CHALMERS | GÖTEBORGS UNIVERSITET

GEOMETRICAL AND EVOLUTIONARY EFFECTS IN A PREDATOR-PREY SYSTEM

Bachelor of Science Thesis in Computer Science and Engineering

Loanne Berggren Albin Bramstång
Henrik Ernstsson Hanna Kowalska Elleberg
Erik Ramqvist Sebastian Ånerud

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Geometrical and Evolutionary effects in a Predator-Prey System

LOANNE BERGGREN
ALBIN BRAMSTÅNG
HENRIK ERNSTSSON
HANNA KOWALSKA ELLEBERG
ERIK RAMQVIST
SEBASTIAN ÅNERUD

Examiner: S. A. ANDREASSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Abstract

The area of investigation is the dynamics of a predator-prey relationship. To model the relationship, an agent based model is used as a base for a simulation tool. The tool supports multiple trophic levels, and the agents implement behaviours such as grouping, focusing of a single prey, path finding, and inheritance of the parents properties and mutation to simulate evolution.

The system is used to reproduce the emergence of group behaviour between agents of the same population, as well as to answer how properties of the environment affect the populations. Also, with the same rules for interaction, known physical phenomena such as the capillary effect and oil forming droplets in water is reproduced. From this it is concluded that the size of the environment has a direct impact on the dynamics of the populations. Grouping with agents of the same population is beneficial for both predators and prey. The reproduction of well known physical phenomena provides validation for the model and suggests that the tool could be used for further studies in many different scientific fields.

Sammanfattning

Det som undersöks är dynamiken i ett rovdjur-byte förhållande. För att modellera förhållandet används en agentbaserad modell, som implementeras som bas för ett simuleringsprogram. Programmet har stöd för agenter på olika nivåer i näringskedjan, och agenterna implementerar beteenden såsom gruppering, fokusering på enskilda byten, hitta vägar i en karta, samt arv och mutation för att simulera evolution.

Programmet används för att reproducera uppkommandet av gruppbeteenden mellan agenter inom samma population, liksom för att besvara frågor om hur egenskaper i miljön påverkar populationerna. Med samma regler för interaktionen kan även kända fysiska fenomen som kapilläreffekten och oljedroppar på vatten reproduceras. Med detta som utgångspunkt, dras slutsatsen att storleken på simulationsuniversumet har en direkt påverkan på populationernas beteenden, att gruppera sig med agenter i samma population är fördelaktigt för både rovdjur och byten, och återskapandet av välkända fysiologiska fenomen kan ses som en bekräftelse på modellens giltighet, och tyder på att programmet kan användas för fortsatta studier inom olika vetenskapliga områden.

Glossary

Agent the individual instance of a population that inhabits the universe.

Environment the walls, obstacles and all other things in the universe that interact with the agents.

Evolution a population can change its distribution of traits over time, due to the fact that some traits are more suited for survival than others.

Lotka-Volterra equation an equation describing the relation between a predator and its prey. Displayed as a graph, it shows how the population sizes oscillate.

Obstacle an area in the universe that agents can not move across.

Parameters the different ways the user can affect the simulation and its outcome.

Population a collection of agents with the same traits; a species.

Predator an agent that hunts and eats other agents (prey).

Prey an agent that is hunted and eaten by other agents (predators).

Vegetation/grass the population that the prey eat.

Trophic level the position a certain agent holds in a food chain.

Universe the area in which the agents can act.

Vector a mathematical object with magnitude and direction, often visualised as an arrow.

Contents

This page intentionally left empty

1. Introduction

Ecology is a scientific area where living organisms are studied. The organisms can be divided into different systems based on if they live close to and interact with one another; such a system is called an ecosystem. The term ecosystem can be used to describe anything from a limited lab environment to the whole world, and therefore the subject is interesting for scientists in various areas. Biologists study the behaviour and properties of living things, computer scientists use nature as a of inspiration for problem solving, and mathematicians and physicists create models to help explain complex events.

A frequently studied part of ecology is the relationship between predators and prey. It has been modeled in many different ways, and the most famous is probably the *Lotka-Volterra model*. It describes how a predator population and its prey interact on a macroscopic level, that is with regard to changes in population size. Displayed as a graph, it shows the population sizes oscillating. In addition to the model's obvious biological application, it has also been used to model problems in e.g. economics [6].

The purpose of the project is to create a computer model of an ecosystem with populations at different trophic levels. The population members, or agents, will have the ability to evolve and mutate specific traits. The model will also simulate population behaviour and how the agents move in the environment.

Similar simulations have been created before, and one example is an article published in 2011 where landscape connectivity was being investigated. The authors used an agent based predator-prey model to study behaviour in a fragmented environment with patches of land and ways in between to move on. They stated that because of increasing human populations and global changes, it is important to understand the "consequences of habitat fragmentation" in order to conserve biodiversity [1].

1.1. Objective

The objective is to design and implement a system that simulates an ecosystem, with populations that interact and evolve. The populations take the role of either predator or prey, or both. They also have features such as grouping, focusing on a single prey and finding the shortest path around an obstacle. The populations depend on a wide range of parameters, which determine the predator-prey interaction as well as the interaction between agents of the same population. The population sizes in the model oscillate in a similar fashion as the fluctuations in the Lotka-Volterra model, although stochastically. The simulation environment can be customized by adding obstacles or changing its size or geometrical form.

The design of the application is user friendly and modular as described in the section for requirements. The application is designed to be easy to understand, and the data from the simulation is presented in graphs with clear descriptions. Users are able to customize

many of the features of the populations, to test things not covered in this project. This can be useful for scientists such as biologists, physicists and mathematicians, that perform research within ecology, population behaviour and physical phenomena.

Based on the simulations generated by the application, there will be an attempt to answer the questions stated below. If the questions can be answered, the results could be interesting as a subject for further studies.

1. Is the population behaviour different in rectangular and elliptical environments, and if so, in what way?
2. How does a decrease in the inhabited area's size affect the outcome of the simulation?
3. Does the presence of obstacles interfere with the populations, and if so, in what way?
4. Can the emergence of group behaviour be reproduced by evolution?
5. What is the most advantageous of a compact and a scattered group?
6. Can the realism of the interactions in the system be validated by reproducing already known physical phenomena?

1.2. Problem

The main problem is to create an application that simulates a predator-prey relationship, where the interactions are modelled in a realistic way. There are many variables to be taken into account, and the variable space is both large and sparse. This is a commonly known problem when working with these kind of systems and should therefore be taken seriously [11]. Therefore, a considerable amount of time is required in order to tune the variables, to achieve realistic behaviour. Realistic would here be achieved by creating a predator-prey system where the populations manage to survive and move in a way which is visually satisfying and does not stray too far away from behaviour found in animals. For example a single prey should not suddenly turn back and fight against the predator, it should instead try to run away.

In order to obtain realistic behaviour, the populations could be inspired by existing models such as:

1. Lotka-Volterra equation for predator-prey interaction and more realistic extensions.
2. Evolutionary algorithms, such as Genetic Algorithm.
3. Simulation models for animal grouping.

Another issue is how to validate the results produced by the system. Therefore, comparisons with existing models can be done to verify correctness and realism. This implies that the models used for verification have to be found, studied and of course

realistic themselves. Concerning the user interface, the main problem will be to validate if it is intuitive and user friendly. This can be partly avoided by having a design consistent with existing conventions and through user feedback.

1.3. Constraints

There are hundreds of models to explore in the field of population-, diffusion- and evolution theory. Seeing as this project has a short time limit it is not possible to explore and implement all of them. Therefore, this project will only focus on the models and areas previously listed, which have been selected based on their level of difficulty, relevancy to this project and the interests of the authors.

Regarding evolution, this project will only focus on genetics at an abstract level. Genes will be modelled as binary or numerical values, which describe their impact on the gene. There will be no attempt to model the chemical details of DNA on a molecular level, nor will the genes have the property of being recessive or dominant, instead all genes will be equally dominant.

The simulated environment is a two-dimensional representation, which leads to simplifications regarding the agents' movement. Therefore, the agents are unable to interact on different heights. An example of this would be birds flying over agents and not interacting with them. With two dimensions the birds are forced to interact with agents living on the ground level.

2. Methods

The project was done in weekly iterations, with clear goals for each week and member, according to the agile development method [8]. Because of the iterative development the phases of the project are not clearly distinguished from each other; some literature studies have been done during implementation, as well as some analysis. Additional literature studies were undertaken when new information was needed in order to solve a problem.

The implementation phase was the longest, since it required a lot of time and was combined with the other phases, as mentioned above. During the last three weeks, the focus was almost exclusively on analyzing the results from the simulations and finishing the report.

2.1. Work structure

The first step of the project was to gather and read relevant literature, to get a better sense of what had already been done in the area and to find examples of algorithms that could be used to simulate the agents behaviour as realistically as possible. Models for predator-prey relationships were of particular interest because of existing knowledge within the group. A brief study on design of graphical interfaces was also undertaken, to ensure better design choices.

The information was retrieved from the Chalmers library, as well as from various Internet sources. However, since the project was mostly exploratory, the literature studies have not been very extensive compared to the other phases. Instead, the group focused on implementing own ideas, inspired by previous knowledge.

2.2. Pre-study

The project was developed using Java, since this is the programming language the group members were best acquainted with. However, the choice of programming language was not obvious, since it was known from the start that the program would require a lot of complex and time consuming computations. For this purpose a low-level language would be more suitable, but it was concluded that learning a new programming language would steal too much focus from the original objective.

To allow multiple users to edit the code at the same time, the distributed revision control system *Git* was used. For this project, it has proven to be of great importance to have a well functioning revision control system, since most of the project members have worked and experimented in their own branches.

2.3. Implementation

To enable data collection from simulations, multiple real time graphs are used. They are described more thoroughly in section 3.2.2. To gather a sufficient amount of data, enough simulations had to be done to get trustworthy results. The group decided that if a simulation run 5 times with consistent results, for each independent simulation, it is sufficient to be able to draw conclusions. With the same reasoning it was concluded that each simulation should be run for at least 100 000 iterations. The reason for limiting the iterations and number of simulations in this way, is due to the considerable amount of time it would otherwise take to get the results. Depending on the experiment, more iterations might be needed to get the desired result. Examples of such experiments involve one or multiple genes evolving in a population over a longer time.

The work was divided between the group members in three different categories. Two members looked at the effects of changing the shape and size of the simulation universe, two looked at different effects of evolution, and the last two group members looked at the effects of adding different obstacles to the universe. All simulations were run with the same default settings, where only the settings relevant to the investigation area were changed, to ensure that the results could be compared.

3. The application

The application is a simulation tool, in which populations can be studied in various environments. A user can change the environment, modify different parameters, and follow the outcome on the screen. The outcome is presented on a 2D map, together with multiple graphs that display relevant data.

3.1. Requirements

The requirements are divided into categories according to the FURPS-model; Functionality, Usability, Reliability, Performance and Supportability. The separation provides greater detail and a better overview, to help the developers to make sure all parts are covered [7].

3.1.1. Functional

The functional requirements decide what features the application will have. For this project, these are mostly focused on what sort of parameters that can be set for each simulation.

The user will be able to start, pause and restart a simulation. The simulation can also be recorded, giving the opportunity to replay it for further analyzation to find interesting effects and behaviours. Before the simulation is started, the user will be able to choose how the simulation should run by setting different parameters - such as shape of universe, population size and iteration time. While the simulation is running, real time graphs will portray how the populations change and a heat map that display where in the universe the agents tend to be. The user can also change some of the settings when a simulation is already running.

Summary
Settings
The application shall allow the following settings to be changed before running a simulation:

- Shape of the universe.
- Simulation window dimensions.
- What obstacles, if any, to include in the simulation.
- Which prey population(s) to include in the simulation.
- Which predator population(s) to include in the simulation.
- Which vegetation population(s) to include in the simulation.

- Population sizes.
- Iteration delay.
- Limit the simulation to a specific number of iterations.
- Amount of concurrent working threads.
- Record simulation.

Simulation Controls
The application shall allow the user to:

- Start a simulation.
- Pause a simulation.
- Restart a simulation.
- Save a run.
- Replay a run.

Output representation

The application shall display:

- The simulation on a 2D map.
- The mean life length of the populations in a graph.
- The population sizes in a graph.
- The number of iterations the simulation has been run in a graph.
- The proportion of grouping behaviour in a population in a graph.
- A heat map for each population giving information where the agents tend to be most in the simulation universe.
- Statistics after a run.

3.1.2. Usability

Since the application is supposed to be used by users with different background, focus has been on making the user interface easy to understand and navigate. The user should not be overwhelmed by choices before starting a simulation, but should be able to get acquainted with the application in his/her own pace. Therefore, the settings menu can be found in a separate window accessible from the menu in the main window. Also, the advanced settings are not visible at first, the user has to explicitly choose to change them. If the user wants to run the simulation without changing any settings, he/she can simply run the simulation with the default settings. Since the application is meant to be

relatively easy to extend, special attention has been paid to the structure and readability of the code.

Summary

- The application shall be easy to understand and navigate.
- The design shall abide by existing design conventions, if there is no good reason to do otherwise.
- The code shall be easy to understand, with descriptive names for variables and methods, clear structure and java documentation.

3.1.3. Reliability

The simulation should be able to run with a large amount of agents, without throwing exceptions or crashing. Expected exceptions should be handled properly, and the GUI should run on a separate thread allowing it to always respond to commands and never freeze.

3.1.4. Performance

Because the system allows a user to specify the number of agents and how fast each iteration should take, the performance can vary. If there are too many agents the CPU might not keep up, and the graphical representation may seem to lag behind.

Most modern computers have several processor cores. This has been taken into account, and the system must be developed to always keep the cores busy, as running all the algorithms in one thread slows down performance.

Java might not be the most performance optimized language, but due to optimized code and multithreading, the system should be able to run with a large amount of agents, without causing the program to slow down.

3.1.5. Supportability

The application should be built in a modular way, so that it easily can be extended with new different behaviour and algorithms. Since Java is used, our system will be supported by Linux, Windows and Mac. Some unit tests should be made in order to minimize bugs that may occur later on. Other benefits with Unit testing is that problems can be detected early and it simplifies the integration with the rest of the system. The Java testing library called JUnit will be used for Unit tests.

3.2. User Interface

The user interface is designed to be easy to use for a wide variety of user. This is done partly by following design conventions, such as grouping things that belong together, implementing keyboard shortcuts and tool tips, and "hiding" settings that are more advanced, so less experienced users can use the application without feeling hindered [2].

3.2.1. Settings

The settings menu consists of two parts, a window with the basic settings, and a window with advanced settings, that can be reached from the first one. The basic settings include setting the shape of the universe, set different maps with obstacles, choosing which populations to include and their initial size, as well as if the simulation should be recorded or not. The advanced settings allows the user to limit the number of iterations, set a delay between iterations, change number of working threads and set the dimension of the simulation universe. The simulation is activated from the settings menu.
Figure 3.1 shows the settings menu.

