it allows the bug to burst into a bigger number of small bugs.
- High fear rating ensures the survival of an individual as it will probably start running away from the predators more often.
- High vision allows seeing both predators and prey at a higher range.
- Colour does not offer any advantage, apart from being an indicator to a user.

So, the bigger bugs chase smaller ones and grow until they burst and reproduce. Then their offspring start consuming each other and move around. On birth, a bug inherits all the statistics of a parent and has a small chance that one of its statistics is going to mutate. Like this, evolution is achieved.

4.1.3 Observations

It was quickly proven to me that the bugs with bigger initial size have a strong advantage over the others and are more likely to survive. Isolated bugs die soon if their speed and vision are low, bugs existing in tight clusters reproduce at a very quick rate and reach overpopulation easily.

In the previous section, a number of statistics were mentioned. However, when their advantages were discussed, they mostly kept in account only one individual and its survival. An interesting thing was that a behaviour visually resembling flocking can be observed when a bigger bug would appear close to the cluster of very tiny bugs - they would form a flock-like group when running away from the big individual.
After careful observation of the behaviour of the bugs as a species, it became obvious to me that rather than profiting from the individualistic survival techniques, bugs tended to engage in a flocking mechanism, forming a colony that relied on each individual being as a part of the mass organism of a group.

After leaving the program running for more than 3 hours I could observe different "societies" being created. Most of them were bugs with high initial size, short lifetime and high speed which provided everyone with a chance to reproduce. Their low vision and flocking reminded me of the behaviour of the fish in the murky water, when they are forced to engage in schooling due to the low vision. In the case of the Bug World, the low vision was the result of many generations of evolution and it was the behaviour which resulted in the high fitness of the individuals that engaged in it.

Some other groups had only one very large individual at a time, a "queen", surrounded by the tremendous amount of slow and blind offspring. The “queen” would constantly grow until reaching a giant size and then burst to leave a large amount of offspring, then the cycle would repeat. This kind of behaviour could be related to the survival techniques found in ants and other insects that profit from hauling. In this case, however, the “ants” didn’t need to bring food to the “queen”, because they themselves are food, therefore they evolved to be slow and blind so they would be forced to stay closer to the “queen”.

4.1.4 Conclusions

It was to my surprise, that I was able to observe various sorts of species being formed. This was higher than my expectations at the time and I remained pleased with the result.
The fact that the bugs ended up profiting from the group behaviour was a curious discovery that deserved further exploration in future work.

The simplicity of the Bug World is something that forces it to remain an experiment. Similar projects have been done and accomplished at a higher degree with greater complexity (e.g. Polyworld) and they offer many more scenarios to study and to experiment with.

In the course of this test, I understood how A-Life agents can be created using a computational approach.

4.2 Hive Mind

The following subsections describe the progress which has been done in creating the concept of the “Hive Mind” which is the name for the application developed as one of the objectives of this thesis.

The requirements listed in this section are for the early stage of the development. It is possible that throughout the study a lot of the mechanics described here will change or be removed completely.

4.2.1 Theme/Setting/Genre

The term hive mind is a sci-fi idea of a universal mind that through telekinetic powers maintains absolute control over species. Like this, the entire species possess one mind.

In this toy, the player takes control over a hive mind of an alien species. Because the species of the characters are alien to us, the story is set in space. It's difficult to characterise this project as a game of a certain genre, apart from a sandbox or a puzzle, it is essentially a toy.

4.2.2 Core mechanics brief

- To produce the sound using keyboard or an external sound synthesiser/musical instrument
- To train the aliens using sound
- To breed and evolve aliens
- To control the alien evolution
- To accomplish small tasks using the trained aliens
- To use real-life sheep herding mechanism as a model of this mechanics

4.2.3 Project description (brief)

The player takes control over the hive mind of an alien species. Using their own language, the player has to communicate with them and teach them to respond to various sounds.
The action takes place on a spaceship and, in order to progress, the player needs to take over the vessel by training their aliens and issuing the commands to them.

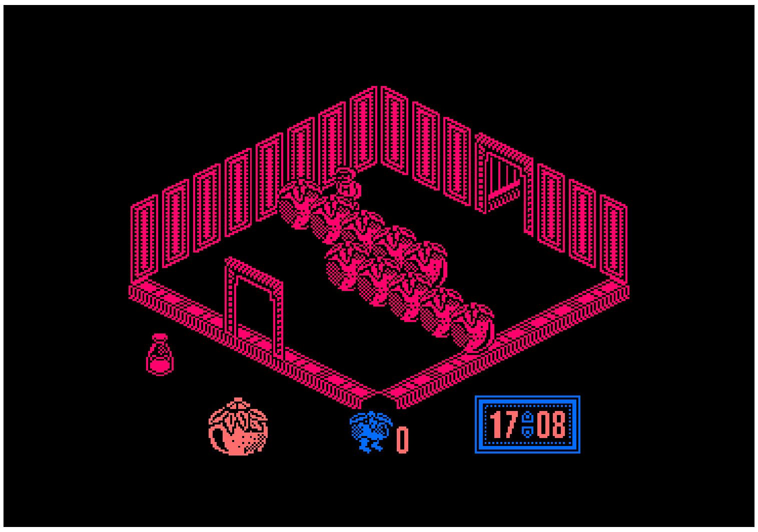


Figure 30: Attack of the Killer Tomatoes game. A screenshot.

The view of the toy is planned to be isometric and visually similar to "Attack of the Killer Tomatoes" game from 1986 (Fatman, Dobbin, Stuart Ruecroft)

4.2.4 Gameplay

Warrior mechanics

The human is master of many animals. Dogs are a very good example of such an animal and one of the fascinating mechanics in which they are involved is herding. Sheep herding is done in order to keep sheep from running astray and also in order to protect them from natural predators such as foxes.

While observing sheep herding, first of all, we see a human who with a series of complex whistles issues commands to a dog. Subsequently, the dog responds to the whistles by moving left, right, forward, backwards, lying down, speeding up, slowing down, etc. The mechanics are based on the fact that the sheep wish to stay away from the dog as it in many ways resembles their natural predators. By controlling the dog, the human is not only able to steer the herd but can also perform more complex tasks such as sorting the males from the females or making the entire herd go in the desired direction. More control is usually achieved with an additional dog.

Using the herding mechanism, I was exploring the fact that perhaps instead of commanding all of the aliens at once, sound commands of the hive mind reach only specific individuals called "warriors".

The player can select any alien and make them evolve into a warrior. The warriors are bigger than the regular aliens (workers) and can perform some additional actions. The sound commands of the player are only heard by the warriors. Using these commands, it is possible to train the warriors so they can be controlled in a more or less precise way which will allow the player to use them for steering the aliens around.

An advantage of these mechanics is that it will make the application much cleaner in the way that not everyone will respond to player’s commands which can cause chaos. It will instead create some space for improvement of player's skills by making only the warriors directly controllable.

Another advantage is that the player will be able to control a large number of aliens at once.

The main advantage is the possibility of creating puzzles based on learning. An example of this is where for the completion of the level we may need to lead the aliens through a narrow passage, it is therefore important to keep them together. Having two warriors on either side might help, but each will require learning different commands in order to make this work. We might need two warriors where we teach one warrior to respond to the same commands in reverse. If the passage would as well have lasers or any other type of danger on either side, the player would need to steer each warrior out of it at the same time which would provide a nice challenge.

Hive mind

The hive mind is unique and represents the entire species. It is intelligent, knows everything and is aware of all aliens. The player is the hive mind and has a strong connection with the queen – it is possible to issue direct commands. But controlling the queen is not enough since she is slow and vulnerable and cannot move. In order to be successful, the player needs to take control over the rest of the aliens (workers). In order to accomplish that, the player has to use language to talk to them. Language needs to be alien. The entire keyboard offers a variety of sounds and each key produces a different synthesized insect-like noise.

Another considered possibility for implementing the sound controls is using the external sound input through the mic, maybe through a separate analogical mini-keyboard with sounds. This possibility however, could lead to unwanted complications with aliens understanding the commands, so first-hand the more basic computer keyboard variant will be implemented.
There are two sounds that the player is offered to input at the very beginning. These sounds will remain locked throughout the play and the aliens will be pre-trained to perceive them: YES and NO.

Using the YES and NO, hive mind can train warriors to move around according to a player's wish using the reinforcement learning.

Each sound produced by the hive mind is heard by all warriors. The idea is to use warriors strategically to lead aliens into the areas where the goals are.

Alien types:

There are 4 types of aliens:

1. Regular aliens (workers)
2. Queen
3. Breeders
4. Warriors

Workers

They are small and mindless creatures that may be present in large amounts. They are the most numerous of all aliens. Each worker can move up, down, left, right, pick up an object, put down an object and produce a noise. They cannot see but they can hear very well. They do not hear the commands of the player. They tend to stay relatively close to each other, using the sounds to orient themselves. They also keep a distance from the warriors. Each of them has personal statistics like, for example, speed and lifetime - they represent the genotype of this individual.

Queen

The Queen cannot move through the level. It can lay eggs if fertilised by a breeder. The Queen also serves as an objective point where workers need to bring objects.

Breeders

The player can select any worker or warrior to evolve into a breeder. When they become breeders, they perform the hard-coded breeding. After they breed, they die. Subsequently, the queen will lay eggs that will hatch into the workers that share the genotype of the breeder with the small possibility of mutation.

Warriors

Warriors are the main feature of this toy. The player can select any worker to evolve into a warrior. When a worker becomes a warrior it means that it starts being able to hear the player.

Warriors can be taught using the hard-coded YES and NO commands to learn what other sounds mean. When the player produces a sound, the warriors try to interpret it and to respond to it in a certain way. The player then says YES or NO according to whether they guessed the action or not. Through this process of training they learn voice commands. Example:

- Player produces "beep" sound and they want a warrior to respond to it by going right
- Warrior responds with going left
- The player produces "NO" sound
- Warrior remembers that “beep” means not going left
- The player produces "beep" sound
- Warrior responds with going right
- The player introduces "YES" sound
- Warrior remembers that "beep" means right and now will always try to go right when the player produces that sound

Warriors are similar to herding dogs. The sheep (workers) try their best to keep the distance from the warriors. By positioning the warriors strategically, a player can lead workers to the current level objective.

4.2.5 An example scenario

The program starts and the player is offered to introduce the sounds that are associated with YES and NO. After that the player can see the map as shown in Figure 25 (this is only a schematic representation of the map).

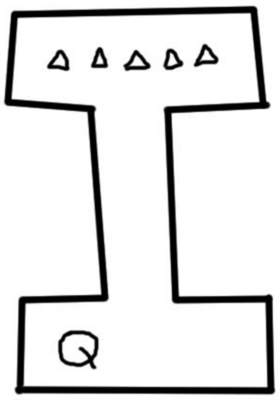


Figure 31: Queen and the objectives (triangles).

Q is the queen and the little triangles are objects that need to be brought to her in order to complete the level. The Q doesn't move and stays in place. The player can click the Q and order her to lay eggs. The Q is then going to lay eggs and many workers will hatch (Figure 26).

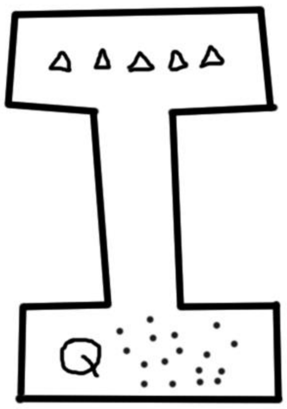


Figure 32: Queen, workers (dots) and the objectives (triangles).

Each worker is a small computer that is going to listen to the sounds that other workers make. Using the sounds to orient themselves they will try to avoid the collision with others while trying to stay closer. These mechanics are similar to flocking behaviour described in Craig Reynolds’s “Flocks, Herds, and Schools: A Distributed Behavioral Model”.

Each worker has a lifetime countdown and will inevitably die when the time comes. They can move around, pick up objects and put them down when they touch them.

Now, in order to actually start doing something, the player needs to create a warrior. For that they need to select one of the aliens so it evolves. Warrior is marked as W in Figure 33.

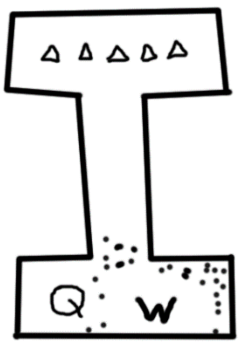


Figure 33: Queen, Warrior, workers (dots) and the objectives (triangles).

As one can see, all the workers keep some distance from the W. If we could control the W we could probably push some workers through the narrow passage so they would collect the little triangles and then we could "herd" them back in order to deliver the triangles to the Q.

In order to be under our control the W needs to be trained. We can teach the W to respond to certain sounds with movement. The training is accomplished using the sound keyboard and the YES and NO sounds that it already knows. It is done with a form of reinforcement learning.

When we produce any sound apart from the YES and NO the W is forced to respond in some way. If it does, we can tell it if it guessed using the YES and NO. Weights in W's neurons will be adjusted and then we will be able to control it more precisely.

In Figure 34 the W that is under our complete control was able to lead the rest of the aliens through the narrow passage towards the objectives.

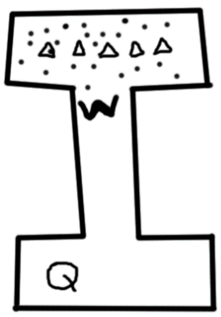


Figure 34: Warrior brings workers (dots) to the objectives (triangles).

It is possible to create various level layouts with different puzzles using these mechanics. This, of course, was a basic example but we can design levels that require not only more warriors but also challenge a player's reaction.

In the following layout, we have a room with moving deadly lasers (Figure 35). If the aliens touch them - they die. For this scenario the player would need to have two W’s that would sandwich the workers from both sides, making them move in a thin line. The problem here is that the W’s themselves have to avoid the lasers. This requires quick responses from the player.

Because there are two W’s, the player either needs to teach them polyphonic sounds or to train one of them to respond to the commands in reverse so one sound could make both of them move either closer to each other or away from each other and closer to the outer sides of the map. Shepherds use multiple dogs in sheep herding so using multiple warriors is definitely one of the ways of solving the problem.

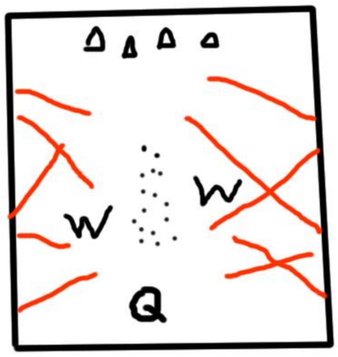


Figure 35: A possible play scenario with deadly lasers.

Some scenarios might involve aliens having to move really fast, or not having enough time to complete the level and therefore dying because their lifetime is over. In such situations, breeders come in. Creating a breeder from the fastest alien may let us evolve individuals with higher speed which would let us complete the level. Some hard-coded mechanics could prevent the evolution from scaling - for example limiting the number of eggs the queen can produce during the current mission.

4.2.6 Further possibilities of the mechanics

Further possibilities described here are just ideas and they do not represent the functional requirements for the final prototype. Some of these possibilities might be explored if necessary, but for now, they are just options that do not relate themselves to the goals of this thesis.

As I understand, the mechanics described in the above sections offer a number of possibilities for adding more complexity to it. For example, some levels might require the aliens to have certain skills such as shooting projectiles, melting walls or forming a shell that makes them invincible to lasers. A consumable object granting one of these skills would be located in an area right next to a dangerous item (possibly a deadly virus). The player would have to herd the workers with great caution in order to pick up the "good" item without touching a deadly item. The worker who picked up the item would then gain an ability. The player would then have to lead that worker back to the Q and make it into a breeder so it can breed with the Q and produce the offspring that share the ability.