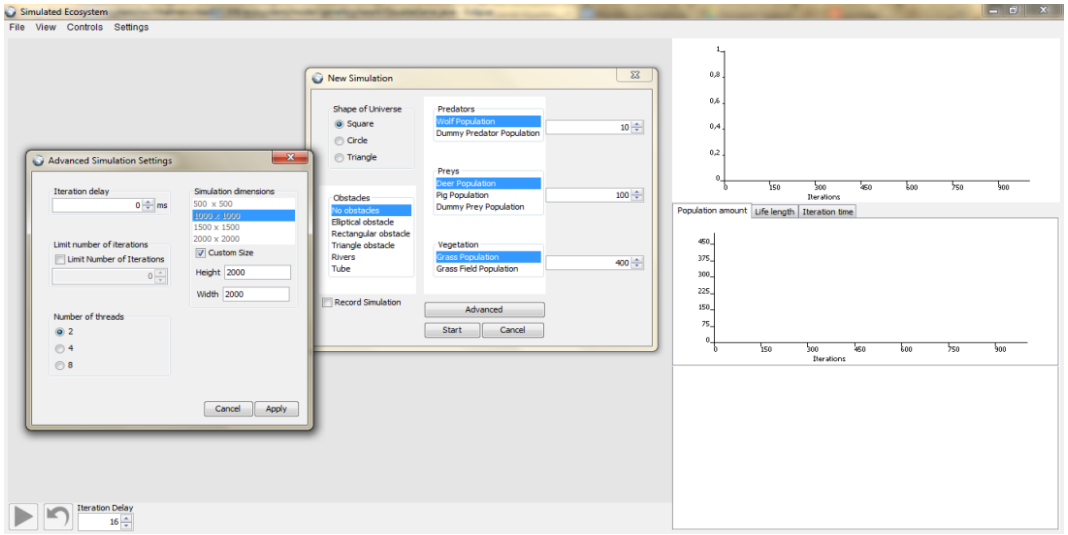


Figure 3.1: *The application interface with the settings menu open and no simulation running.*

There is also another settings menu, called Population settings, that allows the user to make much more sophisticated changes with regard to the agents behaviour. It is displayed in Figure 3.2. With this menu, the experienced user can make advanced changes to affect the behaviour of the agents that constitutes the different populations. The user can e.g. set the vision range or the maximum speed of the agents in a certain population. The genetic settings can also be changed, for instance if a behaviour such as

grouping should be active at birth and/or be mutable. The Population settings menu is not described in the requirements section, since it was added at a late stage to facilitate the large amount of simulations that was needed for the analysis phase.

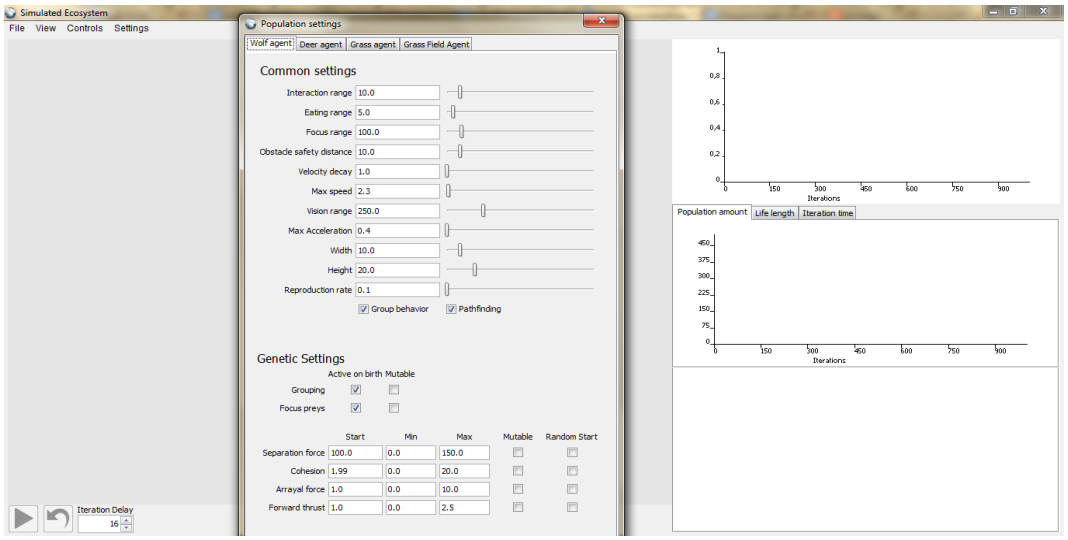


Figure 3.2: *The application interface with the settings menu open and no simulation running.*

3.2.2. Simulation and simulation output

The outputs from our application will be represented in a number of ways. The main one, that is the focus of the application's interface, is a simulation of all the active agents, their movements and interactions. The interface with a running simulation is displayed in Figure 3.3. The predators are represented by red arrows, the prey by blue arrows, and the grass by green dots. The agents can also be colour coded differently to show that the agents have different genetic traits. The simulation can be paused and unpaused using the button at the bottom of the window. The population can also be restarted with the same settings (but with different initial positions for the agents) from another button next to the first.

The results are also made visible through a number of graphs, which are placed to the right of the simulation panel. At the top is a graph that displays the grouping proportion for the populations. Below are three graphs in different tabs: the first one shows the populations sizes, the second one the mean life length of the agents in a population, and the third one simply shows for how many iterations the simulation has been run. At the bottom is a heatmap that also has three tabs, one for each active population. The heatmap shows where in the simulation universe the the populations have been most during the simulation.

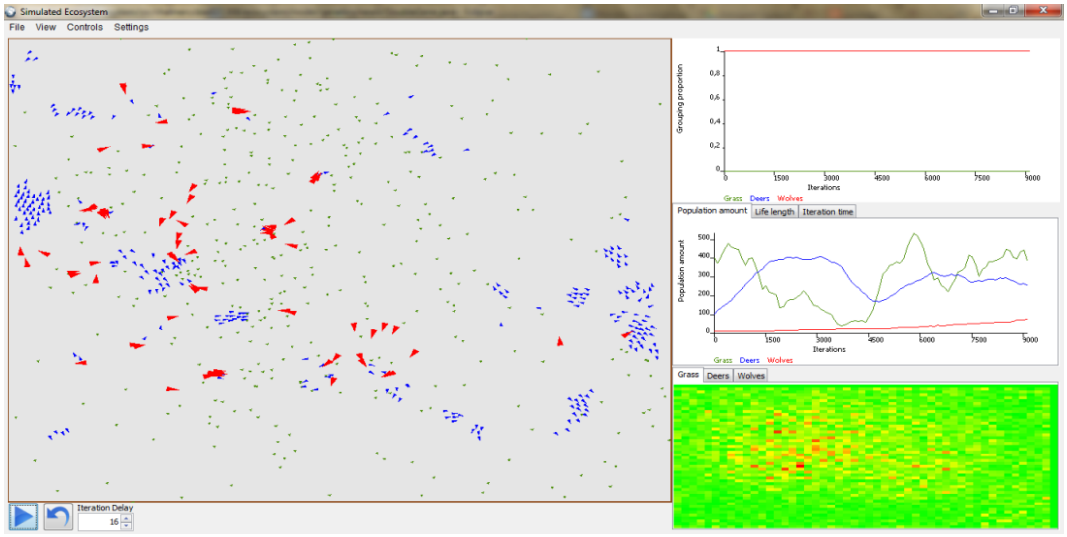


Figure 3.3: *The application interface with a running simulation. The large panel to the left shows the agents in action. To the right are two graphs displaying the population mean life length and the population sizes, and below them is a heat map that shows where in the universe the different populations tend to be the most.*

3.3. Architecture

Since the application should be modular and have the possibility to be extended, it will follow the Model-View-Controller pattern (MVC). MVC separates the system into three parts, that each have its own area of responsibility [4]. This allows us to make changes in the model without affecting the presentation layer. To keep the communication between the model and view simple, they implement the Observer pattern. Which notifies the listening view when the model changes its state [9].

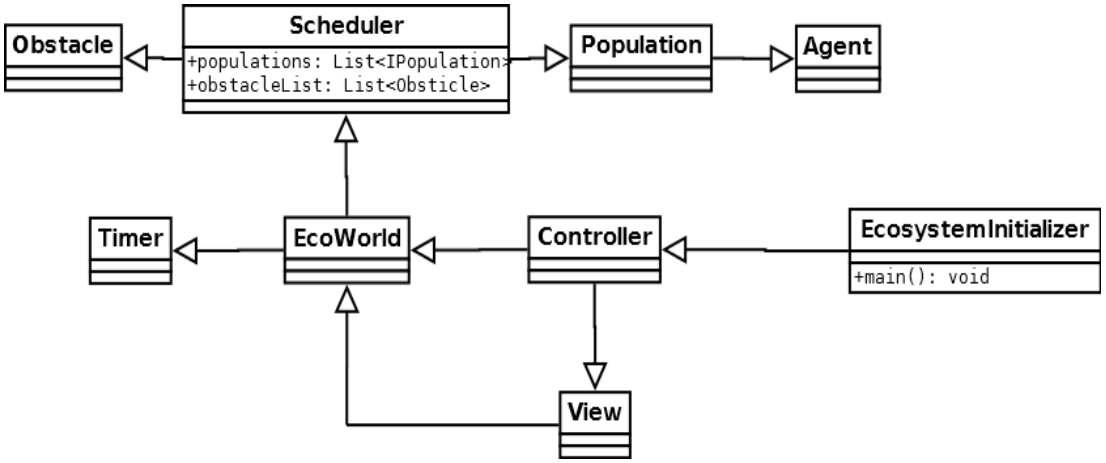


Figure 3.4: *A simplification of the abstract architecture of the system. The model is clearly independent of the controller and the view, as defined by the Model-View-Controller design pattern.*

3.4. Complexity and Performance

The time it takes to compute one iteration can vary between 1 ms to about 100 ms. To allow an even amount of iterations, a Timer class is used to solve this problem. Its purpose is to slow down fast iterations, so that every iteration takes about the same amount of time. The Timer class can be set to an arbitrary value for speeding up or slowing down the simulation. The result of this is a smooth graphical playback.

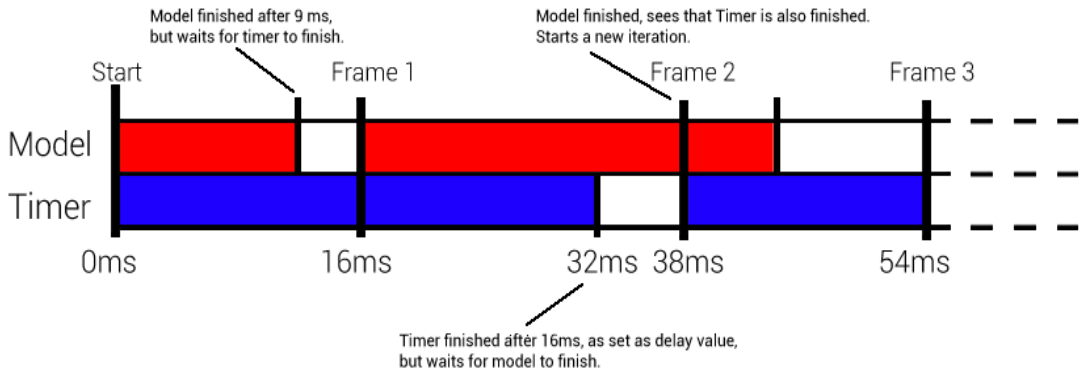


Figure 3.5: *Algorithm for synchronizing the model and the Timer.*

The delay and the updating of the model is done in parallel, as shown in Figure 3.5. If the model finishes one iteration before the Timer expires, the model has to wait until it starts a new one. E.g., if the delay is set to 16 milliseconds, it will have a iteration rate

of 60 iterations per second. The minimum time each iteration will take is 16 ms, while there is no limit on maximum iteration time.

3.4.1. Multi-Threaded for performance

To improve the performance, multiple threads are used in almost every part of the program. The view has its own Swing/AWT threads and the model has its own thread pools. The gains of using a multithreaded systems are great and outperforms a single-threaded one. For updating the model, two Java Thread pools are used, T1 and T2, as shown in Figure 3.6. All threads in the thread pools are reused to improve performance. The number of workers in T2 can be set to an arbitrary value, default is 4 worker threads.

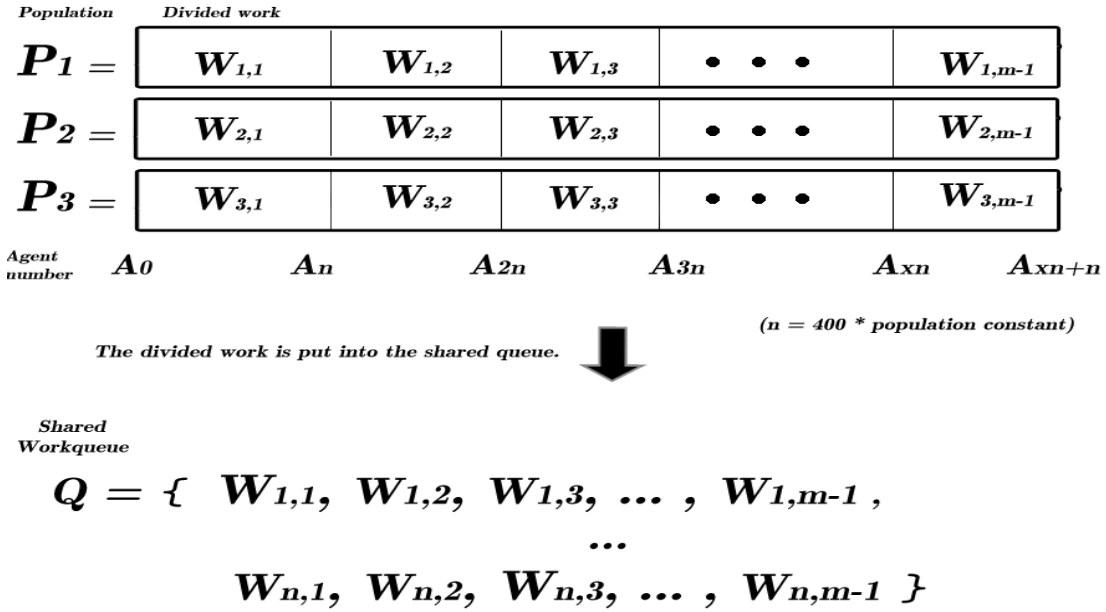


Figure 3.6: *The solution for dividing the populations to a shared work queue.*

When one iteration needs to be calculated, T1 divides every population into small work pieces, each containing about 400 agents, and puts them on a shared thread-safe work queue, as shown by Figure 3.6. T2 then puts all of its worker threads to execute the work queue until it is empty. Meanwhile T1 waits for the work in Q to be finished, using a concurrency principle called barrier synchronization.

T1 only consists of one worker that receives signals from the Controller. It also synchronizes the timer and T2 for smooth simulation, and notifies T2 when to calculate the next iteration. For every iteration, T1 divides one model update into small pieces, puts them into the work queue and tell T2 to execute them with all of its threads, and finally waits for them to be finished using barrier synchronization.

3.4.2. Taking advantage of the GPU

To improve the performance even further, the system takes advantage of the unused graphics card for the graphical representation of agents and obstacles, instead of having the CPU taking care of the drawing. This allows the CPU to focus on other important computations.

The cross-platform graphics library OpenGL is used for handling the drawing. OpenGL is written in C, so to be able to communicate with OpenGL, a library called JOGL is used as interface between the Java system and OpenGL. JOGL only elevates the method-calls to the corresponding C method. The methods used in Java is exactly the same as in the C version of OpenGL.