Another idea is that it would be possible to add killable humans or other beings and enemies that will have a basic AI and try to shoot the aliens.

A further way of adding depth and difficulty to the toy would be adding the possibility of deadly viruses that spread to the nearest aliens.

4.2.7 Activity Graph

A UML diagram in figure 36 explains how the transition through various stages of the toy is accomplished.

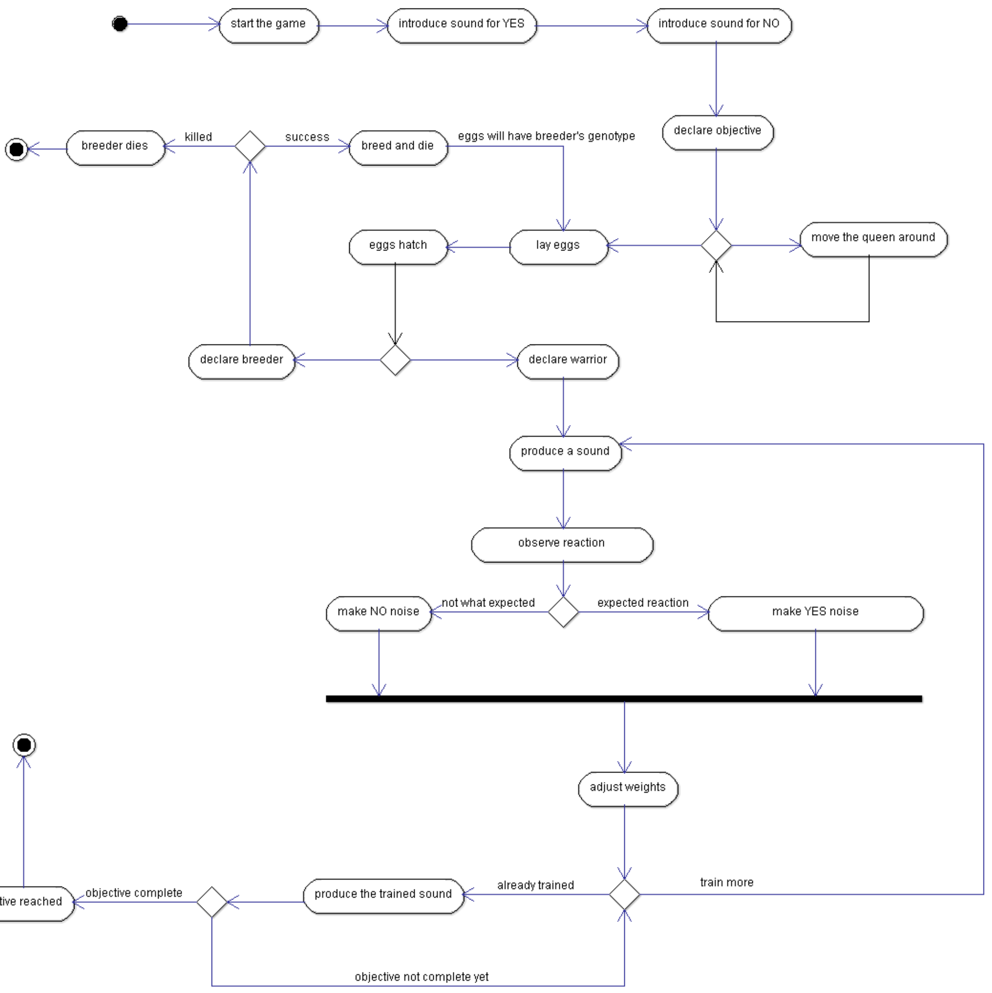


Figure 36: Activity graph of Hive Mind.

5 PRACTICAL WORK

This chapter describes the practical work done throughout the second semester which involved multiple experiments in creating ANN using Processing and Java (Eclipse). The code written during this phase of the research will be discussed and described.

The major focus of this chapter is the analysis of the work done so far in chronological order towards the development of the Hive Mind application and listing of the decisions that had to be made in order to bring the application to its final form.

5.1 Exploring ANN

This section describes the attempts to create ANN in Processing ranging from simple perceptrons to more complex DNN. Several experimental applications are shown and analysed.

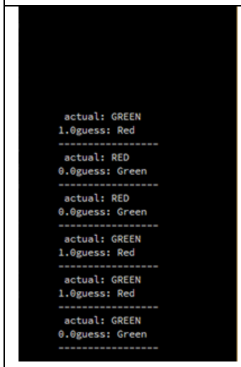
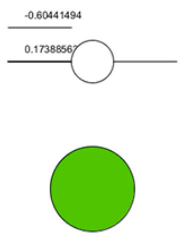
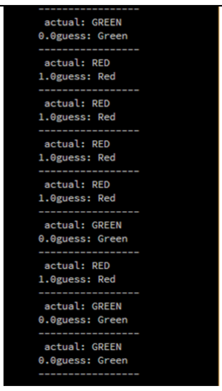
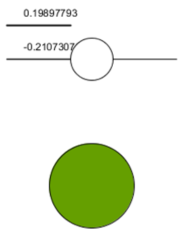
The process described here was essential in order to produce an ANN framework/library that would serve as a basis for the final application.

5.1.1 Basic perceptron colour guess

To grasp a full understanding of ANNs I started by programming a simple sketch in Processing which would utilise a single perceptron to learn to associate a name of a colour with an RGB value. I only used two colours – red and green, because of the binary nature of a single perceptron. In this scenario, the correct answer was backpropagated after every guess.

The circle would change its colour on each click and the ANN would print a guess, after that the guess would be compared with an actual colour. After a couple of clicks, the frequency of the correct answer would improve as weights would get adjusted quickly due to the simplicity of a perceptron. Table 5 shows screenshots of the untrained perceptron output vs trained perceptron, along with a simplistic schematic of a perceptron and its weights.

Table 5: Comparison of trained and untrained outputs of Colour Guess application.

Untrained output	Trained output
 	 

```
 actual: GREEN
1.0guess: Red
-----------------
 actual: RED
0.0guess: Green
-----------------
 actual: RED
0.0guess: Green
-----------------
 actual: GREEN
1.0guess: Red
-----------------
 actual: GREEN
1.0guess: Red
-----------------
 actual: GREEN
0.0guess: Green
-----------------
```

```
-----------------
 actual: GREEN
0.0guess: Green
-----------------
 actual: RED
1.0guess: Red
-----------------
 actual: RED
1.0guess: Red
-----------------
 actual: RED
1.0guess: Red
-----------------
 actual: RED
1.0guess: Red
-----------------
 actual: GREEN
0.0guess: Green
-----------------
 actual: RED
1.0guess: Red
-----------------
 actual: GREEN
0.0guess: Green
-----------------
 actual: GREEN
0.0guess: Green
-----------------
```

5.1.2 Increasing complexity in deep colour guess

Subsequently, the challenge had to be increased because implementing a simple perceptron would not be able to accomplish the goals of this project. In order to increase the understanding of ANN and make a first attempt in backpropagation algorithms, a DNN with one hidden layer and MSE gradient descent backpropagation was created. This time the application had more colours to guess from, therefore there were 7 outputs instead of a binary output.

The sketch would draw 7 rectangles of different colours, DNN would have to write a correct name underneath each of the rectangles. This time, the DNN was not designed not learn in real time and its training was done beforehand. The table shows the outputs with different numbers of training cycles, ranging from 0 – no backpropagation to 10000 backpropagations. The examples shown in Table 6 can vary depending on the initial setup of weights, the examples that were selected to be put in were chosen through a purely aesthetic analysis, as the ones that do not deviate far from their average. As calculations were done almost instantly, it was easy to experiment with different numbers of cycles. As expected, accuracy increases with the number of cycles.

Table 6: Comparison of Deep Colour Guess outputs.

Output							Cycles	Accuracy	Score
🟥 blue	⬜ magenta	🟩 grey	🟦 cyan	🟨 red	🟦 grey	🟪 green	0	0%	0/7
🟥 red	⬜ green	🟩 yellow	🟦 grey	🟨 green	🟦 blue	🟪 green	10	14%	1/7
🟥 red	⬜ magenta	🟩 grey	🟦 blue	🟨 magenta	🟦 grey	🟪 cyan	50	28%	2/7
🟥 red	⬜ magenta	🟩 yellow	🟦 blue	🟨 yellow	🟦 magenta	🟪 grey	100	42%	3/7
🟥 red	⬜ green	🟩 cyan	🟦 blue	🟨 yellow	🟦 cyan	🟪 magenta	200	71%	5/7
🟥 red	⬜ grey	🟩 green	🟦 blue	🟨 yellow	🟦 cyan	🟪 magenta	500	100%	7/7
🟥 red	⬜ grey	🟩 green	🟦 blue	🟨 yellow	🟦 cyan	🟪 magenta	10000	100%	7/7

5.1.2 "Jumping over ball" example, exploring mutation

Fascinated with the results of previous tests, I decided to design a more interesting challenge. I would like to see a DNN to play a game. I saw some examples of ANN learning to complete retro games such as Mario, play Go (Alpha GO) and even playing DOOM (DL4J), however using an existing game would be a big challenge, which I decided to save for later for in the current situation I needed to focus on the basics.

With that in mind, a simple environment was designed: a 2D world populated with obstacles in the form of ellipses. The creatures that existed in this world were presented as smaller ellipses, each possessing a DNN, a reworked version of the previous deep colour guess example. Each of the creatures would be able to perform one of the simple actions each frame – walk right, walk left, initiate jump and do nothing. Apart from the actions, the creatures could “sense” the distance to the nearest obstacles and were aware of their current Y position in order to distinguish the jumping state.

The objective was to train these creatures to successfully move from the left side of the map all the way to the right by successfully avoiding the obstacles.

A problem arose immediately – how to train an ANN without knowing the correct answer. Of course, I could hard code such things as – when you are next to an obstacle, the correct answer is to jump or when you are on the ground, the correct answer is to walk forward. ANN would not be required for the implementation of this, and it would be required to program unnecessary complexity (more detail on this can be found in subsection 2.6.5). If a programmer can create correct patterns for creatures to follow, then there is no need for ANN. Instead of doing that I could just hard code those actions and watch the creatures solving the problem easily since the beginning. What I really wanted was to observe emerging behaviour, patterns that a machine would come up with on its own.

To increase the complexity of the task, positions of circles would be randomised each time so DNN would never be able to rely on time in order to learn correct actions. It would have to rely on sensors, similar to self-driving cars.

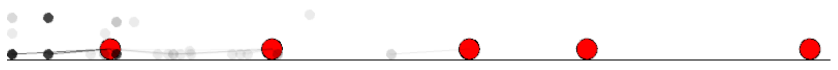
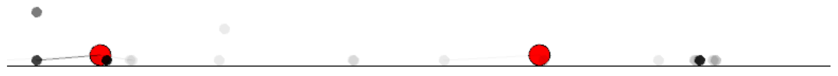
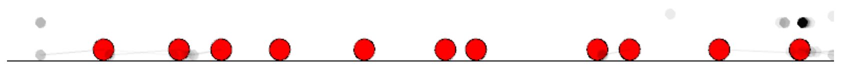
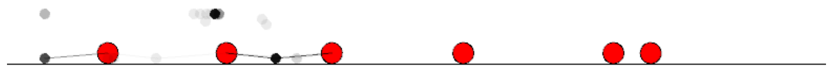
In the current situation, the optimal solution to this problem was removing backpropagation and replacing it with mutation. It would increase the frame rate of the application as agents would not have to backpropagate each of their actions, however, a successful mutation mechanism would require a large population of agents and a fitness function.

I decided to avoid using a fitness function as I like to see it as natural pattern that is based on survival as opposed to an equation, as an alternative, fitness based on multiple functions could be another path to an interesting emerging complexity.

The sketch was playing each generation for 1000 frames before removing all of the agents, solving the crossover and mutation, creating a new generation, randomising the obstacles and playing the next generation. Each generation would be composed of 100 agents. The fitness was calculated taking in account the X position of the agent, increasing along with the X value, as well as number of obstacles which were avoided without touching them.

In Table 7 some screenshots of the application can be seen. Although, the evolution from the 1st to 4th generation is not clearly noticeable as the primary evolving element is animation, screenshots allow to make additional comments about the program.

Table 7: Screenshots of various generations of Jumping over ball.

Generation number	Screenshot of generation
1	
2	
3	
4	

As seen in Table 7 creatures are represented by black semi-transparent circles. The reasons why some of the circles are darker is because that area contains many overlapping individuals. This kind of visualisation allows the observer to see the behaviour which is predominant in a generation.

As a result, it proved possible for the agents to figure out correct jumping times and movement actions after around 20 generations, depending on the number of nodes in a hidden layer, number of inputs and mutation chance.

5.1.3 Increasing complexity through dynamic sensors

The original sketch of jumping over a ball from the previous sub-section lacked good sensors. Improved new sensors would definitely generate more interesting emerging behaviours for the agents. The new sensors needed to be designed in a way, so that an agent could switch between them and choose in which direction to look. It would be interesting to see a behaviour where agents make decisions about their sensors and know in which direction to look in each situation.

A new sensor was designed which was represented in the form of a line which could sense whether something is within a small circle around its tip. Apart from that, each agent had a set of new actions which corresponded to pointing a sensor in each of the four directions. Inputs now included not only the output of sensors, but also a value indicating which direction the sensor is currently pointing at.

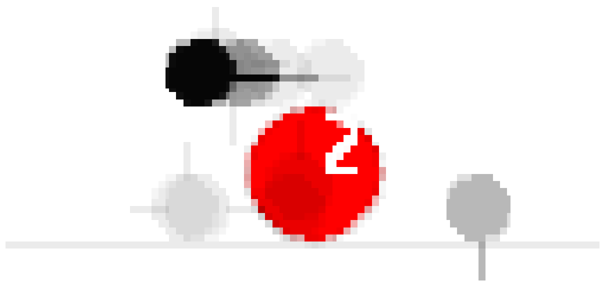


Figure 37: Agents jumping over a ball. Sensors (lines) can be seen.

As expected, the newly implemented dynamic sensors brought more interesting behaviour into the sketch. While the speed of learning did not increase or decrease significantly, the accuracy of the jumps that the agents were doing had increased as now, while an agent is in the air, it could point the sensor downwards in order to scan the area underneath in order to adjust its landing trajectory. Many times agents evolved a behaviour where they were pointing the sensor backwards during the landing in order to land as close as possible to an obstacle that they were jumping over, this kind of behaviour helped them to avoid landing on top of the next obstacle and leave more space for further manoeuvres. A more sophisticated form of this behaviour that could also be observed was a rapid switching of sensors between front and back while landing from a jump which allowed the agents to align themselves nicely between two obstacles.

In conclusion, it was interesting to observe new behaviours being formed as a result of multiple mutations. Individuals were able to solve problems relatively fast which can serve as a proof of successful implementation. It is possible that there are better solutions involving computer learning but the current example worked and resulted in the interesting behavioural patterns. A video of this application can be seen here: https://vimeo.com/277782107

5.2 Development

In this section, the development of the final application was described, featuring the steps that were taken towards reaching the result.

Problems of learning through sound, artificial ear system, ways to avoid implementation of sound-prediction and constant frequency analysis are described here.

The concept of the final project is described in the preliminary work chapter under the title of "Hive Mind". It is outlined as an interactive digital toy where user can interact with various A-Life agents. For a more detailed description, it is advised to consult section 4.2. This entire section covers the progress that was made towards its implementation.

5.2.1 Reinforcement learning

The first thing that needed to be implemented was the user interaction. As described in the concept outline, the interaction is done through sounds in the following order: user produces a sound, agent responds with an action. After that the user must aesthetically evaluate the action by pressing a key or producing a sound corresponding to yes (correct) or no (wrong). A user, who clearly knows the right answer, reporting it to the agent in a simple way was a difficult issue. As a user could not report an actual answer to an agent, they were limited to binary input. And so the first problem that was encountered during the starting phase of implementation was “How an agent can learn without knowing the right answer?” If the agent is guided only by YES and NO user input, how it can learn and adapt its behaviour according to what the user expects from it?

Of course, the answer to this question is given in the previous chapter when the implementation of “Jumping over Ball” is described. Translating it to the current scenario, YES would be a positive fitness value, while NO would be negative. EA are a viable option in many computer learning programs, however, in this scenario something that worked with the effect that was only achievable through backpropagation was needed. One simple agent was required, rather than a group of them (a generation), it also had to learn in real-time, listening to a user and immediately evaluating the state of the world they are in together with their own output.

This sounded a hard enough problem in itself. I would need to look into LSTM networks and how to train a network by giving sounds data overtime. It would not be an optimal solution to train a network where it receives the entire sound data at the same time because it would have too many notes and be computationally impossible. The closest possible solution would be text prediction with RNN, as the architecture of a sound predictor would be roughly the same as what is needed for the sound recognition.

Subsequently, this created more questions. How clunky would such a system be, would it run smoothly in java using the Processing library? These architectures were designed for large datasets and they learn effectively over very long periods of time, which was not suitable for the implementation of a little game where the agents needed to learn in real time. The solution needed to be simple and compact.

5.2.2 Deep Q-Learning

Q-Learning allows agents to learn using a simple binary response of YES or NO. It was based around generating tables of pairs of actions and states which an agent would analyse in order to pick the right action next time.

Because an agent needed to analyse a very complex and detailed world state as an input (entire sound spectrum), Q-learning alone was not enough. ANN was needed in order for it to function as intended. DQN seemed like an optimal solution. It is essentially a combination of an ANN and a Q-Learning table.

A resulting ANN would consist of several Java classes:

- Replay – an object of this class would be able to store a single experience replay with all relevant data about it.
- Neural Network – this class was a re-written and refined version of the old colour guess network that I had written during the process described in the previous section.
- Layer – class that would define a single layer of the network, again, it was re-used from the previous section examples.
- DeepQNetwork – this class describes all additional functions that would manipulate ANN and replays in order to make them function in accordance with DQN flow by saving all replays, feeding them forward and backpropagating in the correct order.

5.2.3 Avoiding use of prediction

As it was said previously, sound prediction implementation is a complex task (mainly for the reason of requiring libraries of learning examples) which required an easier, and a more efficient way in the current situation to solve the sound recognition problem.

One of the answers to this question was using a limited library of extremely basic sounds which would be easily stored in the memory and would not generate the range of infinite sound combinations.

5.2.3 Creating library of sounds

One of the original ideas for this project involved turning the keyboard into a piano so each key would produce a sound which would be clean enough for any of the virtual creatures to interpret.

I looked for insect sounds online in order to use them as a reference and consequently I started creating my own sounds using Audacity. Various foley recordings were made of simple objects, such as paper, plastic bags and a water tap which were later edited to appear like sounds produced by insects.

After mapping the sounds to the keyboard keys, I was able to produce them and the agent could respond to them.

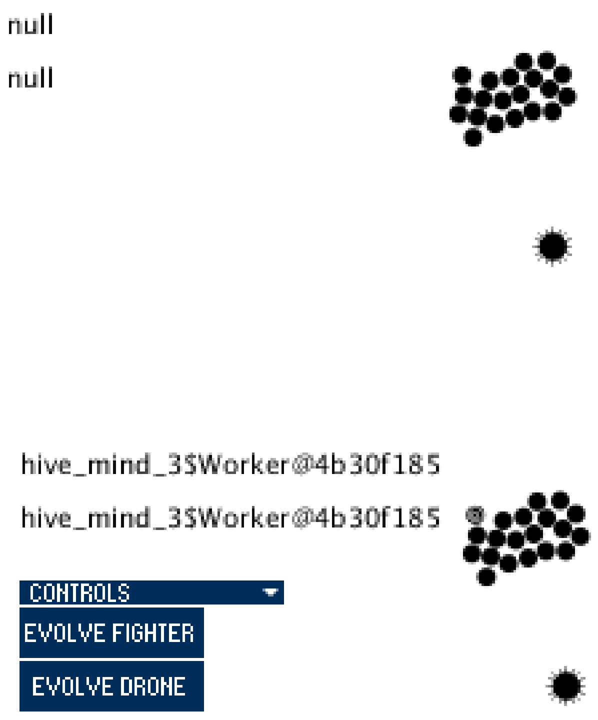
This link contains a video with various sounds that were produced during this stage: https://vimeo.com/277789895

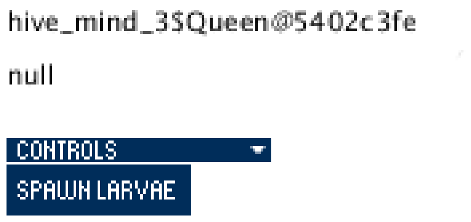
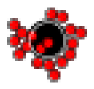
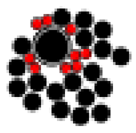
5.2.4 Designing basic mechanics

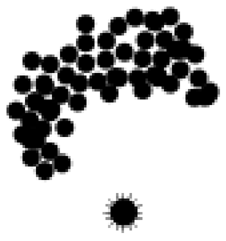
At this point the application contained several agents and interactions between them. There was also the user interface and the possibility to interact with various agents. Table 8 shows the various functions that existed at that point of the application.

Several changes were made to the concept described in the first part of this thesis. One of them was changing the name of the user-controller agent from warrior to fighter. A reason for this change was the fact that both warrior and worker words started with “w”, which could cause confusion in schemes and abbreviations. Therefore, when the word “fighter” is mentioned, it refers to “warrior” from section 4.2.

Table 8: Functions in early version of Hive Mind.

Function	Graphical representation
Possibility to select and deselect an agent.	 null null hive_mind_3$Worker@4b30f185 hive_mind_3$Worker@4b30f185
A visual effect when mouse is hovering over an agent.	

Menu with buttons, showing functions of selected agent.	 hive_mind_3$Queen@5402c3fe null
Queen spawning larvae	
Larvae growing into workers	

Workers evolving into fighters	
Workers avoiding fighter but clustering together	

5.2.5 Distinguishing sound inputs

The most obvious problem that could come to anyone's mind with this model – is whether the agents can perceive the same sound multiple times and remember various bits of composition in order to react to them the same way as before.

And no, they couldn't do it, at least not easily enough. The problem was that each frame of sound would be perceived by an agent as a large array of the audio spectrum which would definitely, in the most cases, be different from the audio spectrum of a previous frame.

Figure 38 represents spectral analysis of a very simple half a second long sound produced by a bird. To humans it sounds like a uniform noise, however even this sound has variation. Every column of lines on the image represents the audio spectrum of each individual milliseconds long frame during which the sound was played. While the variety is minimal, it is still present. In this case it would be possible to train ANN to recognise any frame of it relatively easily, however as the complexity of sounds increases, so does the time it takes for the network to learn them. Ultimately, such techniques as sound prediction will be required to train a network in an effective way.

The way of solving this problem was creating new sound files which would have very slight spectrum variation or even none. In this case it meant getting rid of insect sounds and creating a new batch of sound files where each one of them would be a plain noise without any spectrum variations over time.

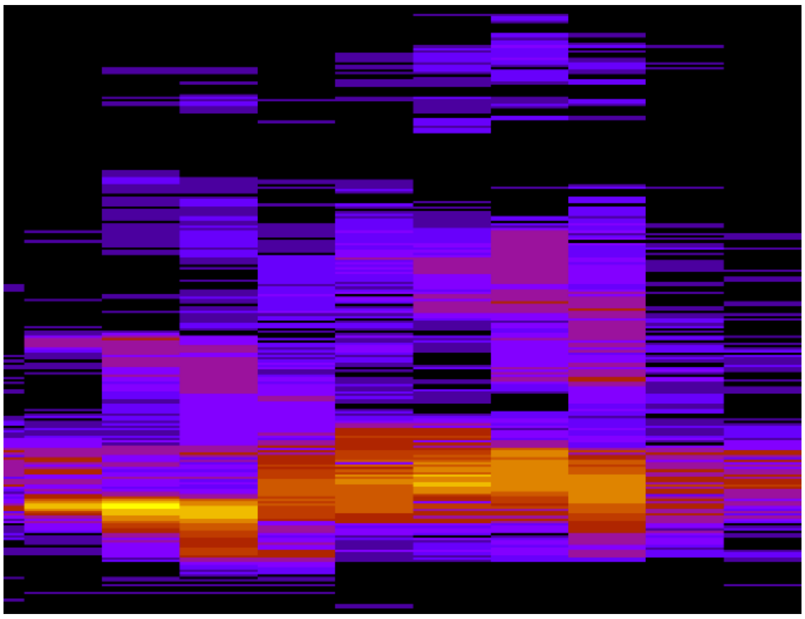


Figure 38: Spectral analysis of a bird sound.

5.2.6 Microphone option and sound levels

For the sake of curiosity, I decided to try and test the learning capabilities of an agent without the use of pre-defined sounds and instead utilising the microphone as an input. After the microphone was enabled as an input, I noticed patterns in the behaviour of the agent that definitely showed me that it was responding to the sound in a similar way as before. Its movement became more erratic, similar to Brownian motion of particles, but whenever a loud sound with a constant spectrum was produced, the agent would switch into a more consistent behaviour.

The reason for erratic movement turned out to be a constant noise produced by the microphone. The agent was constantly responding to the humming of computer fans and random background noises. In order to decrease and possibly eliminate the effect of background noise, the minimum audio level was defined for the agent. The agent would only listen and perceive the sounds when the audio level was higher than a certain value.

This method worked as while the levels were not surpassing the defined value, the agent would remain in the state of rest, without performing any erratic movements. Whenever a loud sound was produced, audio levels would increase, enabling the agent to listen to the input and react to it in the same consistent way.

Later a slider for levels (Figure 39) was added so the value could be changed dynamically by the user, allowing adjustments during the experience itself.

Figure 39: Levels slider.

5.2.7 Creating electronic piano

With the new microphone interaction the agent was listening to real world sounds. While various musical instruments, voice and scratching of the microphone were making an agent be-have in a relatively consistent way, its behaviour was much less accurate than the one that was achieved through digital sounds that came from a keyboard straight into the sound card.

A spectrum of mostly real-world sounds tends to be not as consistent as it might seem in some cases (for example, flute). The previous digital sounds used to be more consistent and were accompanied by an accurate response. It was natural to assume that the reason behind the less accurate behaviour was the inconsistency of the audio spectrums from the real world object sounds.

So what if the sounds that were produced in the real world were generated by electronics and were consistent enough for the agents to perceive them as intended? In order to find an answer to this question, I used an Arduino board, a set of buttons and a miniature speaker to create a prototype of an electronic “piano” that would play sounds similar to the original consistent digital sounds but through a speaker (Figure 40). A short demonstrational video of this “piano” can be seen here: https://vimeo.com/277783441

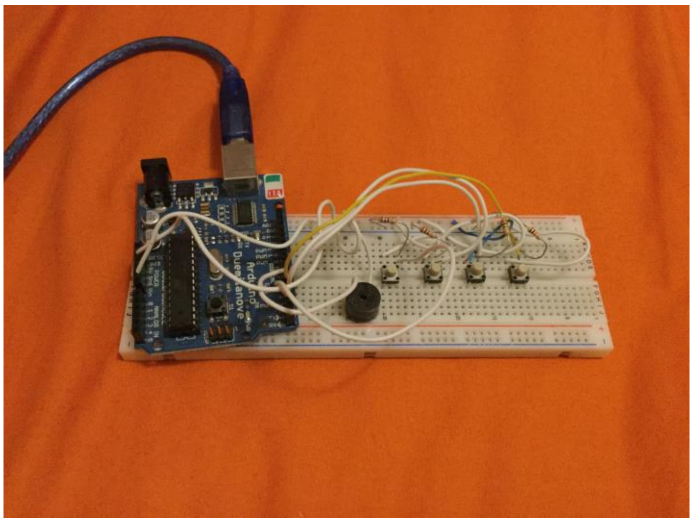


Figure 40: Arduino "piano".

The piano did work and the creatures were responding to the sounds with a more consistent behaviour, but for this behaviour to manifest, the little speaker needed to be right next to the microphone. I assumed that the volume of it was just not enough.

As I kept working on the electronic piano, I decided to try out an option that was neglected previously – synthesisers available in Garage Band app on an iPad.

With a synthesiser, it was important to find one that had settings that would disable all kinds of echo and reverb effects, so the sound would stay pure and consistent. This iPad option turned out to work much better than the Arduino piano as it could achieve higher volumes and also had more choices of sounds.

At this point, the project had to move on towards the next stage, as the major issue with the sound-based learning was solved.

5.2.8 Binary input

As was outlined in the concept, the agent would be taught with YES and NO commands. By issuing these commands it would be possible to teach the agent to respond to sound commands in the correct way.

Table 9: Command log of an agent.

Command	Action	Reward
GO UP	DOWN	NO
GO UP	LEFT	NO
GO UP	UP	YES
GO UP	UP	YES
GO DOWN	UP	NO
GO DOWN	RIGHT	NO
GO DOWN	UP	NO
GO DOWN	DOWN	YES

Table 9 shows a log of commands that were issued to an agent (fighter), its responses and the consecutive rewards. Commands in this example would be represented by sounds, which the user chooses to associate with a certain action. For simplicity, they were presented in the table as direct commands. From this table, we learn that it is required to input a large (from the user’s perspective) set of commands and rewards before the agent adopts a behaviour wanted by a user.

5.2.9 Disadvantages of binary user input

There are many aspects of the YES/NO interaction that point at its inefficiency:

- Additional actions added to the action pool of an agent increase the amount of user input required for each successful movement memorization
- YES command is not required as a simple lack of input replaces its functionality. The agent will maintain the last action as the NN will remain unaffected.
- The agent requires a lot of time in order to re-learn commands. The difficulty increases when the agent has to perform actions which were previously rewarded with NO.
- The agent simply takes a lot of time going through actions because it needs to guess something based on very basic binary input

These reasons make user interaction tedious, filled with unnecessary actions that also need to be performed in a fixed order. A better form of interaction was required.

5.2.10 Swipe mechanics

Looking at the games available for Nintendo or iOS mobile devices and observing the interaction available in them, I came up with new learning mechanics. The new mechanics were based on swipes, in it the previous YES/NO interaction would be replaced with swipes in different directions.

Holding a mouse button and swiping in a direction would issue a reward for the action which would also serve as a correction.

Table 10: Command log using swipes.

Command	Action	Reward (swipe)
GO UP	DOWN	UP
GO UP	UP	-
GO LEFT	RIGHT	LEFT
GO LEFT	LEFT	-

Because a correct action is known to the user and it is delivered to an agent, the use of DQN stops being a strict requirement. It is possible to use DNN and to backpropagate a correct result directly. However, it was decided to keep using Q table, just with a smaller batch number, as there was no need for the large number of replays any more. Rewards were automatically calculated, because there was no longer a clear YES/NO instruction, so an action was just compared to a reward and if they did not match, it was recorded as NO.