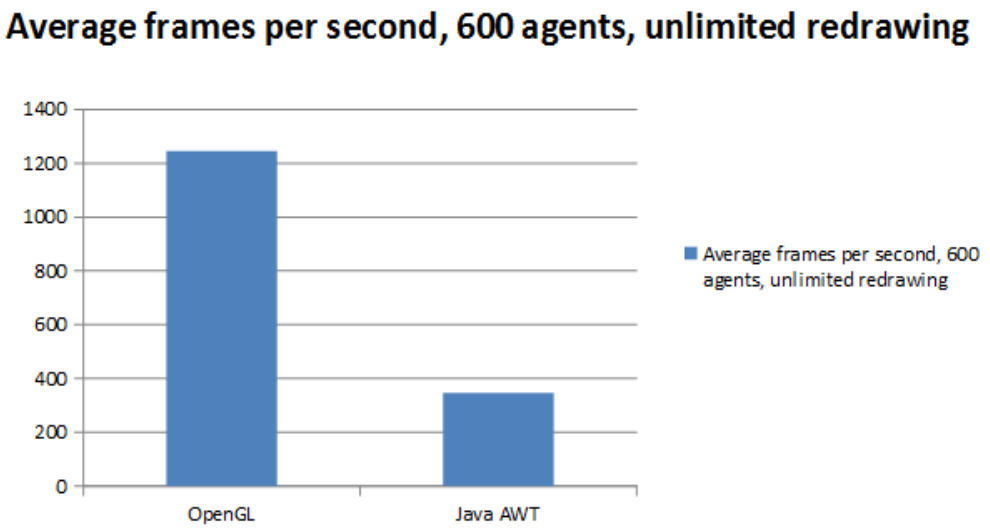


Figure 3.7: *A benchmark between OpenGL and Java AWT.*

To prove this is a good method, a benchmark of the Java AWT software rendered graphics and the OpenGL rendered graphics was created. Two versions of the same simulation frame, each containing the same algorithm for drawing agents, and the output of them then looks almost the same. A benchmark was created to study performance differences, the result is shown in Figure 3.7

3.4.3. Reducing unnecessary computations

To reduce time spent calculating unnecessary forces, the system has a feature called neighbour list. This list contains the agents that are closest to one and another. When the calculations of forces is done, only the agents in the list is taken into account.

Every agents neighbour list is updated every 10th iteration, with a random start value, of 0 to 10, so that not all agents updates its neighbour list at the same time, as this may be too heavy for the CPU. For improving the performance even further, a limited size priority queue is used, with a maximum amount of 20 agents and with distance from the current agent as the priority value.

4. The Simulation

The simulation is a central part for this project; seeing as it provides all of the results in focus of this report. The simulation consists of a number of agents from different populations that use forces to interact with each other. Calculations are done individually to simulate how each agent represents a single individual. This chapter describes the agents involved.

In the simulation an iteration is defined as a discrete time step where the agents' positions are updated. In each iteration agents can die due to starvation, old age or predators eating them. Also, new agents can be born if the parent has eaten in the current iteration. Each iteration is used as the starting point for the next iteration.

4.1. Populations

The agents are gathered in populations, representing the organisms they mimic. There are three types of populations in the current implementation: predator, prey, and vegetation. Apart from the grass field, agents are visualized by coloured arrows facing the direction of the agent, and where the colour and size of the arrow both depend on the type of agent, as well as the evolutionary traits it possesses. The grass field is instead drawn as a green circle, to better represent a big field of grass.

The forces that dictate the behaviour of the agents are calculated in the same way for all types of agents. However, there is a difference in how the forces affect each agent, depending on which population the agent is a member of. The forces determining the different behaviours are listed below. A more detailed explanation behind the forces can be found in Appendix A.

- F_{pred} - A The force an agent feels from present predator.
- F_{prey} - The force an agent feels from present prey.
- F_{Fenv} - The force an agent feels from the environment (walls and the obstacles).
- F_{mi}- The force an agent feels from interacting with nearby neutral agents (agents on the same trophic level). The agent is subject to attraction and repulsion from other individuals in a group.
- F_{array} - The tendency for an agent to equalize its velocity with nearby neutral agents.
- $F_{forward}$ - The tendency of an agent to keep moving in its current direction.'
- F_{rand} - The force describing the behaviour of an agent that cannot be explained properly and is therefore said the be random.

The first two forces determine the interaction between predators and prey. The environment force will make sure that no agent is on a position where it cannot be, i.e. outside the 2-dimensional environment or inside an obstacle. The mutual interaction force, the arrayal force and the forward thrust are forces that determine how the agent behaves in the presence of other agents of the same population. These are the forces that will determine if the agent wants to group with other agents. The forces for grouping are all taken from the book on ecological diffusion by A. Okubo [10]. The random force is a force that the agent is affected by that can not be properly explained and is therefore said to be random. This could for example be the error the agent does in estimating in what direction it should go.

4.1.1. Predator

The role of the predator is to hunt and eat prey. In the current implementation the predator population is named "wolves" but is not necessarily implemented to mimic the behaviour of real wolves. The predator, being on top of the food chain, need not to worry about being eaten, instead they focus on hunting the prey that are below them in the food chain. In order to hunt nearby prey they will first try to minimize their distance to all of them. When a predator finds a prey to be close enough, it will put all its focus on catching that single prey. If one predator catches a prey, only the predator that caught the prey will be able to eat it. The predators use a method called path finding (see appendix A.5) to calculate clever paths to their preys.

4.1.2. Prey

The prey have the the most complex job situation in the simulation. They have to find vegetation to eat, so they can replenish their energy, and reproduce, at the same time as they are being chased by the predators. A great dilemma for the prey lies in deciding which direction they should flee. In the event of being hunted by a single predator, the best would be to flee in the direction straight away from the predator. If there are several predators closing in on the prey, from different directions, the best way to flee is less obvious. In the current implementation the prey tries to maximize the sum of distances to all nearby predators. The distance from a closer predator is weighted higher than that of a predator further away, to simulate the higher threat of a nearby predator.

4.1.3. Vegetation

There are two types of vegetation representing the lowest part of the food chain: simple grass straws, and fields of grass. Both types have the same task: providing food to the organisms one trophic level above them, but the way they operate and interact with other agents is quite different.

The first type of grass, the straws, spreads into a wider and denser area, up to a space limit. With a high consuming pressure from the prey, this type of grass may be completely eaten, leaving the prey to starve.

The second type of grass, the grass fields, are larger than other agents, representing that grass is often not a small patch, but rather covers a large area. The fields store a larger amount of energy and, unlike the simple grass, will never be extinct, instead a period of regrowth is required, in which the energy is increasingly restored allowing, once again, to be fed upon.

4.2. Evolution

At the moment the agents do not breed, instead they reproduce through cloning. During a cloning there is a small probability that a gene mutation may occur. The mutation (algorithm inspired by Wahde M. 2005 [13]) is simulated by randomizing a number **r**. If **r** is less than the mutation probability p_{mut} , a mutation will occur. There are two types of genes. Type 1 is binary, and simply indicating whether the individual has a certain behaviour or not. During a mutation the value switches. Type 2 is a numeric value used for deciding the impact the gene has on the behaviour. For type 2, **r** is generated and compared to p_{mut} for every bit used representing the numeric value, and mutation occur through flipping the bit.

Due to the fact that prey and predators are called deer and wolves, respectively, in the current implementation, the choice of genes were inspired of the behaviour of the real animals. The predators have genes for grouping, including separation factor, cohesion, forward thrust and arrayal force. They also have the ability to focus on a single prey. The prey have, as the predators, genes deciding group behaviour, as well as genes for stotting behaviour. The stotting feature has attributes such as a range deciding how close a predator may come and a length determining how long time the prey will stot. For further explanation and calculations, see Appendix A.

The vegetation has no specific genetic material.

4.3. Obstacles

In the system there is support for adding obstacles to the simulation environment. The user can via a map editor (see Appendix A.7 for more details) customize maps by placing obstacles of different shapes at any location on the map. There are three shapes of obstacles: triangular, rectangular and elliptical.

Some simplifications were made to the triangles in order not to spend too much time on the obstacles. The triangles are all isosceles and are parameterized with a width and height. The rectangular and elliptical obstacles are also represented by a width and a

height. All obstacles can be rotated in any angle, which enables complex structures of the inhabited environment.

One purpose with obstacles is to have the ability to create and simulate situations that are likely to occur in real life. Adding a rectangular obstacle to the middle of the inhabited environment can simulate the deforestation of a large area, in an already existing and well balanced ecosystem. The simulation might suggest that the deforestation of a large area would set a real ecosystem out of balance, given the same population dynamics.

4.4. Shapes

There are three different shapes for the simulation universe: square, circle and triangle. When the shape is changed, the area of the simulation universe is also affected, i.e. the area of a square of the size 1000*1000 distant units is actually bigger than the area of a 1000*1000 circle. (The distant units refer to the the width and height of the universe.) This is vital to remember when doing comparisons of the outcome with different shapes.

The objective for the shapes is to make different environments for the agents to move around in. This is done by giving the agents the positions of the left, right, bottom and top edges of the world they live in so they can use this information in order to calculate the environmental forces. These positions depend on the shape and size of the environment as well as current position of the agent.

For the square, finding these positions is very straightforward. They are a direct result of the size of the environment, because the square shape covers up the entire environment. For the circle and triangle it is a basic case of using trigonometry to find the edges, given the current position and the size of the environment. For further explanation and calculations, see Appendix A.

5. Results

The result produced by the application contains information about: the amount of agents in each population, the average life length of agents in a population and the population diversity, i.e how many in each population has a certain evolutionary trait. Analyses of the agents' movement in the environment were also made by viewing the heat map. The results are interpreted based on information visualized by graphs.

All of the results are compared to the environment created with default settings. With the default settings the environment was rectangular and its size was set to 2000*2000 distance units with no obstacles in it. The predator population was initialized to 50 agents, the prey population to 250, and the grass population to 600. The predator and prey were set to group up without the possibility to evolve or mutate.

5.1. Simulating with different shapes

For the purpose of investigating what effect the shape of the simulation universe has on the outcome, the simulation was run 5 times for at least 100 000 iterations with the square and circular universes respectively. Since alternating the shape also affects the size, the simulation dimensions had to be set so that the area was the same for all shapes. The square universe was set to 1772*1772 units and the circular to 2000*2000 units.

When simulating with a square universe the agents were concentrated at the left 6 out of 10 simulations. In 3 of 10 they were spread out over the universe, and only in 1 out of 10 were they concentrated at the right. When simulating with the circular environment, 9 out of 10 cases, the agents were concentrated to the right, and in 1 of 10 they were to the left. The simulation outputs are presented in Figure 5.1 and 5.2.

The predators move over a smaller area than the deer, for simulations with both the square and the circle universe. The results are presented in Figure 5.3 and 5.4. There are no differences in how the population sizes oscillate.

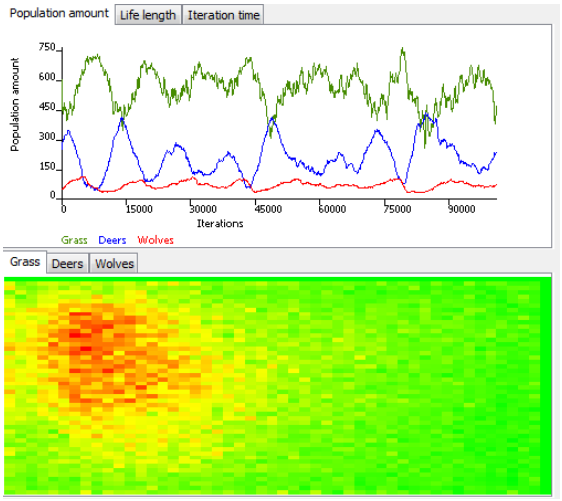


Figure 5.1: *The output for a simulation run with the square universe. The agents are mostly at the left of the universe.*

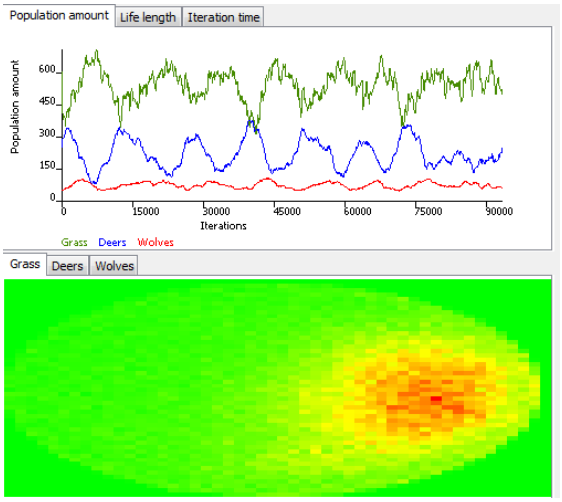


Figure 5.2: *The output for a simulation run with the circle universe. The agents are mostly at the right of the universe.*

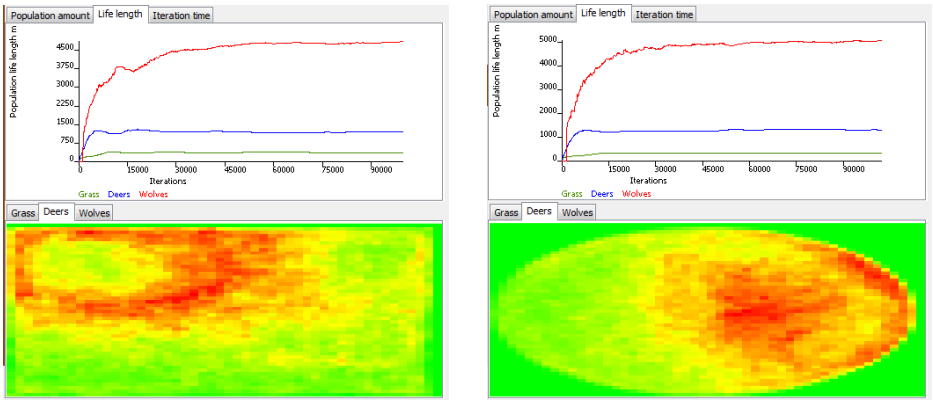


Figure 5.3: *The deer population move over a large part of the universe. This is the case with both the square and the circle.*

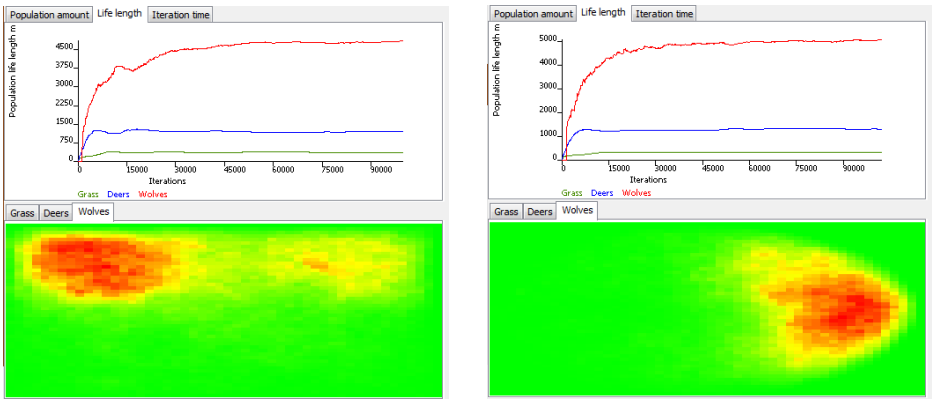


Figure 5.4: *The wolf population is concentrated in a small part of the universe. This is the case with both the square and the circle.*

5.2. Simulating with different sizes

For the purpose of investigating what effect the size of the simulation universe has on the outcome, the simulation was run 5 times for at least 100 000 iterations with the four standard sizes available in the settings menu, that is 500*500, 1000*1000, 1500*1500 and 2000*2000 distant units, and the custom size 4000*4000 units. All simulations were run with the square universe.

With the universe set to 500*500 distant units, the prey ate all the grass 7/10 times, resulting in the extinction of all the populations almost immediately. The outcome can be seen in Figure 5.5. The other 3 times, the predators ate all the prey just as fast, which led to the grass being the only population left.