Table 11: Comparison of different elements of each user input mechanics.

	Swipe	YES/NO
Back Propagation	Gradient Descent using correct value	Gradient Descent using multiple replays (Q table), trying to predict a correct value considering previous successes and failures
Interface	Moving the mouse in a direction	Pressing either Y or N key
Number of actions supported	Limited by number of swipes the user can memorize and use	Very few actions, as the learning time drastically increases with more actions
Efficiency	All swipes are useful, cannot observe any unnecessary functions	YES command is almost useless
Changing an action associated to a sound	Easy by back propagating the correct action.	Can take a very long time, as an agent won’t be willing to try the action for which it was previously punished

As can be observed in table 11, YES/NO lacks a lot in comparison to the faster and more efficient swipe method. As a result, it was decided to remove the YES/NO mechanics, replacing it with the new swipe mechanics.

Here are some characteristics of the swipe mechanics:

- The actions are memorized faster as only a few corrections are needed because the right answer is backpropagated directly through the network.
- User performs very few actions – one swipe
- Re-learning is extremely easy as the agent’s actions are immediately adjusted through backpropagation.
- No need to go through all of the actions
- All commands are useful
- Possibility of having more actions by adding more swipes

5.2.11 Increasing rewards efficiency

Even with the audio level control, the audio spectrum is still affected in many ways by background noise. This results in a perfectly consistent by default spectrum being distorted and appearing to have a lot of variations from one frame to another.

As we are backpropagating the spectrum of a last frame where the sound was played, very often this frame ends up being distorted which results in it being not very characteristic of a sound that is being analysed.

As a result, when the sound is played next, only at the moment when the sound matches to the distorted learning example that was given previously, an agent will behave as expected.
In order to solve this problem, it was decided to backpropagate a certain number of last inputs instead of one.

The audio spectrums of a number of last frames where the sound was played are saved into a list. If they exceed a set value, the most outdated spectrum is removed in order to maintain the list to be of a fixed length. Then, instead of rewarding only the last input, the entire array of spectrums is rewarded and a correct action is backpropagated for each one of them. The optimal number of spectrums was 20 because using bigger numbers is risky, as older spectrums might belong to a different sound played earlier, while 20 allows it to encompass a broad number of latest outputs.

This solved the problem of only the last frame being evaluated. It also increased the learning speed of an agent, as it increased the number of times the correct value is backpropagated and therefore made it possible to make quick changes in weights that deviate massively from the target and therefore are difficult to change by backpropagating once.

5.2.12 Designing tutorial

Table 12: Tutorial script.

Tutorial text	Completion condition
"The black circle in the center of the screen is the queen. You can select her by clicking on her. Now try to select the queen"	Queen is selected.
"The queen is now selected. You can deselect her by right clicking anywhere. Try that now."	Queen is deselected.
"Good. Now select the queen again and click the button on the top left to produce larvae. Create 3 of them. They will grow rapidly to become Workers. "	Spawn Larvae button is pressed 3 times in the Queen menu.
"Now that all workers have hatched, select one of the workers and evolve it into a fighter by pressing a button on the top left"	Evolve Fighter button is pressed in the worker menu.
" A Fighter is hatching..."	Fighter pupal stage is finished.
"We now have a Fighter. Fighter responds to the sounds from the mic and also mouse drags in one of four directions. Hold left mouse button and drag to the right. "	The right swipe command is issued.
"Dragging the mouse in a direction teaches a Fighter to respond to previous sounds by going in the direction in a drag. It might take several attempts for a Fighter to learn. Producing a sound and then dragging is a way of teaching it."	700 frames had passed since the message appeared.

One of the aspects of the project at this point was the tutorial. It was created using Observer programming pattern which allows one object to listen to various events and notifications from other objects. For example when a subject of an observer walks to the right, it sends a notification “WALKING_RIGHT” to the observer. As a result, observer can respond in a certain way. In the case of the tutorial, the response would consist of advancing the tutorial by spawning a new text box. The script of the tutorial can be seen in table 12.

The message appeared in the right side of the screen, giving advices to the user on how to proceed next.

5.2.13 Project state

This section is a brief description of what the application looked like at this point and how all of the agents were represented. Following link contains a video of this version of Hive Mind application: https://vimeo.com/277795353

```
hive_mind_alpha_1$Queen@567b8994
null
```

Figure 41: Screenshot from early stages of Hive Mind application.

Figure 41 shows the queen in the centre of the screen with the larvae and workers clustered around it. The red circles are larvae – they cannot move on their own, unless pushed by another object. The queen spawned them because a user selected that in the menu. Black dots are workers. It takes time for larvae to grow into a worker.

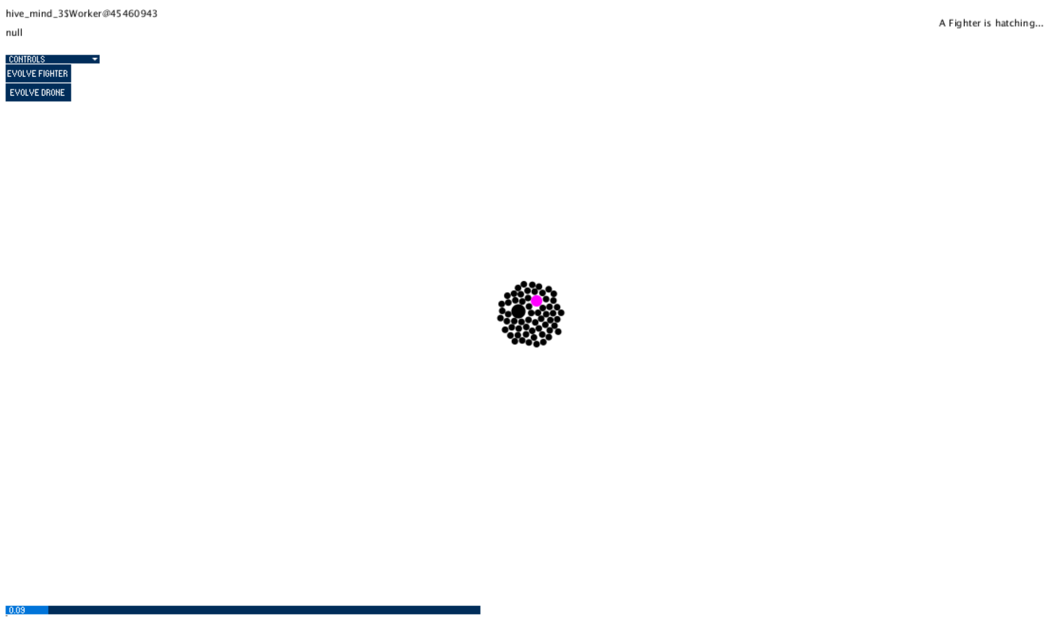
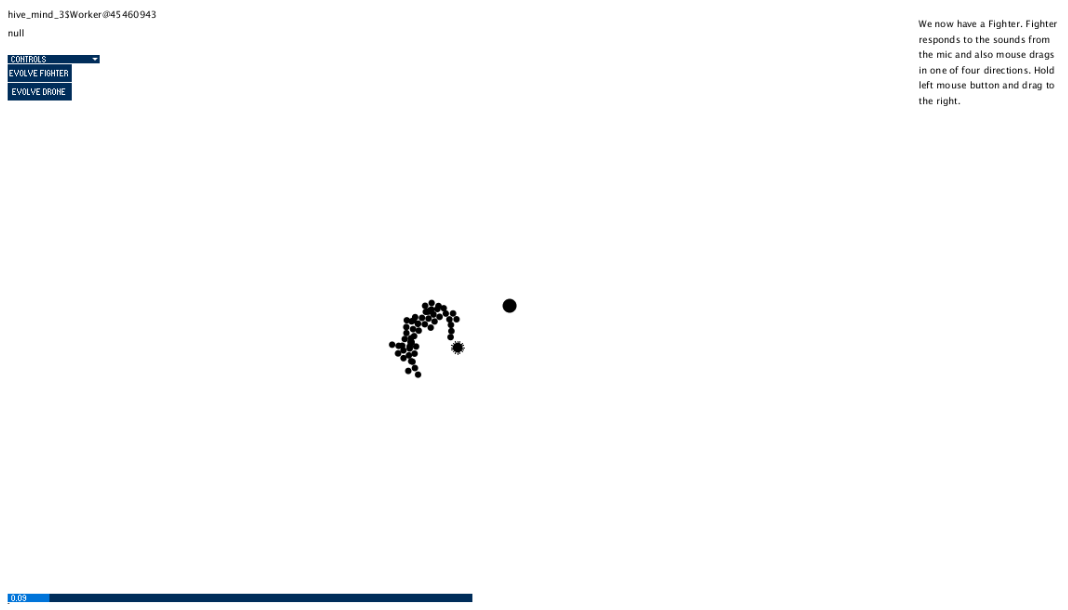


Figure 42: Screenshot from an early version of Hive Mind featuring evolving fighter.

In figure 42 the purple dot – evolving worker can be observed. The tutorial message in the top right suggests that it's evolving into a fighter. The top left menu shows available options for the worker which are to evolve fighter or to evolve drone.

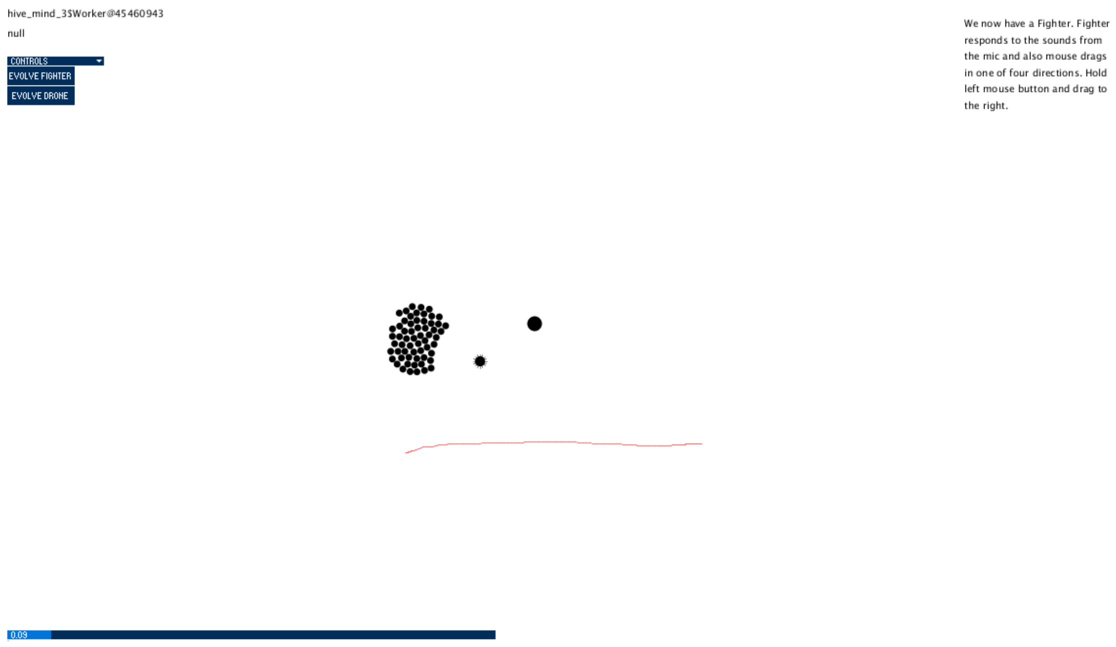


```
hive_mind_3$Worker@45460943
null
```

We now have a Fighter. Fighter responds to the sounds from the mic and also mouse drags in one of four directions. Hold left mouse button and drag to the right.

Figure 43: Screenshot of an early version of Hive Mind, showing a hatched fighter.

Figure 43 shows the hatched fighter (big spiky dot at the bottom left of the queen). The workers are avoiding it while clustering together.



```
hive_mind_3$Worker@45460943
null
```

We now have a Fighter. Fighter responds to the sounds from the mic and also mouse drags in one of four directions. Hold left mouse button and drag to the right.

Figure 44: Screenshot from an early version of Hive Mind, featuring a mouse swipe.

In figure 44 the red line below the agents represents a horizontal mouse swipe. When the swipe is performed, the line is drawn to indicate the registered swipe to the user.

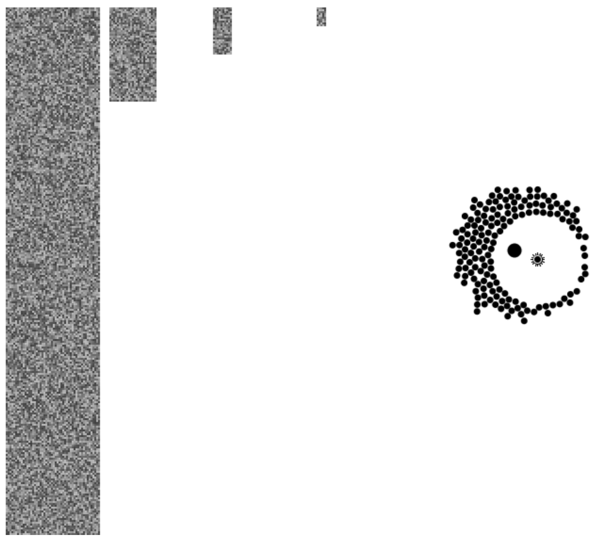


Figure 45: Screenshot from an early version of Hive Mind, featuring ANN visualization.

On the left side of figure 45, a graphical representation of agent's ANN weights can be seen. As there are a large number of inputs, the input layer occupies a lot of space. The output layer is a small rectangle on the far right. Changes in brightness can be observed on the grids, which points at the fact that learning is being accomplished.

5.2.14 Workers interacting with Fighter

At this point a hard coded system of workers avoiding a fighter and clustering together was taking place. However, it was not an easy task to make workers perform these actions reacting only to sounds on a computer learning level. ANN that would be a part of each creature would require a higher level of complexity, which slowed the system down a lot, as it needed to calculate the values for many agents at once.

There were various attempts at creating such a behaviour using DNN with the help of various sensors. Sound pings were also attributed to the workers so they could report their position to others. Figure 46 shows lines on some of the workers symbolising a direction from which they hear sounds. Circles surrounding the workers mean that they are producing a ping.

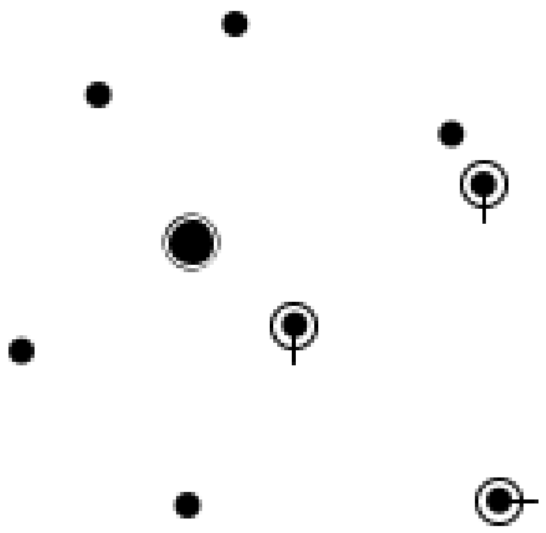


Figure 46: Workers with sensors producing sounds. Sounds are graphically presented as ellipses around the workers.

Simpler ANN would not work, as the positions of a fighter and other workers are constantly changing so the learning would either take longer, or it would require higher complexity.

In the end it was decided that such a system was not necessary, as the goal of its implementation was a deterministic behaviour that was already designed and imagined as it should be.

Figure 47: Screenshot of an early version of Hive Mind showing workers avoiding a fighter.

5.2.15 Ear system

In the attempts to implement a complex response to the other agent producing sounds, it was needed to create a system that would serve as an audio sensor (an artificial ear) that the agents would use in order to perceive sounds. The output of the ear could serve as not only the input for DNN, but also as regular values useful for programming the actions of agents.

The main problem that required a solution was the fact that sound-producing agents, instead of emitting constant sound, were transmitting small pings at an equal interval. This behaviour seemed to be closer to the reality and the solution would also be applicable to the cases where the sounds were being produced only sometimes.

To solve this problem a system was created. It incorporated a grid of sensors around a listening agent. The idea was partially inspired by SOSUS sensors that US used during the Cold War. When a sound is produced around a sensor, those sensors report the frequency of the heard sound to the listening agent. The data transmitted also contains the number of the sensor, as DNN input, however, each sensor could serve as an input so DNN would always know which sensor fired. Frequencies were used instead of audio spectrum simply because the spectrum data contains 513 values, which might require some time to process if there are many listening agents and sensors calculated in the same program.

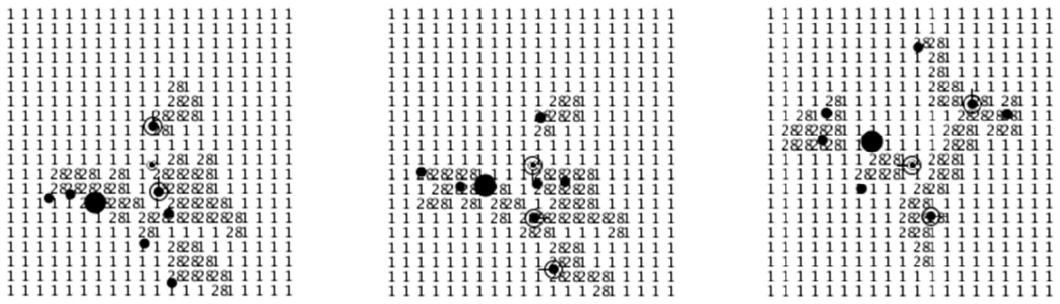


Figure 48: Screenshots of sound sensor grid.

Because the sounds are being transmitted as pings, it was necessary to implement a temporary memory system for each of the sensors. Whenever the sound is produced, a sensor fires and then keeps the value for several frames, unless it is overwritten with a different value.

Figure 48 shows 3 scenarios where frequencies of sounds are displayed in a form of a table. Note that sensors which are not detecting any sounds are marked with 1, while in reality their spectrum is 0. Numbers 1 on the table mean 0.01 inside the code. In the code, values from 0 to 1 are used in order to make it more convenient for ANN calculations, for the same reason 0s are not used because they can disable the function of some layers (as n*0=0, therefore the value of n wouldn’t matter). 0.01 is marked as 1 on the table because numbers are portrayed on a different scale and as the table is a visualisation created with the purpose of debugging, 0s needed to be tracked, so the value 0.01 is portrayed as 1 rather than 0.

5.2.16 Attempts at creating a complex behaviour with NDj4

Now that all of the agents (workers) were equipped with an audio sensor, I started wondering whether it was still possible to also add an ANN to each of them. Some of the ANN written for this project are DNN and therefore some of them rely on n-dimensional arrays (tensors). It is common to assume that such arrays require a pre-written library, as they are not a part of Java programming language. There are lots of libraries available online that provide them, however, none of them were used in this work. All of the calculations were written (and some designed) by me. The n-dimensional arrays that were used in this work are as well of my own design which, as I thought, could be the cause of the problem where the calculations would slow down immensely for a large number of networks in the code.

A key to my solution was a function that returns a single row from a 2D array. Using this function helped me emulate n-dimensional array functionality, as there was no need for using an array with more than 2 dimensions in this project. Here is what it looks like in the code:

```
float [] getaRow (float[][] array, int i) {
  float myRow []= new float [array[i].length];
  for (int l = 0; l< array[i].length; l++) {
    myRow[l] = array[i][l];
  }
  return myRow;
}
```

In order to test the effectiveness of my solutions, I decided to try the ND4J library which is providing n-dimensional arrays for Java. The arrays are specially designed for use with ANN and can perform calculations on the GPU, which, according to their official documentation, could speed up calculations. Besides, many calculations in the library are done in a faster language such as C++.

This particular Java library required Maven and therefore it was not easy to continue using the Processing application, as is does not have any system that allows managing Maven dependencies. As a result, the project had to be transferred into the Eclipse environment, and the entire code had to be modified, as some of the Processing features were no longer available. When the n-dimensional arrays were replaced with ND4J arrays, there was no noticeable change in performance. It seemed that there were no problems related to arrays. Using CUDA in order to process the code on the GPU also did not increase the performance. Per-haps, there was no problem at all, and ANN is just not a good decision for multiple agents.

To make a last check, I decided to include the DL4J (Deeplearning4J) library to the Java project as well. DL4J library is made by the same people who made ND4J and their library is known for having an example which can learn to complete DOOM levels, as well as some examples that play classic arcade games.

The original DNN was temporarily replaced by an identical DL4J DNN network. The first impression was related to the fact that the code was taking a lot of time to load. After long loading, the agents would start acting, but they would slow down around the same point as before, using my network. It happens when the replay buffer becomes full, there is simply not enough computing power in my laptop to process that number of replays (when the buffer is full, the system can pick full batches as well). To clarify, ANN in the library includes its own replay managing system and creates the batches of replays using its own classes. Running the code with a smaller replay batch resulted in faster performance, but the framerate was still too low to make the result enjoyable as a game or a toy. Decreasing the replay batch further would prevent the program from learning.

After consulting additional information on the official web site, it seemed that their ANN are the most efficient when working with large data sets. As there are no large data sets used in this project and all the data is processed in real time, there is no significant advantage in using a library such as ND4J or DL4J.

The project was reverted to what it used to be before using the libraries and it was decided to not continue creating a separate ANN for each of the agents (workers) and instead focus on the user interaction between each other and other agents (fighter).

5.2.17 Designing a new environment

At this point, almost all of the behaviours described in the Preliminary work chapter were implemented. There was the queen, workers, fighters (previously warriors), each acting accordingly. But transforming this into a game-like, toy-like or playable experience was not seen as a very easy task. I had created a system complex enough to serve as a basis for a toy, it was the environment that was lacking features.

It was decided to design a new environment, and remove all unnecessary features that were interfering with the main goal of application. The tutorial, multiple types of creatures, clickable characters, life cycles. All of that was removed. All that remained was one simple agent (fighter) using DNN that allows learning various commands from the user.

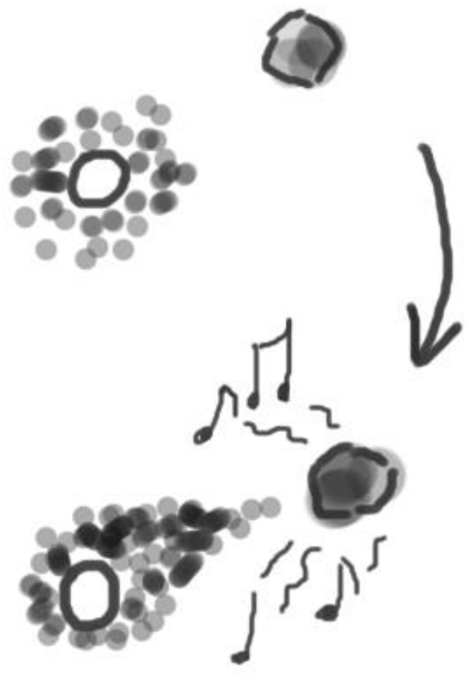


Figure 49: Sketch showing particles being attracted by a sound from an object.

In my mind I was seeing a nucleus of a cell, surrounded by other cell components (particles). All of the particles could move around freely, but they would still cling to the core (nucleus), sometimes reaching for nearby food. They would be blind but could hear sounds emitted by objects around them (pick-ups) and using audio sensors from the subsection 5.2.15 they could determine the location of a sound and its frequency. Particles would also learn which audio frequencies meant food, and which meant danger. A user would control the core using a microphone and teaching it various commands.

Figure 49 illustrates what a core with particles clustered around it would look like. It also shows how the sound produced by an object (pick-up) can influence the behaviour of particles.

5.2.18 Agents reaching for objects

As the system to control the core was already implemented (subsection 5.2.10), the next step was to de-sign the mechanics which would be used by particles in order to reach for objects (pick-ups) or avoid them.

First of all, each particle was equipped with an ear so they could hear and determine the locations of sounds. Because they needed to process sounds and react to them in a unique way, a DNN was added to them. DNN was a reworked version of a network used in the Jumping over ball examples, described in subsections 5.1.2 and 5.1.3. The network was compact enough and the only input that it needed to receive was a sound frequency. The resulting DNN was very fast and it was possible to have many agents, each using its own network.

Apart from that, code that enabled particles to follow the core was added. It was a coded behaviour, but it had a priority set to it, which meant that whenever any sound was perceived by a particle, it would always prioritise its response to the sound over hard coded behaviour. The resulting behaviour was particles clustering around the core and sometimes reaching for pick-ups. Pick-ups that populated the map were dummy geometrical figures that were producing various sounds. Some of the audio frequencies attracted particles, some repelled them. It was a sign that DNN was working. However, there was still no backpropagation or any kind of alternative mechanism for adjusting weights.

5.2.19 Populating environment with passive agents

The objects in the world had no meaning or effect associated with them, therefore new passive objects were required.

It was decided to create 2 types of objects: good and bad. Good objects would cause an agent to reproduce by creating 2 identical individuals (parent's ANN would be copied over to them). Bad objects would cause a particle to explode destroying every other particle around it at a high range.

The way the objects were generated was too sudden, so the unprepared user could end up with the bad object placed right on top of them, therefore a mechanic that allowed objects to grow slowly before starting to have an effect was implemented. Starting with a small dot, the object grows into a full individual over time and can only be good or bad when it reaches its full size. Objects only produce sounds when they are fully grown.

Bad objects have a frequency different from good objects so particles can learn to distinguish them.

Apart from the functions described above, all objects are slowly floating through the map and can collide with each other in which case they slowly float away from each other. The movement was added in order to create more dynamics in the map, however the movement needed to be slow because the user controls are complicated and it can be frustrating catching up with a fast moving object.

If an object is consumed or leaves the map, it is destroyed and it will soon be replaced with a new object which will start growing at a random place on the map.

Touching a bad object causes an explosion, however touching a good object won't cause reproduction unless the agent is choosing to consume it. It should be seen as stepping on a mine versus seeing an apple and having the choice to ignore it or to eat it.

5.2.20 Survival-based learning

A component that was missing was learning for particles. They already had DNN and the environment had various objects in it. It was now required to create a system which would allow the ANN to adapt.

It would be possible to implement backpropagation and just tell the agents which object is bad and which one is good when they pick it up, but it would be too deterministic. It would be more valuable for this project to let the agents decide for themselves what is good for them.

In subsection 5.1.2 "Jumping over ball" example was described. DNN in that example was identical to the one used by the agents in this application. The way the creatures are learning in that example is that at the end of each life-cycle/generation the fittest individuals are selected. After that the fit reproduce so the next generation has better fitness then the previous one.

In this example reproduction is achieved through picking up a good object, while death/punishment is inflicted upon touching a bad object. As a result, newborn individuals will also pick up good objects as they will inherit their parent's ANN. The number of particles that step on bad objects will diminish too as they will soon die out leaving space for particles that don't step on bad objects. The learning is definitely happening already. In order to increase its variety, the possibility of mutation was added, so some new born individuals may end up with a mutated version of their parent's ANN. Agents don't have a fitness function or a value determining their fitness, so possibility of reproduction or death are purely results of their actions.

5.3 Final Application

The content of this section consists in the description, presentation and analysis of the final application. Technical references such as names of equations and algorithms will be done assuming that the reader is familiar with them. Their more detailed description can be found in section 2.6.

5.3.1 Final application description

A demonstrational video of the final application can be seen here:
https://vimeo.com/277741291

The final application is a 2D world populated with various agents. There are 3 types of agents: pick-ups, core (user-controlled) and particles.

The core is controlled by a user using the microphone. The user can input various sounds and the core will respond in a way determined by its DNN. The user is able to manipulate a response by using the mouse. Clicking and dragging in a direction can communicate a direction to the core. For example, dragging to the right creates a “RIGHT” command, dragging in any other direction creates a command in that direction. If a command is issued after the sound is produced, the command will be backpropagated for that sound, so next time when the same sound is produced the response of the core will follow the command. The mechanism will be described in more detail in next subsection.

Particles are smaller agents clustering around the core. Each particle has a DNN which receives sounds produced by nearby world objects (pick-ups) as an input. An output of the network is one of the actions: to move away from the sound, to ignore the sound or to move to-wards it. Particles are clustered around the core and at some point, when there is a nearby sound-producing object, they may reach towards it, ignore it or avoid it.

When particles are too separated from the core they may lose it and remain static within the environment, occasionally reaching for pick-ups or avoiding them. If the core is brought closer to them, they will continue following it.

Because picking up bad objects destroys particles, a low-fitness population reaching for bad objects will soon be exterminated leaving only those particles that ignore or avoid bad objects. When a good object is picked up, the particle that picked it up will reproduce, creating offspring which copies DNN weights of a parent. Sometimes on the reproduction event a mutation of weights can happen which might render a behaviour of offspring to be different from one of a parent.

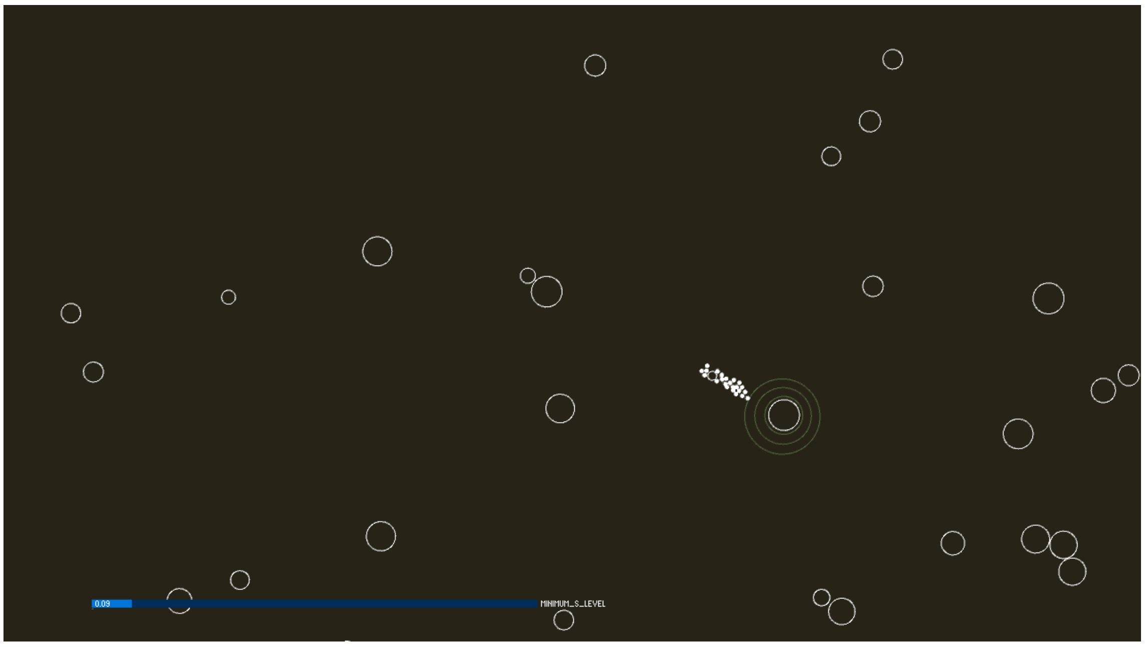


Figure 50: Screenshot of final application showing particles reaching for an object.