Figure 5.5: *The simulation run with the size set to 500*500 distant units, an example when all the populations go extinct.*

With the universe set to 1000*1000 distant units, the populations survived through around 10 000-25 000 iterations, but the simulations invariably ended with the grass being the only population left, as seen in Figure 5.6.

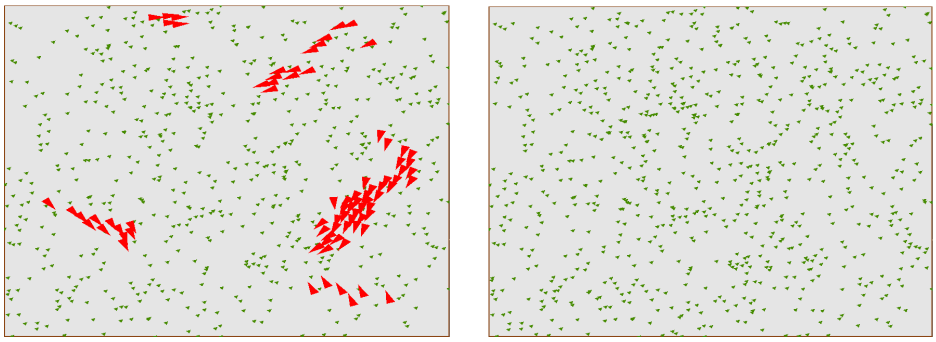


Figure 5.6: *The simulation when the grass is the only agent left, happens sometimes when the size is 500*500 and for all the runs with the size set to 1000*1000.*

The results of the simulation universe set to 1500*1500 distant units did not differ from when the size was set to 2000*2000 units, at least not in a way that can be seen in the simulation output, see Figure 5.7. It was slightly more unstable, with the predators

eating all of the prey once, at around 20 000 iterations, which led to grass being the sole survivor in the environment. The heatmaps differ slightly between the sizes in respect to how much area the populations are concentrated on.

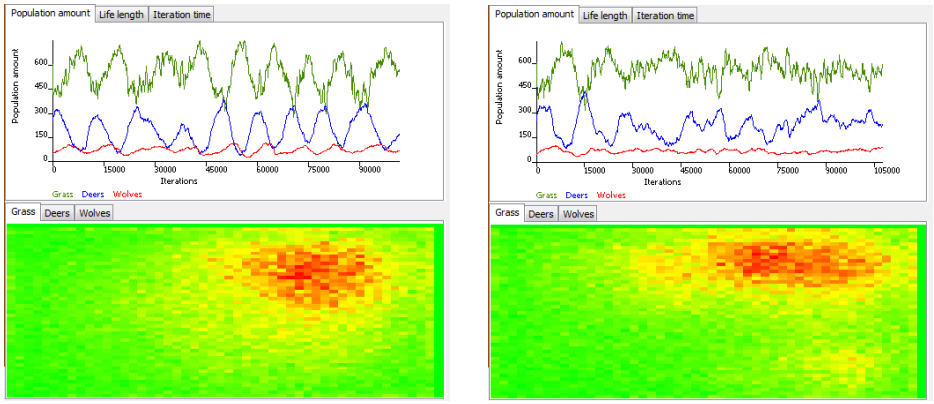


Figure 5.7: *To the left, the size set to 1500*1500, and to the right, the size set to 2000*2000.*

When simulating with a larger environment, 4000*4000 distans units, it looks a bit different compared to the smaller sizes. The results are shown in Figure 5.8. The difference between the amount of agents in each population is significantly changed compared to a 2000*2000 units big environment. There are more grass agents and more predators, but not that much more prey.

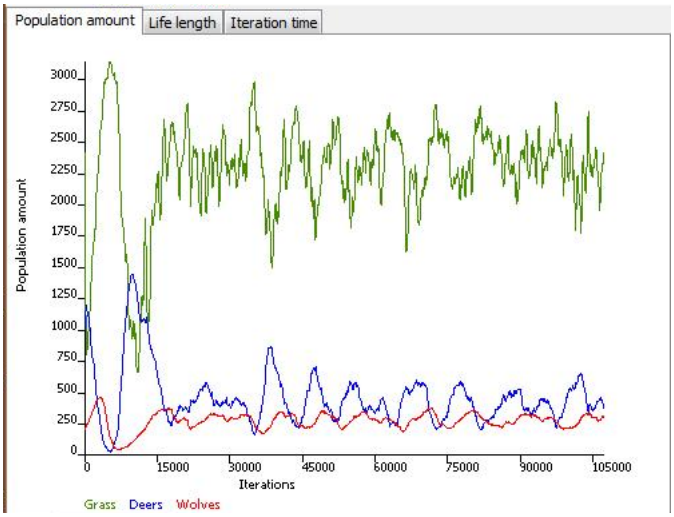


Figure 5.8: *The population amount with the size set to 4000*4000.*

There is also a difference in the actual life length of the prey when simulating with the larger environment, see Figure 5.9. For smaller sizes it ends up around 1500 iterations, but in the big environment they don't manage to stay alive as long and the average life length is around 700 iterations

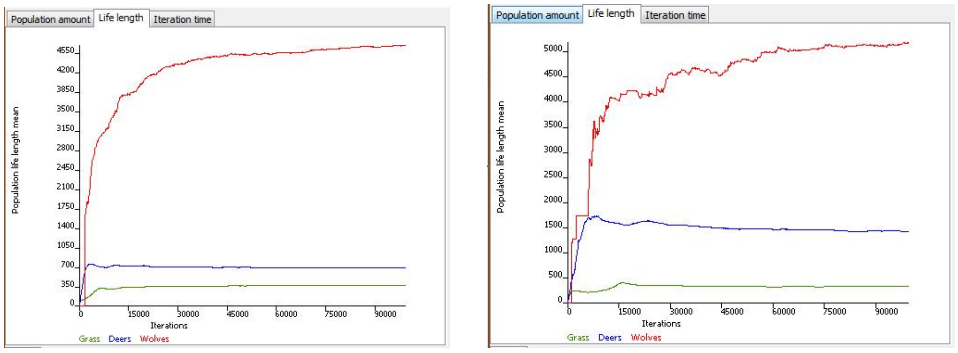


Figure 5.9: *To the left, the size set to 2000*2000, and to the right, the size set to 4000*4000.*

In one of the five cases the predators died out, which led to there being more prey. The prey and the grass then lived on together, but with more oscillations, as seen in Figure 5.10.

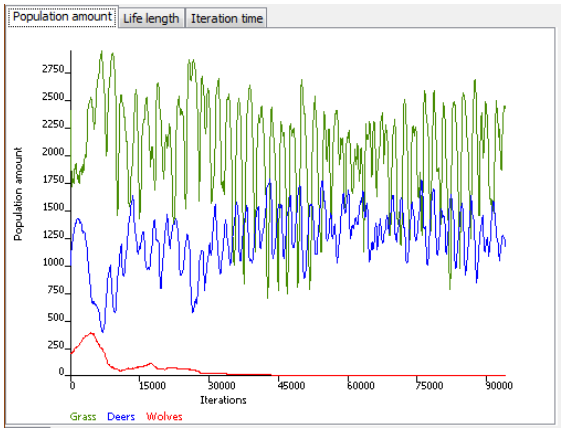


Figure 5.10: *The population amount with the size set to 4000*4000 when the wolves died out.*

5.3. Simulating with different maps

To test the effects of obstacles the map editor was used, and it was a fast and simple way to create different maps. They were created both manually and randomly to see if interesting behaviours could be found, and the results are presented here.

5.3.1. Manually created maps

A total of 8 different maps were used in this experiment, and each map were run 5 times for 100 000 iterations. An empty map without obstacles was also used as a default run. The populations used were grass, prey and predators.

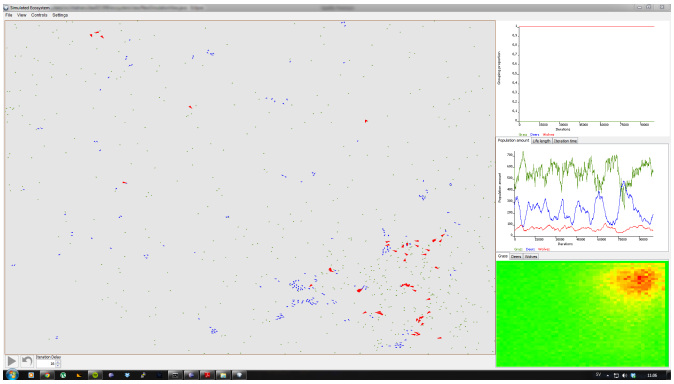


Figure 5.11: *The default run without obstacles.*

At the beginning all three populations were randomly distributed on the map, but as the simulation progressed a high density area in the top right corner formed. It happened because there were a lot of wolves there, so the grass could grow freely without being eaten. When the grass on the rest of the map had been depleted, the deer moved to eat the grass by the wolves. This behaviour resulted in feeding the wolves, without them having to move, so they remained on the grass. When some deer had been eaten, grass started to grow everywhere again which made the deer population increase, before they once more had to return to the wolves. These behaviours looped over and over again, creating oscillations in each population's size.

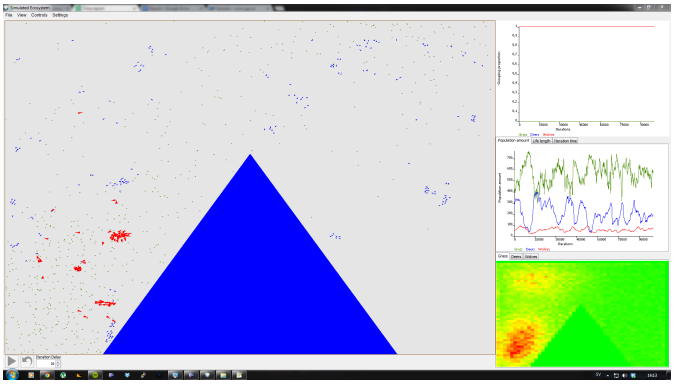


Figure 5.12: *An obstacle in the middle which divides the map.*

The map in Figure 5.12 provided a result which differed slightly from the default run in Figure 5.11. The spot with grass and wolves formed on one side of the triangle, and the

surrounding deer moved on the remaining area. The population size graph does not show significant differences compared to the default run, and all populations survived without difficulty.

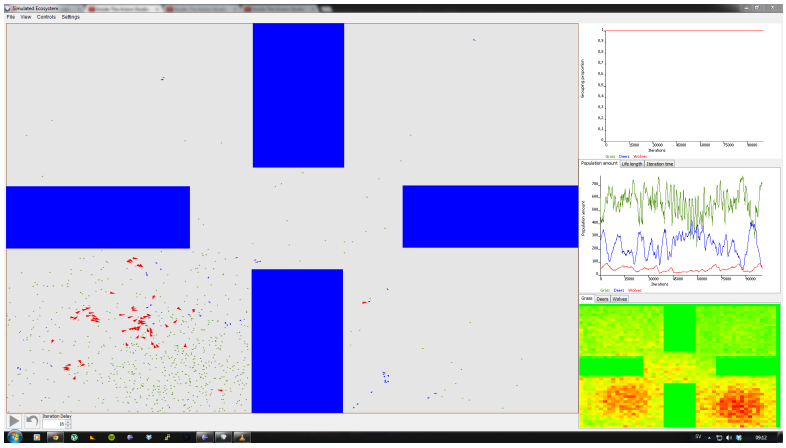


Figure 5.13: *This map is an example of a fragmented environment. The heat map displays the grass population, and it shows that the grass moved around a lot.*

The fragmented map in Figure 5.13 shows that the populations had difficulties living in small areas. The grass and deer filled a box quickly, but they were eaten faster than they could reproduce, leaving the box suddenly empty. Some moved on to reproduce in new boxes and the behaviour repeated. The population size graph shows a higher fluctuation rate than the default run, especially for the grass and prey population.

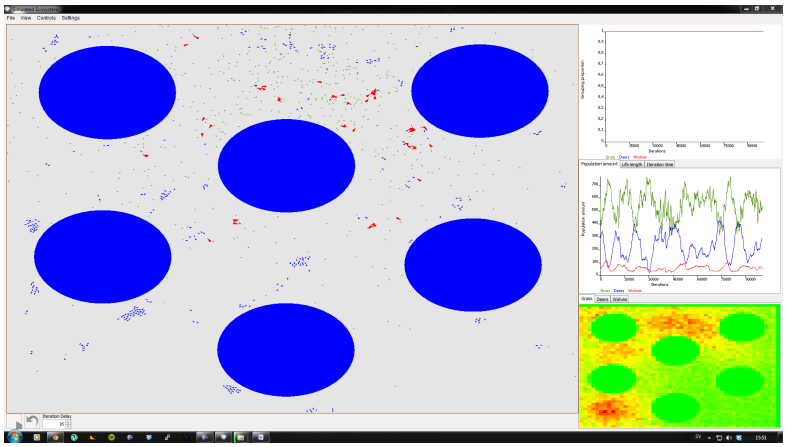


Figure 5.14: *This map simulates an open space with some obstacles such as lakes or mountains.*

Figure 5.14 shows an open map with some obstacles. The reason with this map was to investigate how the populations would behave if they were forced to be spread out, and

according to the heat map they were spread out. The population size graph is similar to the default run.

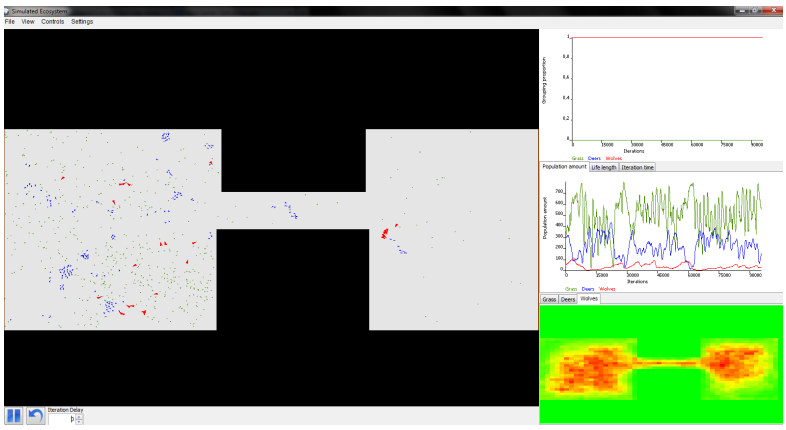


Figure 5.15: *A map divided into two subparts.*

Figure 5.15 shows a map that is divided into 2 subparts, of which can be seen as two ecosystems that agents move between. The effect of this is a more oscillating effect of the population, than the empty map. The predators guards the grass, eating any prey coming close to them.

5.3.2. Randomly generated maps

Another strategy was random generation of maps. 100 maps were randomly generated containing between two and ten obstacles large enough to force the agents to take a detour around it. Every generated map was run for 100 000 iteration.