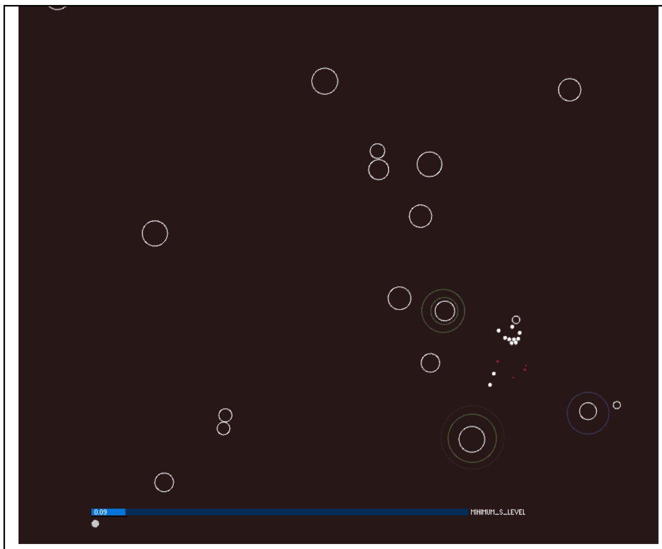
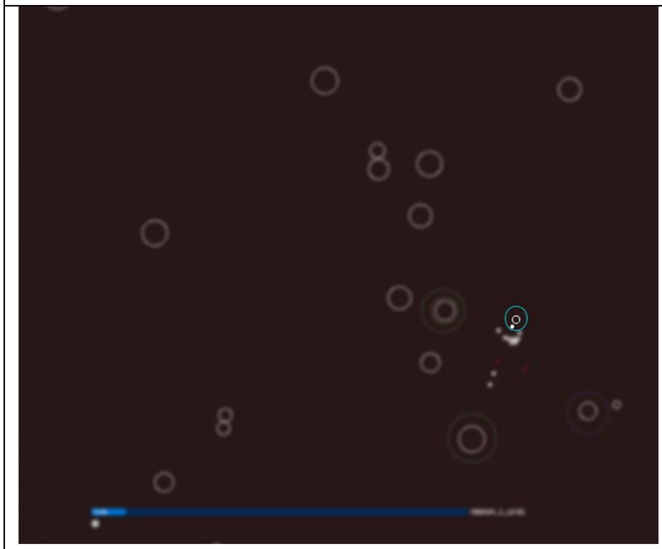
Another type of object populating the environment are world objects (pick-ups). These objects are passive and don't have any kind of computer learning algorithm assisting their behaviour. They can appear on any place of the map at random and will grow until reaching their maximum size. While they are growing, they cannot be interacted with and they are not producing any sound. Once grown, they will start sending pings which can be heard by particles. There are good and bad objects, their sound frequencies differ so that they can be distinguishable. When a good object was picked up by a particle, this particle produces 2 similar ones, however, if it was a bad object, the particle explodes destroying other individuals within a certain range.

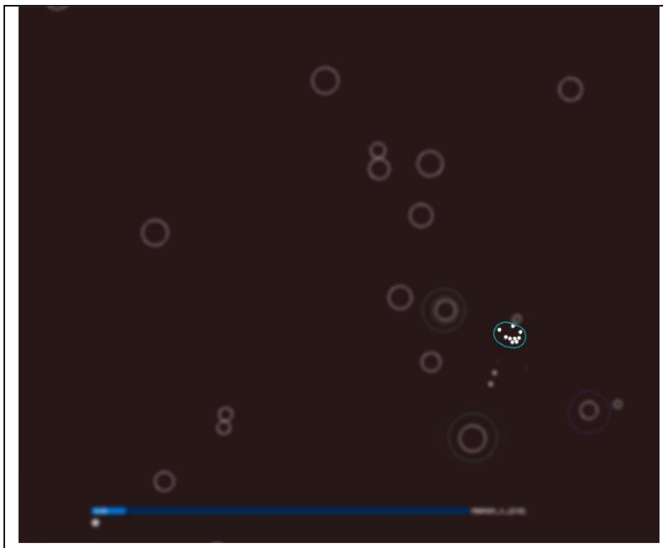
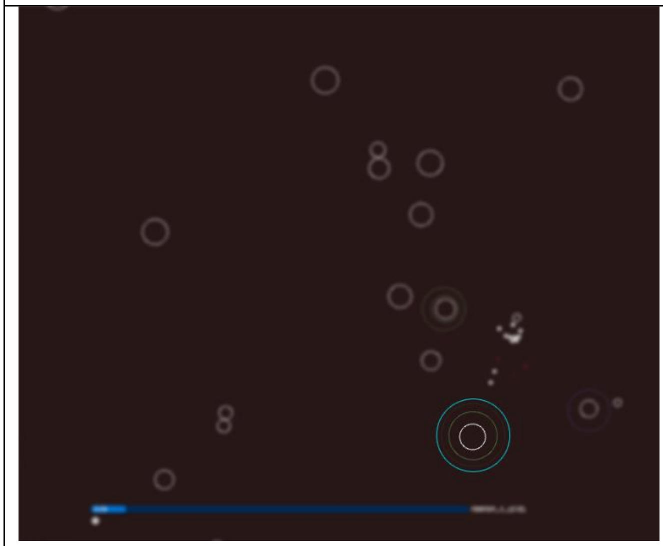
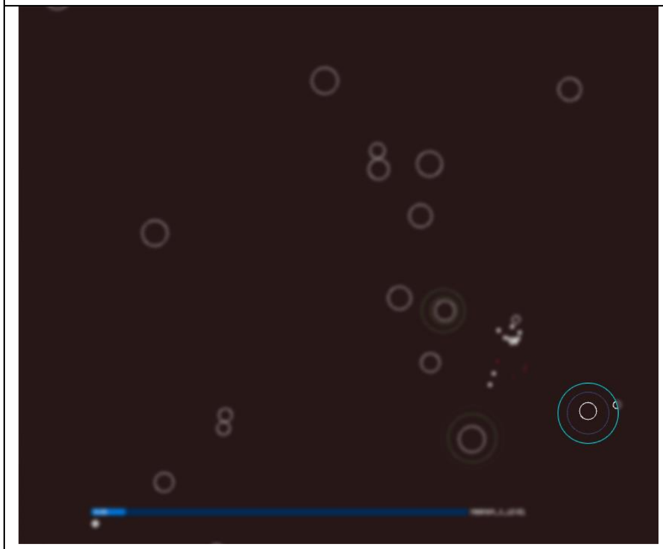
Sounds produced by pick-up objects can be heard by a user when the core is near them. If a user cannot hear the sound of an object, it doesn't mean that the object is not producing it, it's just not close enough to hear it. Sounds that are not in range are muted, but only for the user, the particles can still hear them. The closer a user is to a sound, the louder it will sound. This mechanic was implemented in order to avoid having too many sounds played at the same time, making them indistinguishable to the user.

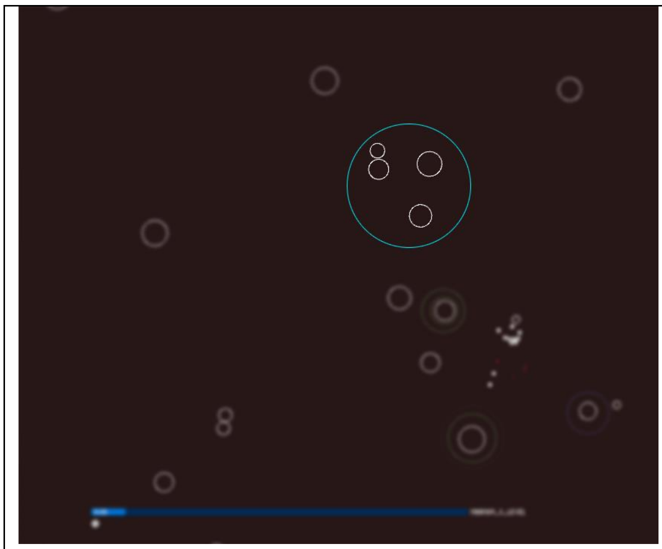
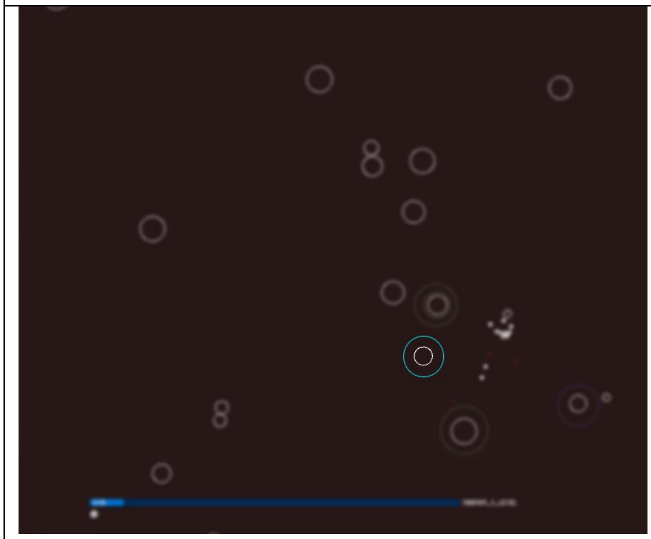
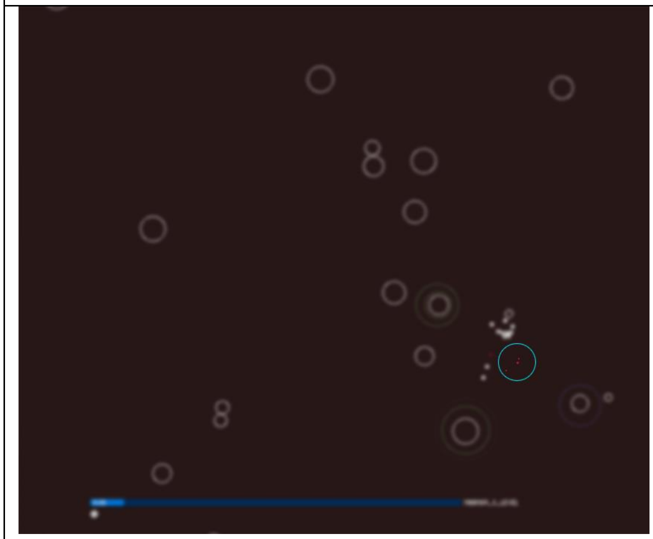
At the bottom of the screen there is a slider for audio levels. The value on the slider determines the minimum level that the sound from a microphone should have in order to be perceived by the core. This mechanic was created in order to prevent background noise from affecting the behaviour of the core.

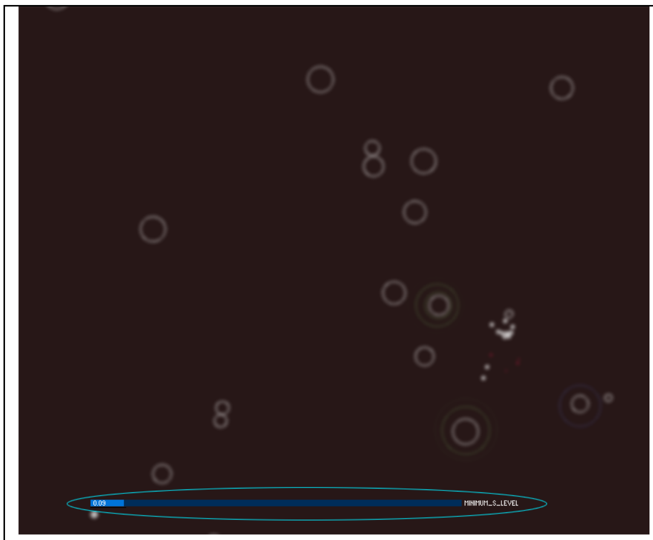
Table 13 shows graphical representations of all elements and agents, it also offers short explanations of why these graphical decisions were made.

Table 13: Agents within Hive Mind application.

	Scene The entire scene is populated with agents. If there are too many pick-ups, scene will stop spawning more until some of them get destroyed either by being picked up or leaving the borders of the screen. Background colour is constantly changing its hue, but remains dark so all objects are clearly seen.
	Core Core is a small hollow circle. It is not filled with colour in order to make it easily distinguishable from the particles.

	Particles Particles are dots clustering around of or close to the core. They are the only filled circles in the application which creates contrast with remaining agents. The contrast is needed in order to under-line their important role.
	Good object / good pick-up They look like any other objects, except that they are producing sounds. When approached by the core, they can be heard, otherwise their sound is still present, but muted to a user. When in range, user can see a sound visual effect in a form of expanding circles. Colour of the visual effect depends on audio frequency.
	Bad object/ bad pick-up Same as good objects, except that the sound is produced at a different frequency, which also results in different colouring of the sound visual effect. In this case it’s blue.

	Out of range objects Objects that are out of user's range are plain circles. Their sound is heard by particles, but muted to the user in order to limit a number of sounds present at the same time.
	Growing object When a pick-up object is born it starts growing. Until it reaches the full size, it won't start producing sound. It is impossible to determine whether the growing object is going to be good or bad. It's impossible to interact with it even if it's in range. It can still collide with other objects.
	Blood When an explosion caused by touching a bad object happens, the exploding particle produces a "blood splatter" effect. Random number of purple particles fly out in random directions. Blood stays on the scene for some time before disappearing.

	Audio levels slider Audio levels slider at the bottom of the screen is the only UI element in the application. It can be interacted with by dragging the slider in any direction or clicking at a desired point of the slider.

The main focus of the application was implementing interesting behaviour and interaction. In terms of the appearance, it was decided to avoid using unnatural shapes and instead utilise ellipses because they reflect the biological nature of simulated creatures. The final look somehow resembles Osmos, which is a 2009 video game available for most of mobile platforms and desktops. Figure 51 shows a screenshot from the application. The topic of the game is survival of the fittest, which matches the concept of Hive Mind.

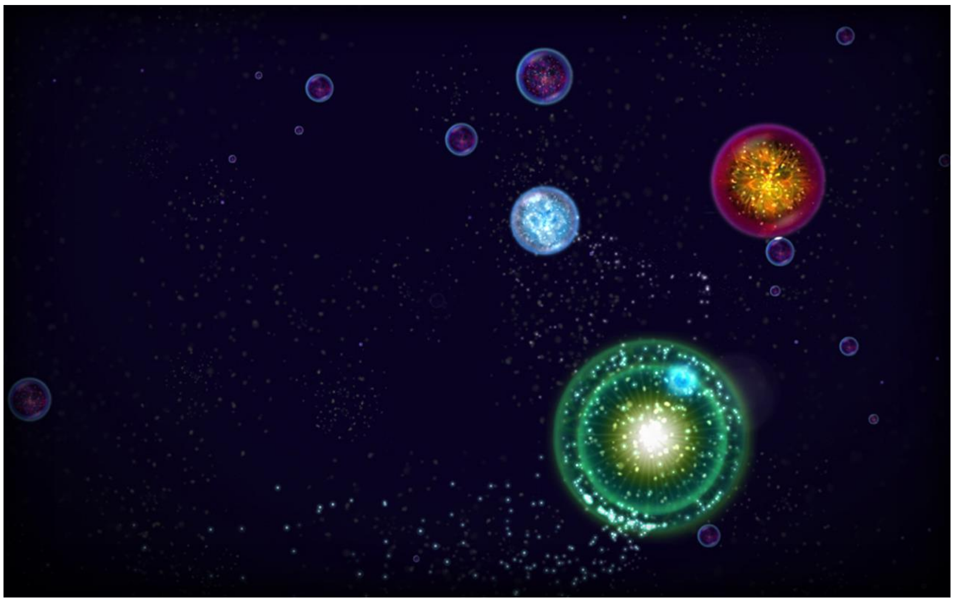


Figure 51: Screenshot from Osmos video game.