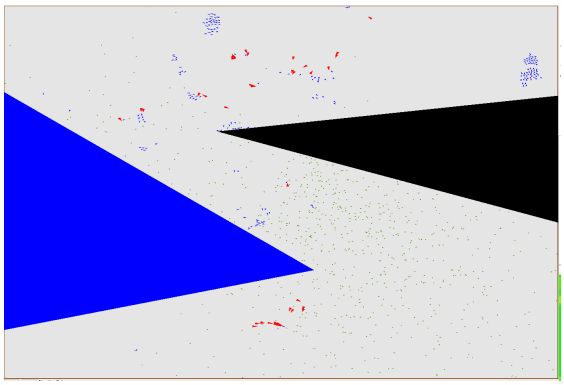


Figure 5.16: *Randomly generated map.*

Most maps did not show any unusual behaviour, but one map that showed some slightly interesting behaviour was a simple map with two triangular obstacles at the left and

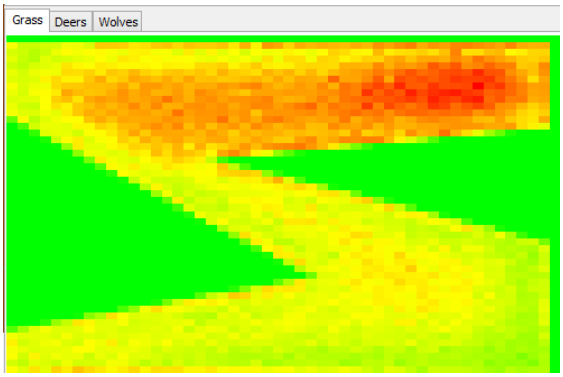


Figure 5.17: *Grass heat map for a randomly generated map.*

right side displayed in Figure 5.16. What was interesting about this map was that the grass mainly grew at the top of the map and the predators circled around the left side, guarding the grass. The heat map for the grass is shown in Figure 5.17.

5.4. Effects of evolution

Many simulations were made in order to try and reproduce a certain aspect of evolution. Starting from a situation where no agents wanted to group with agents of the same population, and ending with almost every agent grouping with others. Evolution was also used as a tool for optimizing the parameters for the grouping behaviour.

The grouping behaviour consists of the four forces: Separation, cohesion, arrayal and forward thrust. The parameters were initialized randomly from start and every child inherited its parent's parameters with some small random mutations. In Figure 5.18 and 5.19 box plots are shown for the prey and predator populations. The left box in each plot represents the random distribution from start, where the right box is the distribution after hundreds of thousands iterations. The figures suggest that it is favourable for both populations to have a high separation force, a low cohesion force and arryal force and and an average forward thrust. Table 1 and 2 shows the means in the start and after the simulation. A two sample t-test of equal means was made to verify that the observed changes in mean were not likely to be random. All tests in the tables are significant.

Table 1: *Showing the means of the parameters, in the beginning and in the end of the simulation, for the prey population. The changes in means are all significant according to a two sample t-test of equal means.*

	Separation	Cohesion	Arrayal	Forward
old mean	56.99	5.08	5.10	1.01
new mean	101.47	1.62	2.45	1.89
p-value	0.00	0.00	0.00	0.00

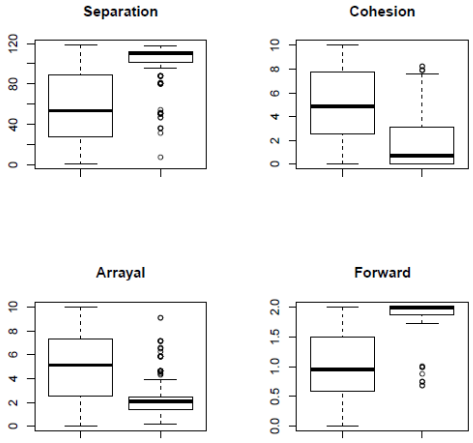


Figure 5.18: *Showing four boxplots for the distributions of parameters for the prey. The boxes to the left are the distribution from the start and the boxes to the right are the distribution towards end of the simulation. The changes in the means of the parameters are significant.*

Table 2: *Showing the means of the parameters, in the beginning and in the end of the simulation, for the predator population. The changes in means are all significant according to a two sample t-test of equal means.*

	Separation	Cohesion	Arrayal	Forward
old mean	71.06	9.70	5.01	1.28
new mean	131.98	0.89	1.35	0.93
p-value	0.00	0.00	0.00	0.00

The purpose with the main experiment of evolution was to reproduce the emergence of group behaviour in any of the populations. In the experiment no agents had any group behaviour at all from the start. With a small probability a gene of a child, which determined whether the agent had group behaviour or not, was mutated. The top right panel in Figure 5.20 shows the proportion of agents grouping with each other. The blue line is the proportion for the prey and the red line is the proportion for the predators. It can be seen in the figure how no agents have grouping behaviour from the beginning. Then one agent evolves and gets the chance to spread this gene to its childs. The gene gets spread and evolved by more agents until finally almost the entire populations is dominated with agents grouping with each other. The proportion for the grouping behaviour is also very stable for the predator population. There are some variations in the proportion for the prey where it during a short time drops below 0.6. However, the proportion is above 0.8 for most of the time.

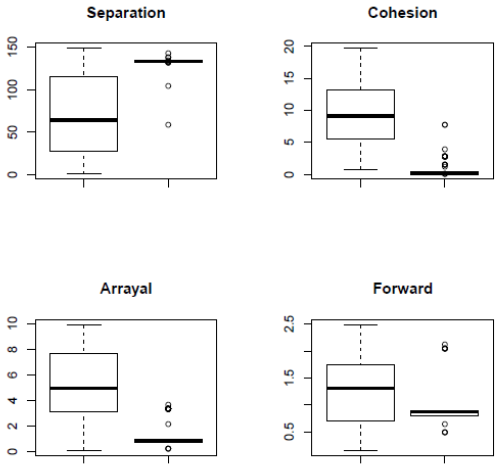


Figure 5.19: *Showing four boxplots for the distributions of parameters for the predators. The boxes to the left are the distribution from the start and the boxes to the right are the distribution towards end of the simulation. The changes in the means of the parameters are significant.*

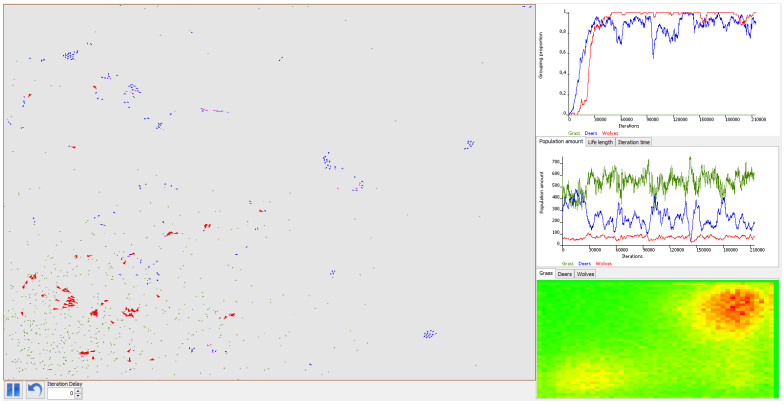
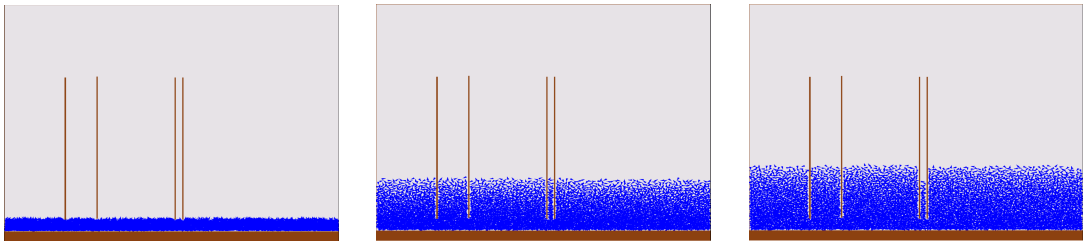


Figure 5.20: *Showing the scenario where no agent in the predator and prey populations have grouping behaviour from the start. In a relatively short time the majority of both populations have evolved grouping behaviour. It is also shown how the proportion of agents having grouping behaviour is stable after it is evolved, except from one greater dip for the prey population around 90 000 iterations.*

5.5. Physical experiments

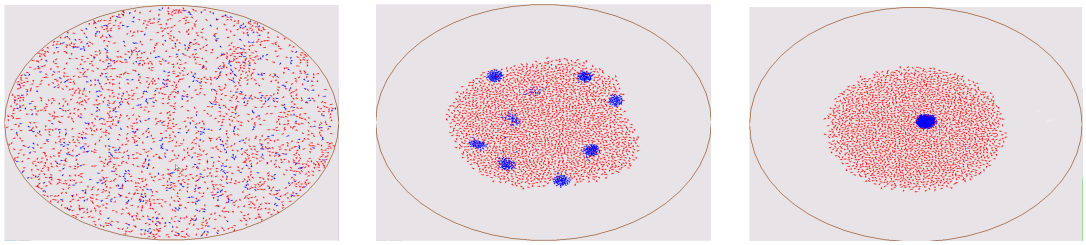
In the category for physical experiments mainly two different kinds of simulations were made. The first experiment managed to reproduce the capillary effect similar to mercury (the mercury lowers in the tube). The result from this simulation is shown in Figure 5.21. Many simulations with different parameters were also made in order to try and reproduce the capillary effect similar to water where the water rises in the tube. Unfortunately, the capillary effect was not reproduced with the parameters in the simulations.



(a) (b) (c)

Figure 5.21: *(a) showing the starting position of the agents with the surface below the caps. (b) shows the simulation after a few seconds where the agents are almost fully diffused. (c) Shows the end of the simulation where the capillary effect, similar to the one for mercury, has taken place.*

The second experiment tried to reproduce the effect of oil forming drops in water where the water completely surrounds the oil. The result of the experiment is shown in Figure 5.22. In the figure it is shown how the all agents are randomly distributed from the beginning, and towards the and the blue agents have formed a single drop surrounded by the red agents. In the experiment behind the figure the blue agents had a greater intermolecular force, dragging them towards each other, than the red agents. The separation force between agents of the same color is the same for all agents, and the red and blue agents also repels from each other.



(a) (b) (c)

Figure 5.22: *Two populations (red and blue) with different properties are shown. Both red and blue agents wants to group with agents of the same population. In (a) the random uniformly starting distribution for the two populations is shown. In (b) the blue agents have after a short time clumped together in a small number of drops. (c) is showing the blue agents forming a single drop completely surrounded by the red agents. This is very similar to what you see if you place a drop of oil in water.*

6. Discussion

A major part of the project consisted of balancing parameters, since the system is very sensitive to changes in them. Adding a new parameter to the system often required rebalancing many of the other parameters. Small changes in the rate of which a population reproduces could set the whole system out of balance.

It was important to understand the impact each parameter has on the system. An example of this is how the two parameters reproduction rate and max energy both limit the size of a population, but in two different ways. The reproduction rate controls how fast a population reproduces, while the max energy controls how long an agent can survive without food. In conditions where the predators are many and the prey are few, the amount of predators need to decrease in order to give space for the prey to reproduce. The first thought might be to decrease the reproduction rate of predators, to prevent the population from growing too big. Instead it is the max energy parameter that is needed to be lowered, making the predators ability to survive lower.

Towards the end of the project, simulations were run to be able to discuss and draw conclusions from the results. The results are categorized based on what was investigated: shape, size, the presence of different obstacles and evolution of different traits. Also, a few simulations of physical experiments were made, to try and reproduce already known physical phenomena, as a way to validate the interactions between agents.

6.1. General Observations

In the beginning of a simulation all agents are uniformly distributed on the map. Grass which spawn close to prey will be eaten immediately. However, grass close to predators have less probability to be eaten as a result of prey avoiding predators. This gives the grass a chance to grow undisturbed and after a while the density of grass is higher around the predators, compared to the rest of the map. When the scattered grass has been consumed, the prey are attracted to the spot of grass, giving the predators the opportunity to catch them without having to move long distances. The predators guarding the grass may give the impression of them being intelligent, when in fact the reason is that there is no need for them to move as the prey are always nearby.

6.2. Shape

The simulations show that the shape of the environment has a negligible impact on the outcome of the result as long as the total area remains the same. There is a slight tendency for the agents to be drawn to different sides of the environment depending on what shape is used, but this is hard to assert without more testing, and more accurate interpretation than just visual.

A more interesting event develops in corners, since the prey in their search for grass sometimes end up trapped by the predators, with nowhere to escape when they reach a corner. However, this has no long-term effects on the stability of the simulation, and does not seem to be a greater issue than prey being caught by predators while following the edge of the environment. In general the behavioural patterns remain the same no matter what shape is used for simulation. This could be a result from the fact that grass tends to group up with predators staying in the same areas which reduces the amount of hunting needed by the predator. If the grass spawned entirely at random there might be a bigger difference between the shapes.

6.3. Size

The results of the simulations shows that the size of the inhabited universe very much affects the outcome. Changing the size upsets the balance in the system, and prevents the populations from recovering. Different sizes also favours different populations.

With the 500*500 universe, the universe is too small for the populations, and the outcome is somewhat chaotic. The fact that either the prey or the grass population goes extinct first, and that it happens almost immediately, shows that sizes this small do not support any stability for populations.

The 1000*1000 universe is interesting since it always favours the grass. The grass seems to have an advantage in this particular size. Perhaps because in such a small environment, the predator can hunt the prey down effectively, since the prey spawn next to its parent, and is therefore already in the vision range of a predator that hunts the parent. The grass, however, can spawn far away from the nearest prey, as well as right in front of it.

When the size of the universe is set to 1500*1500 or 2000*2000, balance between populations occurs. This was not a surprising result, since the parameters have been tuned to be balanced for 2000*2000 distance units. The differences in the heat maps presented in the result section are a natural effect of the total area being smaller.

For larger sizes, such as 4000*4000, the balance shifted again. Because of the large area, the grass can grow more freely, which leads to the prey having much to eat. But the amount of prey does not increase as much as the grass. In the smaller universes the highest amount of prey often reaches the lowest amount of grass, but in the 4000*4000 environment the amount of prey is far below that of the grass, after the initial instability. Instead it is the amount of predators that reach the amount of prey.

The explanation for this is probably closely related to the fact that the grass groups up. The prey that are at the outskirts of this patch of grass quickly finds plenty of food to eat so they can reproduce, but the close proximity to predators lead to easy food for them as well. The rest of the environment however, is scarcely populated by grass, and the prey that roam this large area is not in any direct danger of being eaten by

predators, but may instead die of starvation due to how long it takes them to find new grass to refresh their energy. This is reflected in the short life length of the prey.

It is interesting that there was a case when the predator population was the only one to go extinct, since this did not happen with any other sizes. It happened just in the beginning of the simulation, while both the grass and prey were spread out which caused the predators to die of starvation. However, this result, as well as many of the other ones, need to be viewed critically. Since both the iterations and the number of times each simulation was run was limited to about 100 000 and 10 respectively, the statistics can not be fully trusted. There is a, not negligible, possibility that the results could have been quite different if the simulations were run 100 times with the same configurations, or for 1 000 000 iterations.

6.4. Obstacles

According to the results, obstacles had no or little impact on the agents. It was not expected, in fact the opposite seemed more probable. In Figure 5.12 and Figure 5.14 one large obstacle is used as well as lots of small ones, but none of these maps created any significant changes compared to the default run in Figure 5.11. The only variation was the geometrical distribution, but the populations were still stable.

To affect the agents their available space had to be limited, which could be done with obstacles. One such example is the fragmented map in Figure 5.13, where the obstacles divide the map into small areas which were too small to support stable populations. Both the grass and the prey were eaten faster than they could reproduce, which resulted in high fluctuation rates for the populations. The behaviour was similar to when the size of the environment was decreased to 500*500 distance units, but because there were several “boxes”, the agents could move around instead of completely die.

The limited impact of obstacles might be related to where new grass is spawned. New grass agents are positioned close to their parents, which results in the grass growing in a limited area significantly smaller than the total area of the environment. If new grass was spawned randomly, prey, and eventually the predators, would move around and probably be affected more by obstacles. This issue was discovered late in the project, during the simulation phase, and therefore no time was left to adjust the spawning behaviour.

6.5. Evolution

When running simulations where the parameters for group behaviour are allowed to change, there seems to be parameters significantly more favourable for the agents. For both predators and prey it is highly desirable to keep a certain distance to agents in their group and in order not to be too close. This can be seen by the increase of the separation force parameter scaling the force between agents when they come too close. A

possible explanation to this is that when prey are keeping the distance from neutral agents, they are less likely to collide with them when making sudden moves trying to avoid a predator. For predators it is more favourable to keep the distance in order to spread out and cut off possible ways that a prey could escape in.

In contrast to the parameter for separation, it is favourable to have a low cohesion parameter. In this way the agents are allowed to group with each other without it conflicting with the advantageous behaviour of staying at a certain distance from each other. It is also suggested from the experiment that the arrayal force parameter should be greater than zero, but kept relatively small. The effect of the arrayal force being too large will inflate the agents ability to change their direction quickly, since they will insist on going in the same direction as agents of the same population. The same can be argued for the forward thrust parameter. It is important for the agent to keep heading in its own direction, but blindly insisting on going the same direction (large value of forward thrust parameter) will ignore both neutral agents, predators and/or prey, resulting in a certain death.

Another experiment strongly suggested that group behaviour as a whole is favourable for both the predators and prey. Both populations started with no agent trying to group with others, and after a relatively small number of iterations the populations were dominated by agents with group behaviour. A reason for the wolves to evolve group behaviour could be that it is easier to catch a prey when many wolves are forming a wide group. In that way it is harder for the prey to escape either right or left. Instead, the prey are chased straight forward into a wall or until they are caught.

For the prey it is less obvious why it is beneficial to stay together. One possible explanation is that it helps them find areas with higher density of grass. When a prey sees an area with grass, it feels the attraction to that area, and inspires nearby prey with group behaviour to follow his way. It also helps the prey when they have to choose between moving to a small grass patch that is nearby, and a larger area that is further away. It is more likely for other prey to be at the larger area of grass, and therefore the group behaviour helps them choose the large area. This phenomenon will help prey with group behaviour to reproduce and spread the gene.

6.6. Physics

The experiment with capillary effect is one of the most interesting result. Reproducing a well known physical phenomenon suggests that the interactions between the agents are done in a realistic way. This also suggests that the agents in our system have properties similar to the properties of mercury. It would require more knowledge behind both the phenomenon and mercury to decide whether our agents do have similar properties or not. Experiments with different parameters were also made in order to try and reproduce the capillary effect similar to the one with water. Unfortunately this was done without success. Again, more knowledge of the phenomenon would probably be required in order

to reproduce an experiment with results similar to the capillary effect of water.

The experiment reproducing the effect of oil forming drops on a surface of water is also an interesting result. A considerable amount of time was spent finding the parameters making it happen. Noticeable here is that the parameters were changed to specifically reach the effect of oil floating on water, and not set to be similar to the real properties of water and oil. Therefore, there is a possibility that the results of this experiment could not be reproduced in reality with substances having similar properties to the agents in the simulation. There is also a possibility that there exists substances, with similar properties to the agents in the simulation, that would behave in this way in reality. If the latter is true, simulations could be done to predict the behaviour of substances not yet experimented with.

7. Conclusion

From the simulations and discussion it can be concluded that the size of the environment has a direct impact on the dynamics of the system. The results of the simulations with a smaller area strongly indicates that drastically decreasing the size of a balanced ecosystem will have devastating effects for the populations within it. Increasing the size of the area also has an impact on the populations. Simulations with larger sizes indicates that both predators and prey sometimes have problems finding food in a large and sparse environment.

The shape as well as the obstacles have negligible impact on the populations, based on the performed simulations. Obstacles have the biggest impact when they split the environment into small parts. This results in chaotic behaviour similar to when simulating with a smaller environment. A probable cause is the spawning behaviour of the grass, which directs the movement of the prey and predators.

The evolution simulations showed that group behaviour benefits both the prey and the predator populations. The prey which evolved grouping managed to survive longer and had more time to eat and reproduce, which resulted in the majority of prey moving in groups. The predators which evolved group behaviour were significantly more successful in catching prey. The predators moving as a group eliminated many of the possible directions a hunted prey could escape in resulting in the prey getting eaten with a higher probability.

The fact that already known physical phenomena are able to be reproduced, using the same rules for agent interaction as in the other simulations, supports the validity of the models. More in-depth knowledge is needed in order to be able to draw more sophisticated conclusions. However, it still inspires confidence that the system may be of further use in simulations of ecosystems as well as for simulating other physical phenomena.

References

[1] J. A. Baggio, K. Salau, et al, "Landscape connectivity and predator-prey population dynamics" in *Landscape Ecology*, vol. 26, Netherlands: Springer, 2011.

[2] J. Berndtsson, I. Domingues. (2013-05-15). Användbarhet i praktiken [Online]. Available: `http://anvandbarhet.se/index?page=index`.

[3] D. Eberly. (2013-06-05). Distance from a Point to an Ellipse, an Ellipsoid, or a Hyperellipsoid [PDF]. Available: `http://www.geometrictools.com/Documentation/DistancePointEllipseEllipsoid.pdf`.

[4] R. Eckstein. (2013-05-20). Java SE Application Design With MVC [Online]. Available: `http://www.oracle.com/technetwork/articles/javase/index-142890.html`.

[5] D. Harabor, A. Grastien. (2013-05-20). The Australian National University, Online Graph Pruning for Pathfinding on Grid Maps [Online]. Available: `http://www.nicta.com.au/pub?doc=485`.

[6] A. Kamimura, G. F. Burani, H. M. França. (2013-05-09). The Economic System Seen As A Living System: A Lotka-Volterra Framework [Online]. Available: `http://web.ebscohost.com/ehost/detail?sid=520f50f6-36b2-4646-a997-16142f7d1995\%40sessionmgr4&vid=1&hid=28&bdata=JnNpdGU9ZWhvc3QtbGl2ZQ\%3d\%3d#db=buh&AN=70109948`.

[7] R.S. Kenett, E.R. Baker, *Software Process Quality: Management and Control*. New York: Marcel Dekker Inc., 2005.

[8] C. Larman, *Agile and iterative development*. Boston: Addison-Wesly, 2004.

[9] I. Maier, M. Odersky, T. Rompf. (2013-06-05). Deprecating the Observer Pattern. Ecole polytechnique fédérale de Lausanne [Online]. Available: `http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf`.

[10] A. Okubo, S. A. Levin, *Diffusion and Ecological Problems: Mathematical Models*. New york: Springer, 2001.

[11] K.V.S. Prasad, Associate Professor, Department of Computer Science and Engineering, Chalmers University of Technology, 2013.

[12] A. S. Swaab. (2013-06-05). Random point in a Triangle - Barycentric Coordinates. Technical Repository [Online]. Available: `http://adamswaab.wordpress.com/2009/12/11/random-point-in-a-triangle-barycentric-coordinates/`.

[13] M. Wahde, *Biologically inspired optimization methods: an introduction*. Great Britain: Orca/WIT Press, 2008.

[14] World Wide Web Consortium. (2005-01-05). Document Object Model [Online]. Available: http://www.w3.org/DOM/.

A. Mathematics behind agent behaviour

One essential part of this project is to mathematically model how the agents in the system will move and interact with each other. Mathematics is also important in other parts of the application. One example is the graphical representation of agents. If the agent is to be represented as a triangle on the screen, pointing in the direction that the agent is heading, the need for mathematical calculations is needed in order for the triangle to be correctly drawn.

A.1. Agent movement and behaviour

The modelling of agent movement can be done in various ways. In the application the movement of an agent is based on Newton's laws and equation of motion. The agent is affected by a discrete number of forces and the sum of those forces, divided by its mass, determines its acceleration ($\mathbf{a}(t) = F(t)/m$).

For simplicity, the mass of all agents is set to 1. The acceleration is applied to the agent's velocity, which in turn is used to determine the position of the agent in the next timestep $t + \Delta t$. The forces, accelerations and velocities are all represented by 2-dimensional vectors, since the agents live and interact in a 2-dimensional space. According to Newton's equation of motions the acceleration of an object is the sum of forces that it is affected by.

The forces listed in Section 4.1 can be split into two groups; the forces explaining the agent's own will and the forces an agent have no control over. The predator force, prey force, mutual interaction force, arrayal force, forward thrust and random force are all group as the forces explaining the agent's own will. The environment force is a force that the agent can not control. An example of this is that an agent chooses to flee from a predator hunting it, but it can not choose to ignore the force of colliding with a wall.

The acceleration of own will $\mathbf{a}_i^{own}(t)$ of an agent is determined by the following equation:

$$\mathbf{a}_i^{own}(t) = C_1\mathbf{F}_{pred} + C_2\mathbf{F}_{prey} + C_3\mathbf{F}_{mi} + C_4\mathbf{F}_{array} + C_5\mathbf{F}_{forward} + C_6\mathbf{F}_{random},$$
where C1,...,C6 are constants weighing the different forces. (A.1)

If the norm of the acceleration exceeds the value *maxAcceleration* for the agent, the norm is scaled back to *maxAcceleration*. After the acceleration has been scaled, the environment force is applied to the acceleration of own will to form the final acceleration:

$$\mathbf{a}_i(t) = \mathbf{a}_i^{own}(t) + C_7\mathbf{F}_{env},$$
where C7 is a constant scaling the force to a propriate value. (A.2)

The velocity $\mathbf{v}_i(t)$ an agent has at time t is given by Newtons equation of motion:

$$\mathbf{v}_i(t) = \mathbf{v}_i(t - \Delta t) + \mathbf{a}_i(t) \cdot \Delta t \tag{A.3}$$

If the norm of the velocity exceeds the value *maxSpeed* for the agent, the norm is scaled to back to *maxSpeed*.

The position $\mathbf{P}_i(t)$ that an agent will have in the next time step $t + \Delta t$ is then:

$$\mathbf{P}_i(t + \Delta t) = \mathbf{P}_i(t) + \mathbf{v}_i(t) \cdot \Delta t, \quad \text{where } \Delta t \text{ is set to 1 for simplicity.} \tag{A.4}$$

A.1.1. Predator force

The prey wants to find the direction, in what to best escape the predators hunting them. If only one predator is near, the best way to go is straight away from the predator. If there are two or more predators near, the best direction is no longer as obvious.

In this project some simplifications have been made, in order to get a closed form solution. Since the maximum distance from all the predators is infinity, this becomes a difficult problem. It is easier to solve the problem of finding the position that would minimize the distance to all nearby predators. With the knowledge of that position as a prey, it would make sense to go as far away from it as possible. The problem now consists of finding the position that would minimize the distance to all nearby predators. The fact that predators located closer to the prey are more fearsome also has to be taken into account.

Let $d_{i,j}$ denote the distance between the prey i and the predator j and $\mathbf{P}_i$ the position of agent i. The vector $\mathbf{F}_{i,j} = (\mathbf{P}_i - \mathbf{P}_j)$ is then the vector pointing from the predator to the prey. It is then needed to find the vector $\mathbf{v}_b$ that minimizes the norm of the following expression:

$$f(\mathbf{v}_b) = \sum_{\text{nearby } j} (\mathbf{F}_{i,j}/d_{i,j}^2 + \mathbf{v}_b) \tag{A.5}$$

Since $\mathbf{v}_b$ is a 2-dimensional vector with coordinates x_b and y_b the partial derivatives of $f(\mathbf{v}_b)$ can be taken with respect x_b and y_b and set to 0. The x_b and y_b that satisfies the two following equations will be the x_b and y_b that minimizes $||f(\mathbf{v}_b)||$:

$$\frac{df}{dx} = \frac{d((\sum_j (x_{i,j}/d_{i,j}^2 + x_b))^2 + (\sum_j (y_{i,j}/d_{i,j}^2 + y_b))^2)}{dx} = 2n(n \cdot x_d + E_x),$$
$$\text{where } n > 0 \text{ is the number of nearby predators and } E_x = \sum_{nearbyj} x_{i,j}/d_{i,j}^2. \tag{A.6}$$

$$\frac{df}{dy} = \frac{d((\sum_j (x_{i,j}/d_{i,j}^2 + x_b))^2 + (\sum_j (y_{i,j}/d_{i,j}^2 + y_b))^2)}{dy} = 2n(n \cdot y_d + E_y),$$
$$\text{where } n > 0 \text{ is the number of nearby predators and } E_y = \sum_{nearbyj} y_{i,j}/d_{i,j}^2. \tag{A.7}$$

Setting the derivatives to 0 gives:

$$x_d = -E_x/n$$
$$y_d = -E_y/n$$
$$\mathbf{v}_b = (x_d, y_d)$$

The worst position prey i could be in is then $\mathbf{P}_i + \mathbf{v}_b$. Instead the prey should head in the opposite direction towards $\mathbf{P}_i - \mathbf{v}_b$. This makes the predator force be:

$$\mathbf{F}_{pred} = -\mathbf{v}_b.$$

A.1.2. Prey force

The force that any agent feel towards its prey is very similar to the predator force. Instead of the trying to maximize the distance to its predators, the agent is in this case trying to minimize the distance to its prey in the same way. The predators in the top of the food chain also has the ability to focus on single prey. When a any prey comes closer than the focus range, the predators ignores any other prey and feels a force only to the focused prey. When there is no prey within the focus range the predator tries to minimize the distance to all nearby prey.

A.1.3. Environment force

Since the agents should not move outside the universe, or into any obstacles, the interaction with walls and obstacles need to be modelled in a clever way. To check at every position update if the new position was valid is not a good approach, since it raises the question of what to do if the new position is not valid.

The way this interaction is modelled in the application is again with forces. If an agent is close to the wall, it should feel a force pushing it away from the wall. The closer to the wall an agent is, the greater force it should feel. On the other hand, if the agent is not close enough to the wall, it should not be affected by it. This makes sense since, if an

agent is not close enough to a wall or an obstacle, it might not see it. Also if it is running towards a wall, it will come to a point where it would want to stop in order not to run in to the wall. Before that limit, it can still run with maximum speed.

These two features can be modelled in two steps. The first property, "the closer the wall, the greater the force", is modelled with a simple and well recognisable function, namely y=1/x. Let y be the magnitude of the force and x be the distance to the wall. The function obtained now fulfills the desired properties. The force should be orthogonal to the wall and pointing into the universe.

The next property to model is that the agents should not feel the force if they are not interacting with the wall. This could be modelled with a simple Heaviside step function (f(x) = 1 if x greater than 0 and = 0 otherwise). A function like that can be seen in Figure A.1. The direction of the force is the direction of the vector pointing from the closest point on the wall/boundary to the agent's position. This is true for the forces an agent feels from both the walls of the universe and the boundary of an obstacle. The equation for the force is then:

$$\sum_{all\ obstacles+walls} Heaviside((C - 1/x)/x^2)(\mathbf{P_i} - \mathbf{P_b}), \quad \text{where } \mathbf{P_i} - \mathbf{P_b} \text{ is the vector from obstacle/wall } b \text{ to agent } i \text{ and } x \text{ is the distance to the obstacle/wall} \tag{A.8}$$

A.1.4. Mutual interaction force

One other desired property of the agent's behaviour might be that the neutral agents want to group with each other. Forming a group might improve the agents chances to survive against predators. The mutual interaction force will make agents far apart steer towards each other to form a group (attraction). At the same time, the agents do not want to collide with each other and will therefore steer away from other agents that are too close (repulsion). The mutual interaction force is defined as:

$$(\mathbf{F}_{mi})_i = \sum_{j!=i} Q(d_{i,j})(\mathbf{P_i} - \mathbf{P_j})/d_{i,j}, \tag{A.9}$$

where $\mathbf{P_i}$ and $\mathbf{P_j}$ is the position of the two neutral agents i and j, and Q is a function of the distance between the two agents. $Q(d_{i,j})$ is defined as follows:

$$
\begin{aligned}
Q(di,j) &= -c_0(d_0 - di,j), && \text{when } 0 < d_{i,j} < d_0 \text{ and } c_0 > 0.\\
Q(di,j) &= c_1 > 0, && \text{when } d_0 < d_{i,j} < d_{vision}.\\
Q(di,j) &= 0, && \text{when } d_{vision} < d_{i,j}.
\end{aligned}
$$

An agent is then subject to a linearly varying repulsive force from another agent if the distance between the two is less than d_0. If the agent has vision of another agent, at the same time as the distance between them is greater than d_0, the agent is subject to a constant attractive force dragging it towards the other agent. If the agents are not in vision range of each other the mutual interaction force is 0 (no interaction).