The use of circular shapes, in the mind of a user, can relate to the experience of looking through a microscope or observing the night sky. Both cases involve looking at a cosmos, at a micro or macro scale. These experiences are the source of inspiration for the visual appearance of the Hive Mind.

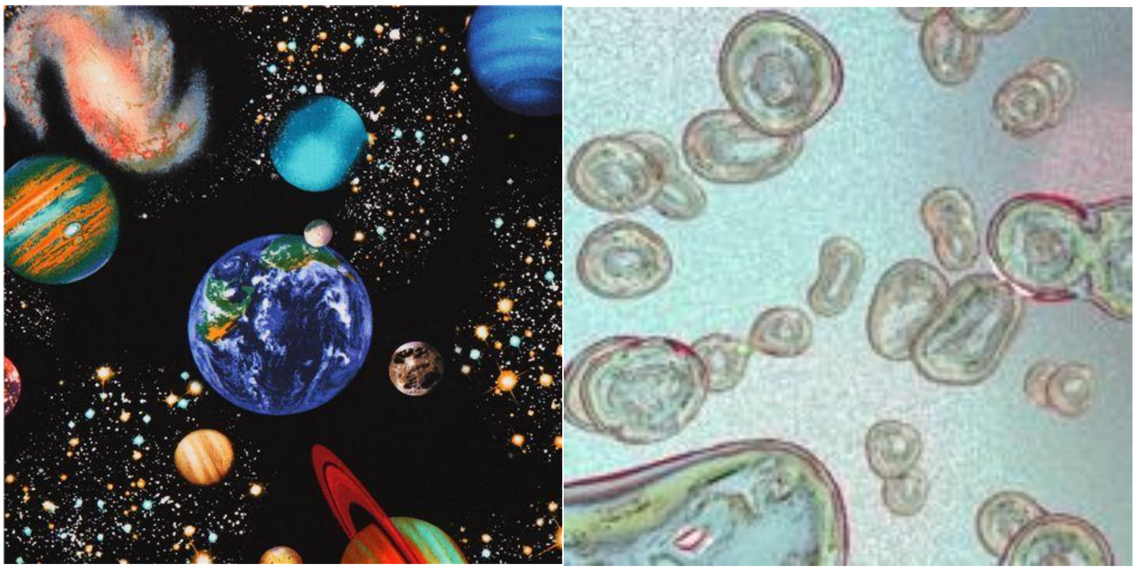


Figure 52: Artistic depiction of stellar objects (left), cellular division though microscope (right).

5.3.2 Core learning

This subsection is going to discuss the functionality behind the learning system that the core (user) is using, primarily DNN and its components.

As it was described previously in the section 5.2, the solution to a problem that was present during the development shouldn't've necessarily been DNN, because the earlier functionality featuring YES/NO feedback from a user is no longer used and therefore there is no strict requirement for a Q table. However, even though the gradient descent backpropagation of a correct result is used, it is profitable to keep a table of replays, as they significantly increase learning speed, allowing to backpropagate multiple cases at once instead of just one. For real-time learning it is a valuable feature as it would be extremely inefficient if a user had to repeat the same action multiple times in order to make the net adjust to their preferences; it's better if ANN does it by itself. Therefore, an initial reinforcement learning DQN had to be adapted in order to function in a way similar to a supervised learning DNN by simply adding the user input with maximum reward to experience replay table.

Figure 53 shows layer configuration of DQN used for the core. Numbers within the circles indicate a number of nodes in each layer. Connections between nodes are established from each node to all of nodes of a next layer. Every connection has weights associated to it. In figure 54 a more direct representation can be seen.

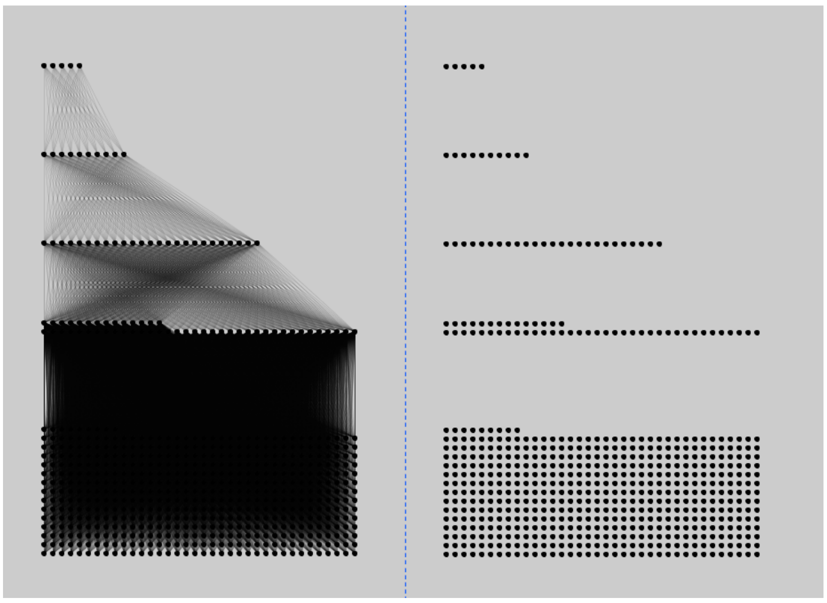


Figure 53: Illustration of all DQN layers and connections (left).

The number of inputs is 513. This is a number of values within an audio spectrum transmitted every frame from a microphone. The core receives all spectrum values and then feeds them forward through hidden layers. Feed forward for each output is accomplished in the following manner:

```
for (int j = 0; j < numberOfInputs; j++) {
      if (inputs != null) {
            outputs[i] += inputs[j] * weights[i][j];
      }
      }
outputs[i] = (float) Math.tanh(outputs[i]);
```

From the above code it can be seen that TANH activation function is used.

Actions resulting from output layer can be performed by the core. There are only 5 actions available to it. They are represented as following:

```
{ "up", "down", "left", "right", "nothing" };
```

Every time when DNN feeds forward audio spectrum values, the core calculates a maximum value of all the outputs, associates it to an action and performs it.

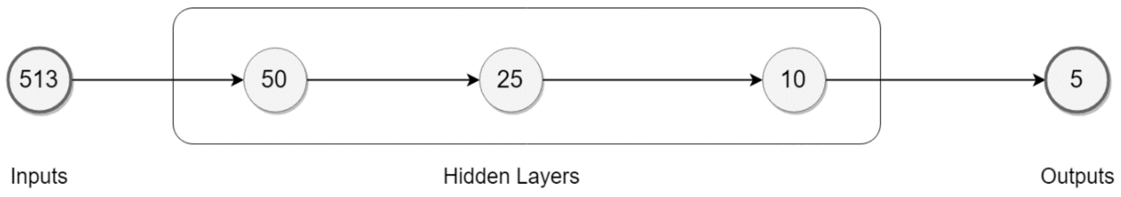


Figure 54: Scheme showing DQN configuration of the core.

Backpropagation of this network won't happen unless there is an input from a user. Clicking the left mouse button and dragging in a direction will create a command associated with that direction. For example, "RIGHT", when the mouse is dragged to the right. If a command was issued by a user, it will be added to the experience replay list together with an input (audio spectrum values) and will have a high reward (100) associated to it.

After the data was added to the replay buffer, a batch of replays will be selected and the net will be trained. Here an additional feed forward is performed on a copy of ANN because in DQN learning both world states before and after the action need to be analysed. Like this the network knows what the input was like before the action and how it changed after the action. However, this is not entirely relevant in this project because a world state (sound spectrum) is determined by a user and is not entirely related to the actions of an agent. After that the network will be trying to predict a future reward based on previous rewards from experience replays. After that a selected replay is backpropagated in main DQN and a copy of the net is updated.

Backpropagation of each layer (calculating MSE) is accomplished as follows:

```
public void BackPropOutput(float[] expected) {
      for (int i = 0; i < numberOfOutputs; i++) {
            error[i] = outputs[i] - expected[i];
      }
      // gamma calc
      for (int i = 0; i < numberOfOutputs; i++) {
            gamma[i] = error[i] * TanHDer(outputs[i]);
      }
      for (int i = 0; i < numberOfOutputs; i++) {
            for (int j = 0; j < numberOfInputs; j++) {
                        weightsDelta[i][j] = gamma[i] * inputs[j];
            }
      }
}
public float TanHDer(float value) {
      return 1 - (value * value);
}
```

It is worth noting that apart from replays there is a list of last actions performed by an agent, paired with the sound spectrum input. The list is limited to a certain number of entries and cannot hold more values than maximum, therefore when new values are added, the oldest ones get removed. Every time when DQN operations are performed, they are performed for each of the entries in the list, instead of just one last entry. A reason for this was non-consistent data from a microphone. Backpropagating a number of last sets of values is superior to choosing just the last set of values, as the audio spectrum shifts a lot from one frame to another and it is important to make sure that backptopagation is done for all variations of one sound.

Additional parameters of this DQN are setup as follows:

```
final int REPLAY_MEMORY_CAPACITY = 12; // the total number of
replays that can be stored in the memory
final float DISCOUNT = .99f; // discount factor
final double EPSILON = 1d; // exploration rate
final int BATCH_SIZE = 12; // how many replays are processed at a
time
final int UPDATE_FREQ = 1; // update rate of network
final int REPLAY_START_SIZE = 12; // how many replays need to be
accumulated in buffer for batches to start being selected
final float RATED_ACTIONS_NUM = 50; // number of last actions that
are going to be rated by a user, there needs to be many actions (at
least 10) for precision
final int INPUT_LENGTH = 513; // size of input that an agent
receives. sound spectrum is the total of 513 numbers
final public float L = (float) 0.001; // learning constant
```

Some of the values above are not completely relevant. One of such values is epsilon, which is the exploration rate. Exploration rate determines how often an output might purposefully deviate from a correct output. This exists in order to enable agents to explore new behaviours instead of sticking to only one that worked last time. With a good exploration rate, agents are more likely to discover better actions in some situations. Because the correct supervised learning value is pre-determined by a user, it is not needed in this scenario and therefore left at 1, which on the scale from 0 to 1 would be equivalent to 0% chance, where 0 would mean 100%. If the learning was accomplished using a previous YES/NO input, having exploration rate would be extremely relevant.

Replay memory capacity should, in most cases, be a larger value, but because learning is happening in real time and the input is the entire audio spectrum, it is set to a low value in order to maintain high framerate. The same applies to batch size.

Update frequency is not completely relevant as update is happening whenever a user is introducing a reward. If a reward was issued every frame, it would be relevant.

5.3.3 Particle learning

Smaller agents (particles) surrounding the core have their own ANN configuration. They are using DNN with the layer configuration shown in figure 55.

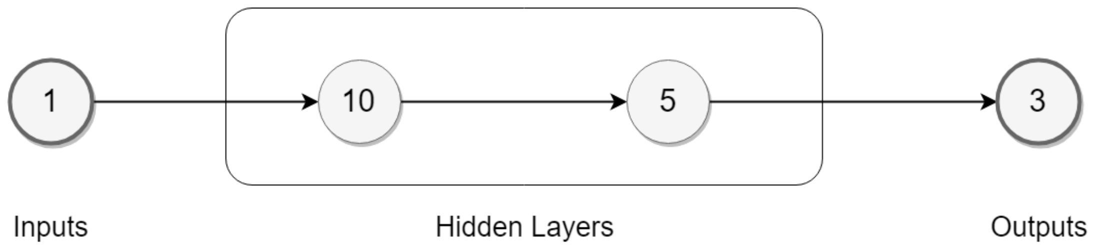


Figure 55: DNN configuration of particles.

The number of nodes in hidden layers is relatively high in relation to the number of inputs and outputs. A reason for that is that more hidden layers in some situations generate more complex and interesting responses which are interesting in artistic terms. Particles were imagined as chaotic creatures choosing their behaviour on their own, without any direct intervention from a user, therefore a complex and erratic behaviour would portray particles as in-tended.

The feed forward process of this DNN is identical to the DQN described in the previous subsection.

In contrast to a previously described DQN, this network is performing feed forward every frame if there is a sound from a pick-up object that can be heard (is in range). Backpropagation, however, is never done. Learning is accomplished solely through reproduction and survival over generations.

When a particle reproduces, it creates two identical copies of itself that contain copies of their parent’s ANN. Therefore DNN of particles that were able to achieve reproduction will stay in the application as fitter ones.

In order for evolution to progress and new behaviours to emerge, mutation algorithms were added. Whenever a particle reproduces, an offspring has a chance to mutate. This chance is determined by mutation chance constant. When mutation happens, some of weights in the network change. There are 7 possible mutations:

- Random value is added to weight
- Sign of weight is inverted
- Random value is subtracted from weight
- Weight is divided in half
- 0.1 is subtracted from weight
- 0.1 is added to weight
- Weight is set to 0

5.3.4 Java classes

The code of the application consists of 14 different Java classes. Their brief descriptions and structure can be seen in table 14 and figure 56.

In figure 56, a class diagram is shown, it is necessary to note that the parent-child connections between all classes and a main class had been omitted in order to maintain clarity. This parent-child relationship that was not shown in the table exists due to the use of Processing library which required every class that used Processing language elements to have main class registered as parent. This is a measure required in order to successfully manage Java classes in Eclipse in a similar manner as it is done in the Processing application, avoiding nesting the classes inside main class.

Table 14: Java classes of Hive Mind.

Name	**Description**
HiveMind	The main class of application
DeepQNetwork	Class that adds functions of DQN, requires DNN to function
Ear	Contains the functions related to the particles's ability to detect the sound around them and to temporarily save the sounds on a grid of sensors
Fx	Visual effects for pick-ups producing sounds and particles exploding
GameLoop	Updates all of the game objects every frame and removes the dead objects. It also places new pick-ups when there are not enough of them on the map.
GameObject	Class for all game objects such as user, particle and WorldObject
Hud	UI elements
Layer	Class that describes an individual layer of DNN
NeuralNetwork	Class containing all basic DNN functions
Particle	Class describing particles and their behavior.
Player	Class describing core and the functions associated to it.
Replay	Replay storage class.
Swiper	Functions determining the direction of mouse swipes, translating them to actions and drawing them.
WorldObjects	Pick-ups and their behavior.