A.1.5. Arrayal force

The arrayal force describes the tendency of agents in groups to move in the same direction. Two neighbouring agents want to equalize the velocities in order to steer in the same direction. Without this force, the agents would just form a group where the center of mass is not moving anywhere. The agents are only influenced by other agents that are within a sphere of influence (V_i). This means that an agent does not try to equalize its velocity with other agents that are too far away, even if they are in the same group. The arrayal force that an agent experiences has the following form:

$$
(F_{array})_i = \frac{1}{N_i} \sum_{j \text{ in } V_i} h(\mathbf{v}_i + \mathbf{v}_j), \tag{A.10}
$$

where N_i are the number of agents within the sphere of influence.

A.1.6. Forward thrust

The forward thrust is the tendency of an agent to keep moving in the same direction. The forward thrust is expressed as:

$$
(\mathbf{F}_{forward})_i = a\mathbf{v}_i/|\mathbf{v}_i|, \tag{A.11}
$$

where $a > 0$ is a constant called the coefficient of thrust.

A.1.7. Random force

The random force is a force is a simplification of the behaviour of an agent that cannot be properly explained and is therefore said to be random. This could also be interpreted as the error an agent does when estimating in what direction to move. In our system

this force is uniformly distributed and implemented to have very little impact. Therefore, it is almost negligible.

A.2. Finding the closest point on a boundary

To find the point on the boundary of the universe or an obstacle that is the closest to an agent is essential when calculating the environment force. In the section for the environment force it was assumed that the closest point was known. In this section different ways of calculating the closest point will be discussed for a subset of geometrical shapes. The obstacles and the universe can be represented as triangular, rectangular and elliptical shapes.

Also any obstacle can be rotated in any angle. Before doing the calculations explained in this section, the positions of the agent can be rotated into coordinate system of the obstacle. After the closest position has been found it can be rotated back to the original coordinate system. Therefore, when calculating the closest point it is assumed that the obstacles are not rotated at all.

A.2.1. Triangle

A simplification of the triangles were made in order to make the calculations simpler. The application only supports triangles that are isosceles. Because of that one can exploit the symmetry of the triangle when finding the closest point on its boundary. All the positions to the left of the centroid of the triangle could be treated as they were to the right because of the symmetry. Therefore it is from now on supposed that the agent is located to the right of the centroid.

Let the top of the triangle always be pointing upwards. If the agent is located above the top, the closest point will always be on the top of the triangle. If the agent is located below the triangle, the closest position is in the bottom corner if the agent is also located to the right of the bottom corner. If the agent is located to the left off the bottom corner, the closest position is on the base of the triangle where the vector between the agent and the closest point is perpendicular to the the base.

If the agent is located below the line, perpendicular to the side of the triangle cutting the bottom corner of the triangle, the closest position is the bottom corner. If the agent instead is located above the line, perpendicular to the side of the triangle cutting the top of the triangle, the closest position is the top of the triangle. If not any of the previously cases explained are true, the closest point will be where the vector between the agent and point is perpendicular to the side of the triangle.

A.2.2. Rectangle/Squre

The case of an agent being inside a rectangular shape is the easiest case of them all. All that has to be done is to project an agent's position onto the sides of the rectangle and compare which of them is the closest. If the agent is positioned outside the rectangular shape , this becomes another problem. In Figure A.1, 8 areas are drawn. Depending on which area the agent is located in, the method for finding the closest point differs. If an agent is located in one of the corner areas (1, 3, 5 and 7), the closest point on the rectangle is the closest corner. If the agent is located in any of the other areas, the closest point is given by projecting the agent's position onto the closest side of the rectangle.

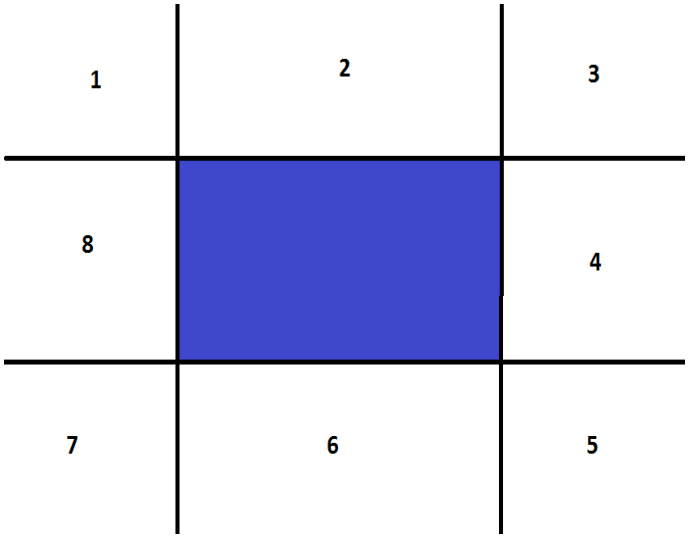


Figure A.1: *Showing a rectangular obstacle in blue. The space around the obstacle is divided into 8 areas. Using the symmetry of the rectangle the areas could be reduced to 3.*

A.2.3. Circle

To find the closest point on the border of the circle, regardless of being inside or outside it, is simple. The closest point is located on the line that crosses the center of the circle and the agent's position. There will be two points on the line that intersects with the boundary of the circle, where the closest one will be chosen.

A.2.4. Ellipse

This is by far the most difficult scenario, since there is no closed form solution for the closest point. Instead one has to take a different approach when finding the closest point on an ellipse. Here three different ways will be discussed and evaluated.

The first way of finding the closest point is to do an exhaustive search along the border of the ellipse. Since the equation of the ellipse is known, N points can be generated along its border. This is a really easy way to find an estimate of the closest point. The downside is that N must be large in order to get a precise estimate. Since N comparisons has to be made in order to find the closest of the N generated points, this algorithm becomes computationally heavy. Especially when there are a lot of agents doing this calculation every iteration.

A better and faster way is to do a recursive algorithm where the symmetry of the ellipse is exploited. In each iteration of the algorithm, points that are known to be farther away than points already found is excluded. In this way the search space will be more limited in each iteration, and after enough iterations a good estimate will be found. The algorithm looks as follows:

1. Initialize starting angle $\alpha = 0$, and step size $t = 2\pi/n_{step}$, where n_{step} is an even integer $>= 4$.
2. Evaluate n_{step} points on the ellipse, where the points are $(a \cdot cos(\alpha + n \cdot t),\ b \cdot sin(\alpha + n \cdot t))$ and $n = -n_{step}/2, ..., n_{step}/2$.
3. Take the n' for which the point is closest to the agents position.
4. Set $\alpha_{new} = \alpha + t \cdot n'$ and $t_{new} = 2t/n_{step}$
5. Go to step 2 with $\alpha = \alpha_{new}$ and $t = t_{new}$, until t is smaller than a stopping value c.

With this algorithm n_{step} comparisons will be done in each iterations, and the algorithm will stop when t is smaller than a threshold c. Since t is divided by $n_{step}/2$ in every iteration, the number of iterations the algorithm will run can be computed. Let x denote the number of iterations, then the algorithm will stop when $\pi(n_{step}/2)^{-x}$ is smaller than c. The least amount of iterations the algorithm will run is then obtained by solving the following equation for x:

$$\pi(n_{step}/2)^{-x} = c.$$

Which gives

$$x = log(\pi/c)/log(n_{step}/2)$$

.

Since n_{step} comparisons will be done each iterations, a minimum $x \cdot n_{step}$ comparisons will be done with the algorithm. In Figure A.2, a plot for $x \cdot n_{step}$ as a function of n_{step} is given. As seen in the figure, for $n_{step} > 2$ the function only has one minimum.

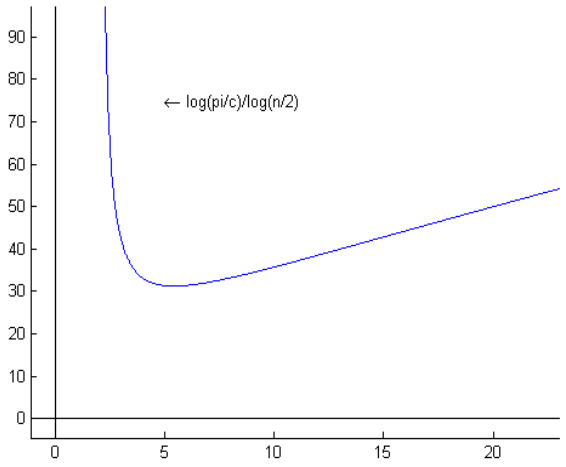


Figure A.2: *Showing the number of position comparisons as a function of the number of splits (n_{step}) in each iteration. The number of position comparisons is minimized for $n_{step} = 2e$.*

Taking the derivative of $x \cdot n_{step}$ and setting it to 0 gives the minimum of the function:

$$\frac{d(x \cdot n_{step})}{d(n_{step})} = \frac{log(\pi/c)(log(n_{step}/2) - 1)}{(log^2(n_{step}/2))} = 0 \tag{A.12}$$

Solving the equation for n_{step} implies:

$$log(n_{step}/2) - 1 = 0 \tag{A.13}$$

This gives $n_{step} = 2e \approx 5.44$. This means that the optimal choice of n_{step} is either 4 or 6. To compare this $x \cdot n_{step}$ is taken for $n_{step} = 6$, minus $x \cdot n_{step}$ for $n_{step} = 4$. If the result is negative, then $n_{step} = 6$ is the best choice. If the result however is positive, then $n_{step} = 4$ is the best choice.

$$\begin{aligned} &6 \cdot log(\pi/c)/log(6/2) - 4 \cdot log(\pi/c)/log(4/2) = \\ &= log(\pi/c) * (6/log(3) - 4/log(2)) = \\ &- 0.31 * log(\pi/c). \end{aligned}$$

Since $log(\pi/c)$ is positive for $c < \pi$, this gives that $n_{step} = 6$ is the best choice for any choice of $c < \pi$.

A good choice of the constant c seems to be 0.01. With $c = 0.01$ the estimate of the closest point will be good enough for the purposes in this project, and the algorithm will find the point with $log(\pi/0.01)/log(6/2) = 5.23$ iterations. This means that the algorithm will run for 6 iterations with 6 comparisons in each iterations. This gives a total of 36 comparisons in order to find the closest point. The estimate retrieved when doing an exhaustive search on the boundary using only 36 points is much less accurate than doing the algorithm explained above.

The last algorithm comes from a more analytical approach of the problem. The calculations for this algorithm are all taken from the paper by David Eberly [3]. The thoughts behind this algorithm will here be explained in order to give the understanding needed to implement it.

Let the equation for the ellipse be denoted as:

$$(x/a)^2 + (y/b)^2 - 1 = 0 \tag{A.14}$$

The closest point from an agent's position $(u,\ v)$ to the ellipse is found where the vector $(u - x,\ v - y)$ is orthogonal to the gradient of the ellipse. This can easily be understood by drawing a picture. The gradient for an ellipse is:

$$\nabla((x/a)^2 + (y/b)^2 - 1)/2 = (x/a^2,\ y/b^2). \tag{A.15}$$

The condition for orthogonality then says:

$$(u - x,\ y - v) = t \cdot (x/a^2,\ y/b^2), \quad \text{for some value } t. \tag{A.16}$$

In this case, only the case where the agent is in the first quadrant $(u > 0,\ v > 0)$ needs to be looked at, since the other scenarios can be found out using the symmetry of the ellipse. Also only the case where $a >= b$ needs to be looked at, since the coordinate system can be transformed to make the ellipse fulfil that inequality constraint.

Solving for x and y gives:

$$x = a^2 \cdot u/(t + a^2)$$
$$y = b^2 \cdot v/(t + b^2)$$

Putting x and y into the equation of the ellipse gives the following equation:

$$F(t) = (\frac{a \cdot u}{t + a^2})^2 + (\frac{b \cdot v}{t + b^2})^2 - 1 = 0 \tag{A.17}$$

The values for t that satisfies the equation $F(t) = 0$ gives the candidate solutions for the closest point to $(u,\ v)$. Since there is a constraint for $x > 0$ and $y > 0$, a constraint for $t > -a^2$ and $t > -b^2$ is also needed. Since $b < a$, only the constraint for t $> -b^2$ is needed.

A problem is that $F(t)$ is a rational function which has no closed form solution for t. There are then two options; Multiply through by the polynomials in the denominator, and analyse the roots of the quartic function obtained, or solve the rational equation numerically. In the paper by David Eberly [3] both solutions are explained, while this report will only cover the latter solution using Newton's method to approximate the roots.

The first derivative of F is:

$$F' = \frac{-2a^2u^2}{(t+a^2)^3} + \frac{-2b^2v^2}{(t+b^2)^3} \tag{A.18}$$

The second derivative of F is:

$$F'' = \frac{6a^2u^2}{(t+a^2)^4} + \frac{6b^2v^2}{(t+b^2)^4} \tag{A.19}$$

If $u > 0$ and $v > 0$, $F'(t) < 0$ and $F''(t) > 0$ for all $t \in (-b^2, \infty)$. Analysing the limits of the function F(t) when t approaches $-b^2$, gives that F(t) approaches infinity. When t approaches infinity, F(t) approaches -1. This means that F(t) is a strictly decreasing function with only one root in the interval $(-b^2, \infty)$.

To approximate the root t^* using Newton's method is ideal since F(t) is convex in the interval of interest. Given a starting value t_0 the iterates of the Newton's method are:

$$t_{i+1} = t_i - F(t_i)/F'(t_i), \text{for } i > 0$$

The only problem is the choice of t_0. An inappropriate choice of t_0 can make the algorithm not converge. An initial choice of $t_0 > t^*$ will lead to a new iterate $t_1 < t^*$.There is a chance that $t_1 < -b^2$ which is outside the domain $(-b^2, \infty)$. Therefore, one has to choose t_0 for which $F(t_0) < 0$ will guarantee that the algorithm converges to the desired root. A choice of $t_0 = b \cdot v - b^2$ fulfils the property of $F(t_0) < 0$ since $F(t_0) = (au/(bv - b^2 + a^2))^2 > 0$. When the root t^* is found, the closest point is on the boundary of the ellipse is $(a^2u/(t^* + a^2),\ b^2u/(t^* + b^2))$.

A quick comparison was made to compare the three different methods. In the comparison the three methods was configured to give approximately the same precision. The time of algorithms was measured when running the each of the algorithms on 10000 randomly

distributed points located outside the ellipse and inside a 1000x1000 square. The ellipse itself was located at the coordinates (500, 500) with b = 150 and a = 50. This was done 100 times to get a good estimate of the mean of the time. Table 3 reveals the results.