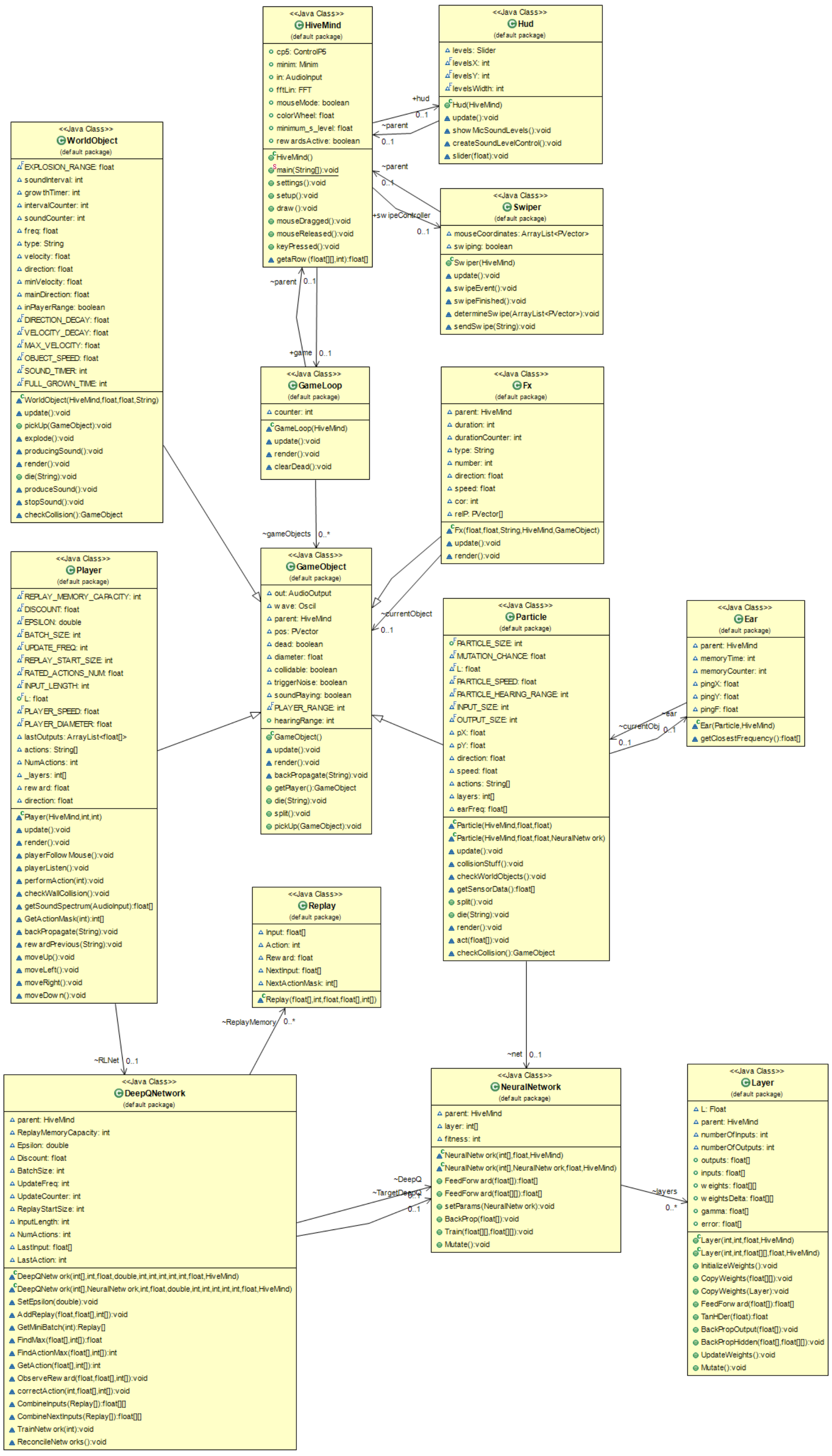


Figure 56: Class diagram of Hive Mind.

5.3.5 Analysis

When application starts a user is faced with a challenge to teach the core agent to respond to sound commands as soon as possible before particles get destroyed by bad objects. Once control is established, the goal of a user shifts into teaching particles to recognise good objects and distinguish them from bad ones. A user will have to rely on particles and their reaction in order to determine which objects are good and which ones are bad. It is a relaxing experience that can be enjoyable as a user is observing an accumulating number of particles.

It is interesting to observe the behaviour of particles when they are reaching towards pick-ups and extending into a line, helping each other to get to objects.

User interaction in this application is established in several phases. When an application starts, a user has no control so far, however, with time, control over the core agent can be established and at which point it turns into a completely different experience. Phase two starts when a user discovers that their actions affect particles too. Particles then follow the core which is controlled by a user and can be steered to avoid bad objects, or to pick up good objects.

Behaviour of particles picking up objects has its own dynamics related to their evolution. Particles that exploded by picking up a bad object will be dead, therefore there will be less particles in a next generation trying the same action, however, mutations can still cause particles to evolve bad habits.

The core itself is controlled by a user through a microphone, however, it can often deviate from its path because of background noise or because a different correct answer was backpropagated through DNN. It can generate unpredictable behaviours that can be interesting to observe.

Particles that were separated from the core remain in place, but whenever an object is moving by, it can be picked up by them. As a result, a whole new colony of particles can be formed. This can result in an unexpected outcome and emerging mechanics.

5.3.6 Possibility of future development

Continuation of development of this project is visible. Its further development and improvement would be directed towards increasing playability.

Adding more different pick-ups and effects for them, adding interesting visual mutations for particles and perhaps adding more types of particles can increase complexity and playability of this project.

Adding clear goals for the user can also transform this application into a game, while currently it remains an interactive toy.

6 CONCLUSIONS

The research was conducted in two different parts. The first part was a comprehensive analysis of the literature to determine key factors relating to A-Life agents. The second part was planning the structure of the software and finally its development.

The objective of the part one analysis was to identify key factors relating to existing A-Life models and to understand how A-Life agents can be used within an entertainment software. Design decisions that have to be made in order to achieve a desired behaviour were investigated within existing examples.

The objective of part two was developing an entertainment software featuring A-life agents. The software needed to include the A-life agents as one of its key elements, allowing a user to observe their behaviour.

Further on, an interaction between A-life agents and the user had to be created, making it possible to manipulate agents and make decisions about their life cycle and evolution. A certain level of control over agents needed to be established.

The final objective was to create a possibility for agents to evolve sound recognition at a certain level, be it through interaction with a user or with other agents.

For the conclusion of part one of this thesis, a study based on literature, computer games/programs and web articles had been accomplished. During this study, an understanding of key elements of the A-Life field had been achieved. Design decisions within studied examples were analysed, making comparisons with other projects and highlighting the pros and cons of some of them.

The use of relevant algorithms has been studied in preparation for part two of the project which is the development phase. Taking into account algorithms and design decisions analysed in the Literature Review chapter, an outline of a prototype was described. With an understanding of key projects within the A-Life field, necessary techniques for a creation of a toy with A-Life agents were exposed.

During the second part of the thesis, an entertainment software featuring A-life agents was developed. The software included multiple A-life agents as its key elements. Their behaviour could be observed and manipulated by a user.

An interaction between A-life agents and a user had been created. It was possible to manipulate the core agent and teach it various movements using sound. When a user established control over the behaviour of the core agent, they also gained control over minor agents (particles), where a user could directly interfere with their evolution.

As for the final objective, which is achieving evolution through sound, it can be considered achieved. The core agent can perceive any sound that was inputted through a microphone and as a result can learn to be controlled using only sound. Audio commands controlling the core agent also affect minor agents clustering around the core which results in them being affected by an audio input as well. Besides, minor agents are able to perceive sounds from their virtual environment (sounds from other agents). They evolve to perceive these sounds and respond to them in the best way possible.

The resulting application can be considered a toy because a user can interact (play) with it and the actions of a user affect the state of the environment and the artificial creatures within it.

LINKS

https://drive.google.com/file/d/1Gso2FCLR9EjKyKeQqvktLObwWoUls6BO/view?usp=sharing – Executable JAR file of Hive Mind.

https://vimeo.com/277741291 - Hive Mind. Video shows the final Hive Mind application and user interaction with the agents. The beginning of the video shows the agent learning various sound commands, inputted through the microphone. Mouse swipes are documented in the upper left corner showing a corresponding direction. Second half of the video shows agents responding to sounds that were taught to them earlier. Bottom right shows how the sounds are produced using Garage Band app on an iPad.

https://github.com/bug-ugly/hivemindgame.git - Hive Mind Eclipse project. This repository contains full code of the final Eclipse project.

https://vimeo.com/277795353 - Hive Mind (early version). Video showing an early version of Hive Mind application which is radically different from the final application and has many discarded functionalities. Video shows a user following a tutorial in the upper right corner by interacting with the application using the mouse.

https://vimeo.com/277782107 - Agents learning to jump over obstacles in Processing. This is a video demonstration of Jumping over ball application where agents learn to jump over obstacles using deep learning. Beginning of the video shows behaviour of untrained agents, it is followed by a time lapse of the agents learning. In the end behaviour of trained agents can be seen. Upper left corner shows the generation number.

https://vimeo.com/252213430 - Bug World. Video showing agents of Bug World in action.

https://vimeo.com/277783441 - Arduino piano. A short video demonstration of functionality of a small piano prototype.

https://vimeo.com/277789895 - Bug sounds. It is primarily an audio (presented as a video) demonstrating the bug sounds designed during an earlier stage of this project.

REFERENCE

Ackley, D. H. (2013). Bespoke physics for living technology. *Artificial Life*, *19*(3–4), 347–364. https://doi.org/10.1162/ARTL_a_00117

Arbib, M. A. (1975). From automata theory to brain theory. *International Journal of Man-Machine Studies*, *7*(3), 279–295. https://doi.org/10.1016/S0020-7373(75)80013-7

Arbib, M. A. (1997). The handbook of brain theory and neural networks. *Neurocomputing*, *16*(3), 259–261. https://doi.org/10.1016/S0925-2312(97)00036-2

Darwin, C. (1859). *On the Origin of the Species*. *Darwin* (Vol. 5). https://doi.org/10.1016/S0262-4079(09)60380-8

De Garis, H. (2002). Special issue on evolutionary neural systems. *Neurocomputing*, *42*, 1–8. https://doi.org/10.1016/S0925-2312(01)00601-4

De Garis, H., & Goertzel, B. (2010). Guest Editorial: Special issue on artificial brains. *Neurocomputing*, *74*(1–3), 1–2. https://doi.org/10.1016/j.neucom.2010.08.007

Dorin, A. (2014). *Biological Bits: A Brief Guide To The Ideas And Artefacts Of Computational Artificial Life*. *Artificial Life*. Retrieved from https://itun.es/au/Jc9TX.l

Evans, R. (2011). Representing personality traits as conditionals. In *AISB 2011: AI and Games*.

Givon, L., & Lazar, A. A. (2016). Generating an Executable Model of the Drosophila Central Complex. *Neurokernel RFC*. https://doi.org/10.1101/051318

Givon, L. E., & Lazar, A. A. (2016). Neurokernel: An open source platform for emulating the fruit fly brain. *PLoS ONE*, *11*(1). https://doi.org/10.1371/journal.pone.0146581

Grand, S., & Cliff, D. (1998). Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems, 57*, 39–57. https://doi.org/10.1023/A:1010042522104

Grand, S., House, Q., Court, M., Cliff, D., & Malhotra, A. (1997). Creatures: Artificial life autonomous software agents for home entertainment. *Artificial Life*, 22–29. https://doi.org/10.1145/267658.267663

Hayman, S. (1999). The McCulloch-Pitts model. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)* (Vol. 6, pp. 4438–4439). https://doi.org/10.1109/IJCNN.1999.830886

Hensel, A. (2005). Conway's game of life. *Conway's Game of Life,* (May). Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Conway+'+s+Game+of+Life#9

Jones, T. B., & Ackley, D. H. (2016). Scalable Robustness. In *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016* (pp. 31–38). https://doi.org/10.1109/DSN-W.2016.44

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Hodjat, B. (2017). Evolving Deep Neural Networks. *arXiv Preprint arXiv:1703.00548*, 1--8. Retrieved from http://arxiv.org/abs/1703.00548

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533. https://doi.org/10.1038/nature14236

Mnih, V., Silver, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning (Dqn). *Nips*, 1–9. https://doi.org/10.1038/nature14236

Olah, C. (2015). Understanding LSTM Networks. *Web Page*, 1–13. https://doi.org/10.1007/s13398-014-0173-7.2

Panne, M. Van De, & Fiume, E. (1993). Sensor-Actuator Networks. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '93* (pp. 335–342). https://doi.org/10.1145/166117.166159

Paszke, A. (2015). LSTM implementation explained. *Web Page*, 1–13. Retrieved from https://apaszke.github.io/lstm-explained.html

Rendell, P. (2013). A fully universal turing machine in Conway's game of life. *Journal of Cellular Automata*, *8*(1–2), 19–38. https://doi.org/10.1109/HPCSim.2011.5999906

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, *21*(4), 25–34. https://doi.org/10.1145/37402.37406

Rojas, R. (1996). Neural networks: a systematic introduction. *Neural Networks*, 502. https://doi.org/10.1016/0893-6080(94)90051-5

Rosenblatt, F. (1960). Perceptron Simulation Experiments. *Proceedings of the IRE*, *48*(3), 301–309. https://doi.org/10.1109/JRPROC.1960.287598

Sarkar, P. (2000). A brief history of cellular automata. *ACM Computing Surveys (CSUR)*, *32*(1), 80–107. https://doi.org/10.1145/349194.349202

Sims, K. (1990). Particle {{Animation}} and {{Rendering Using Data Parallel Computation}}. *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, *24*(4), 405–413. https://doi.org/10.1145/97879.97923

Sims, K. (1994). Evolving 3D Morphology and Behavior by Competition. *Artificial Life*, *1*(4), 353–372. https://doi.org/10.1162/artl.1994.1.4.353

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94* (pp. 15–22). https://doi.org/10.1145/192161.192167

Szigeti, B., Gleeson, P., Vella, M., Khayrulin, S., Palyanov, A., Hokanson, J., … Larson, S. (2014). OpenWorm: an open-science approach to modeling Caenorhabditis elegans. *Frontiers in Computational Neuroscience*, *8*. https://doi.org/10.3389/fncom.2014.00137

Tesla, N. (1919). My Inventions: The Autobiography of Nikola Tesla. *Electrical Experimenter*, 34. https://doi.org/loc?

Turing, A. M. (1990). The chemical basis of morphogenesis. *Bulletin of Mathematical Biology*, *52*(1–2), 153–197. https://doi.org/10.1007/BF02459572

Von Neumann, J. The general and logical theory of automata. In L. A. Jeffress, editor, Cerebral Mechanisms in Behaviour: the Hixon Symposium (1948)

Watkins, C. J. C. H., & Dayan, P. (1992). Q-Learning. *Machine Learning*, *8*(3–4), 279–292. https://doi.org/10.1007/BF00992698

Wexler, J. (2008). Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio's "Black and White" and where it can and will go in the future. In *Spring Simulation Multiconference*. https://doi.org/10.4018/978-1-60960-567-4.ch007

Wolfram, S. (1984). Computation theory of cellular automata. *Communications in Mathematical Physics*, *96*(1), 15–57. https://doi.org/10.1007/BF01217347

Yaeger, L. (1993). Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision and Behavior or PolyWorld: Life in a New Context. *Artificial Life III, Vol. XVII of SFI Studies in the Sciences of Complexity, Santa Fe Institute*, (1), 1–25. https://doi.org/10.1.1.38.6719

Yaeger, L., Griffith, V., & Sporns, O. (2008). Passive and driven trends in the evolution of complexity. *Artificial Life XI: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*, (1994), 725–732. Retrieved from http://mitpress.mit.edu/books/chapters/0262287196chap94.pdf

UNIVERSITY OF COIMBRA Evolving virtual ecology Gloriya Gostyaeva