Table 3: *Showing a comparison between three different algorithms for finding the closest point on a boundary of an ellipse. The algorithm using Newtown's method was by far the best.*

Algorithm	Mean of time (ms)	Standard deviaton of time (ms)
Brute search (N=700)	1869.13	77.32
Recursive search (nStep = 6, c = 0.01)	109.30	2.63
Newton's method (c = 0.1)	48.07	1.56

There is no doubt that Newton's method is the best, followed by the recursive search with twice the time. Doing an exhaustive search takes more than 17 times longer than the recursive search, and is almost 39 times slower than Newton's method. The algorithm of choice for this project is clearly Newton's method.

A.3. Shapes representing the map

As previously mentioned the objective for the shape is to calculate positions which the agent can use to calculate the environmental force which keeps the agent inside the environment. This has to be calculated upon demand in a way that does not slow down the runtime of the application in a significant way. This means the agents do not actually know what the environment looks like, as long as they can ask the shape where the edges are.

The shapes contain no information or attributes, only pure functions. This way all of the agents can know the same shape, and ask it whenever it needs information without knowing the shape actually knowing anything. As mentioned the shapes have methods to find the edges, but they also have a method to generate a random position inside it to make it easier when initiating the simulation, and one method to check whether or not a given position lies inside the shape.

A.3.1. Finding the edges of a shape

Finding an edge in a square is a simple method. There is no computation required since all of the edges are a direct result of the dimensions of the world.

For the circle however, basic trigonometry is needed. For example finding the right wall, from any given position, is done by:

$$X_{right} = r * cos(\alpha)$$

where r is the radius and α s the angle between the hypotenuse and the base line in the circle, as can be seen in Figure A.3 below. P is the current position of the agent.

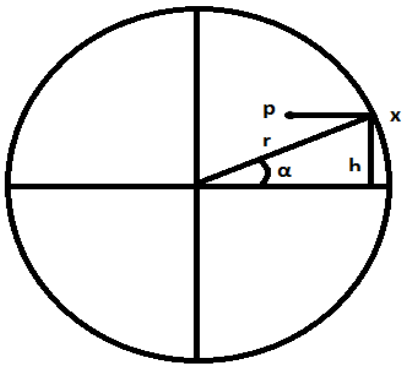


Figure A.3: *Showing how to find the position x in a circle given the position p and the length r.*

For this equation, the angle α needs to be calculated so it can be used to calculate x, which is the length of the adjacent leg in the triangle which is created inside the circle. The only thing the method receives is the size of the world and the position p. It can calculate the radius r and the height h of the triangle from this information and use it to calculate:

$$\alpha = sin^{-1}\frac{h}{r}$$

$$X_{right} = r * cos(sin^{-1}\frac{h}{r})$$

This method will return the position (X_{right}, y) on the right wall. Similar mathematical functions are used to calculate the top, right and bottom edges.

The edge points on the triangle is calculated much in the same way as the circle. Here however, the angle only depends on the relation between the height and width of the triangle and therefore the equation for the angle becomes:

$$\alpha = tan^{-1}(\frac{b}{2h})$$

where b is the length of the base and h is the height of the triangle as can be seen in Figure A.4 below. It is then possible to get the right wall from:

$$X_{right} = (h - y) * tan(\alpha)$$

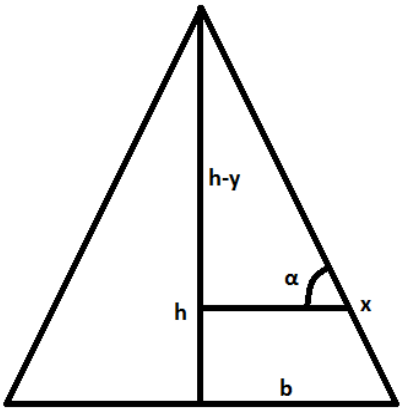


Figure A.4: *Showing how to find the position x in a triangle given the height, base width and P_y coordinate.*

A.3.2. Checking if a position is inside a shape

Checking if a position is inside a shape of specified dimension is a task without much computation needed. For the square the position lies inside the shape as long as the coordinates of the position are greater than zero and smaller than the specified dimension. If both of these are true, then the position lies inside the square.

For the circle it is not much harder. Here the distance between the position and the middle of the environment is found, and if the distance is smaller than the radius of the circle, then it is inside the circle.

The triangle is also easily calculated. The position has to be positive once, just as with the square, and it has to be positioned below the top of the triangle, which is specified by the height of the environment. It also uses the other methods in the shape to find the walls to the right and left at this specific y coordinate, and checks that the x coordinate lies between these two edges. If these conditions are fulfilled, then the position lies inside the triangle.

A.3.3. Finding a random position in a shape

The square has the easiest computational job. When it comes to finding a random position, the only thing that has to be done is generate a random x and y coordinate which both lies inside the specified size, and the task is done.

Finding a random position in a circle can be done in various ways, three ways were considered during this project, but in the end one them was chosen due to the positions being equally spread with this method.

The first way is to start out by creating a random y variable. This y coordinate is used to find the left and right edge at this height, and from get a random x variable which lies in between these two edges. This method however has the disadvantage that the amount of x coordinates which can be generated randomly depends on the width of the circle at that specific height level. Thus it does not generate truly random positions inside the circle, because it clumps up at the top and bottom of the circle.

The second way is to create a random length smaller than the radius of the circle, and rotate this one to get it randomly positioned inside the circle. This seemed to work, but after running simulations it became obvious that this was not the case. Once again the positions generated were not quite evenly spread. This time most of the positions generated lied on the cross which goes through the center of the circle to the top, bottom, left and right edges of a circle.

Because of these issues a third way was chosen to create the random positions. For this one both a random x and y coordinate is generated, in the same way as for a square. A simple check is done to see if this position is actually inside the circle or not. If this is not the case, a new position will be generated. This could in theory loop forever without a position ever being found. In practice this is not something which has a major impact on the system, because the chance of generating a position outside the circle is $(1 - \pi/4 = 21.5\%)$. The second time it loops through to find a position the chance is already at $(1 - \pi/4)^2 = 4.6\%$, and the third time this drops down to 0.99%. This method is also only called upon during the initial setup of the program and therefore does not impact the runtime of the system after it has been initiated.

For the triangle a safer and better method is used than just checking if the generated position lies inside the shape. This method relies on math instead, using Barycentric coordinates. It all starts with the triangle, and its three corners ABC. One is chosen, and vectors are calculated pointing from this vector to the others. Then a percentage of how long along both these vectors the new position should be is generated as can be seen in Figure A.5. To make sure this position does not end up outside the actual triangle a check is made to see if the amount walked along both these vectors is more than 100%. If this is the case, both of them are inverted to be one minus the old percentage. This assures that the calculation gives a random position inside the triangle [12].

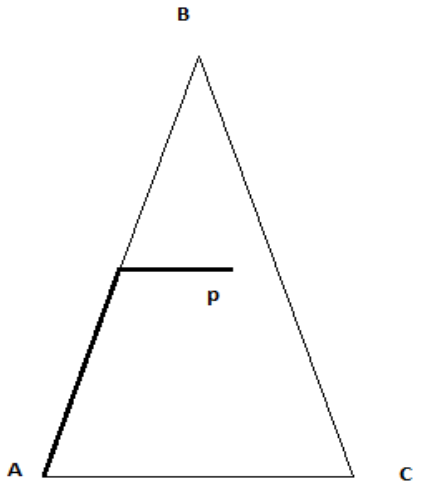


Figure A.5: *How to find a random position p by generating two vectors.*

A.4. Graphical representation of agents

To draw objects with OpenGL in java, one must build the objects out of vertex points that forms a polygon. The simplest geometric form is a triangle (not counting lines and points). This makes it the fastest object to draw which is also very desirable. The problem is then how to draw this triangle given the single position of an agent. The smartest thing would be to draw the triangle with the position of the agent in the center of mass (centroid). In order to find where the center of mass is located in the triangle, it is first necessary to make the assumption that the triangle must be isosceles. The next step is to mathematically calculate where the center of mass is located.

In Figure A.6, b is the width of the triangle and h is the height. The centroid of a triangle can be found by drawing lines from the median of each side, to the opposite corner. The triangle's centroid is located where the lines intersect. Since the triangle is isosceles it is already know that the centroid will be located at y = 0. It is also only needed to draw one line in order to find the centroid. The position for where the line cuts the x-axis gives the position where the centroid is located . The equation for the line is given by:

$$y = b/2 + k \cdot x, \quad \text{where}$$

$$k = \frac{dy}{dx} = \frac{(b/2 - (-b/4))}{(0 - h/2)} = -3b/2h$$

Setting y = 0 gives us:

$$\begin{aligned} b/2 - (3b/2h)x &= 0 \\ 1 - (3/h)x &= 0 \\ 3x/h &= 1 \\ x &= h/3 \end{aligned}$$

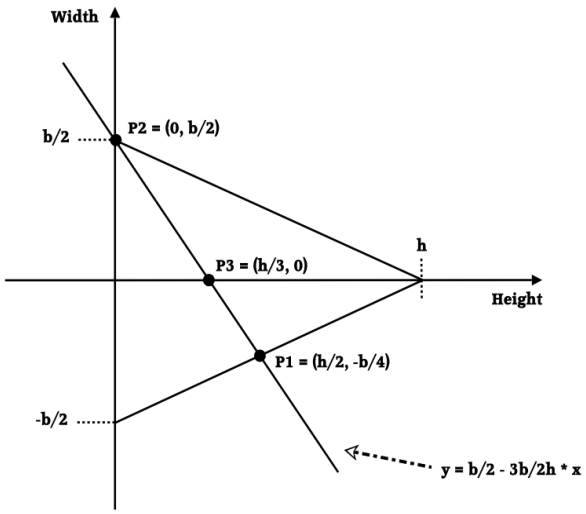


Figure A.6: *Showing a triangle drawn into a coordinate system. As shown in the figure, the centroid of the traingle is located in P3 at the coordinates (0,* $\frac{h}{3}$*).*

As derived above, the centroid is located at ($\frac{h}{3}$, 0), and it is not affected by the width of the triangle, only by the height.

It is now known where the agents location should be in the triangle, but it is also highly desirable that the triangle points in the direction of the agent's velocity. Therefore, the velocity of the agent has to be taken into account when drawing the triangle.

In Figure A.7, there are five different points drawn. The points $Ptop$, P_{left} and P_{right} are the points needed in order to draw the triangle with OpenGL. How to get the these points, given $P_{centroid}$ and $\mathbf{v}$ will here be explained. To get from $P_{centroid}$ to P_{top} one can use the fact that $P_{centroid}$ is located at $\frac{1}{3}$ of the height of the triangle. Therefore, the velocity v has to be scaled to have length $\frac{2}{3}$ of the height and added to $P_{centroid}$:

$$P_{top} = P_{centroid} + \frac{2h}{3}\mathbf{v}/norm(\mathbf{v})$$

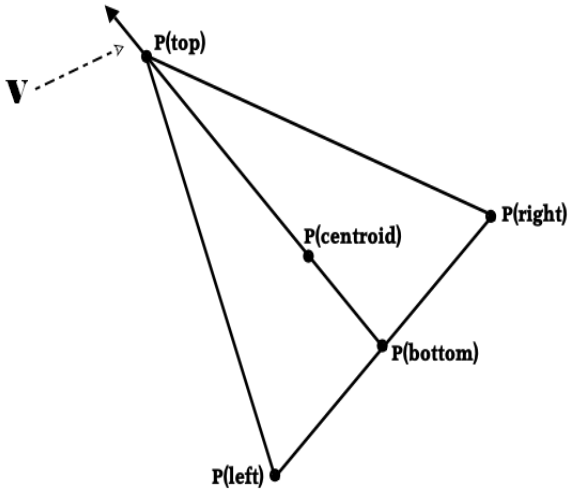


Figure A.7: *Showing a triangle pointing in the direction of the agent's velocity. In the figure the points P(right), P(left) and P(top) used for drawing the triangle is shown*

To get to P_{bottom} a similar thing is done, but instead the vector is scaled to have length $\frac{1}{3}$ of the height and then subtracted from $P_{centroid}$:

$$P_{top} = P_{centroid} - \frac{h}{3}\mathbf{v}/norm(\mathbf{v})$$

In order to get from P_{bottom} to P_{right} and P_{left} a vector, orthogonal to $\mathbf{v}$ with the length $\frac{b}{2}$, needs to be added to P_{bottom}. To form the vector $\mathbf{v_2}$, which is orthogonal to $\mathbf{v}$, the fact that the scalar product of two orthogonal vectors are equal to 0 is used:

$$x_{\mathbf{v}} \cdot x_{\mathbf{v}_2} + y_{\mathbf{v}} \cdot y_{\mathbf{v}_2} = 0 \tag{A.20}$$

and

$$x_{\mathbf{v}_2}^2 + y_{\mathbf{v}_2}^2 = \frac{b^2}{4} \tag{A.21}$$

solve $x_{\mathbf{v}_2}$ from Equation A.20 gives:

$$x_{\mathbf{v}_2} = \frac{y_{\mathbf{v}} \cdot y_{\mathbf{v}_2}}{x_{\mathbf{v}}}, \quad x_{\mathbf{v}} \neq 0$$

Put that into Equation A.21 and the following is given:

$$\frac{y_{\mathbf{v}}}{x_{\mathbf{v}}}^2 \cdot y_{\mathbf{v}_2}^2 + y_{\mathbf{v}_2}^2 = \frac{b^2}{4}$$
$$y_{\mathbf{v}_2}^2(1 + (\frac{y_{\mathbf{v}}}{x_{\mathbf{v}}})^2) = \frac{b^2}{4}$$
$$\frac{b^2}{4(1 + (\frac{y_{\mathbf{v}}}{x_{\mathbf{v}}})^2)} = y_{\mathbf{v}_2}^2$$
$$y_{\mathbf{v}_{2,1}} = +\frac{b}{2 \cdot \sqrt{(1 + (\frac{y_{\mathbf{v}}}{x_{\mathbf{v}}})^2)}}$$
$$y_{\mathbf{v}_{2,2}} = -\frac{b}{2 \cdot \sqrt{(1 + (\frac{y_{\mathbf{v}}}{x_{\mathbf{v}}})^2)}}$$
$$\mathbf{v}_2 = (x_{\mathbf{v}_2},\ y_{\mathbf{v}_{2,1}}) = -(x_{\mathbf{v}_2},\ y_{\mathbf{v}_{2,2}})$$

There will be two solutions for $y_{\mathbf{v2}}$. Depending on which of them is chosen, and the direction of vector $\mathbf{v}$, either P_{right} and P_{left} is given when the addition $P_{bottom} + \mathbf{v}_2$ is done. It is not necessary to specifically know if $P_{bottom} + \mathbf{v}_2$ is equal to P_{right} or P_{left}. For simplicity in, lets call $P_{bottom} + \mathbf{v}_2$ for P_{right} and P_{bottom} - $\mathbf{v}_2$ for P_{left}. If it would happen that $x_{\mathbf{v}} = 0$, then $y_{\mathbf{v}_2} = 0$ and $x_{\mathbf{v}_2} = \frac{b}{2}$ can be set.

The three points P_{bottom} to P_{right} and P_{left} are now know. Therefore, the triangle can be drawn in a desirable way: The agents position is in the center of mass of the triangle, and the top of the triangle is pointing in the same direction as the velocity of the agent. In this way it is possible to easily see which way an agent is heading, and where the agent's actual position is.

A.5. Pathfinding

One big issue with having obstacles in the simulation, is that agents must be aware of the environment, where the obstacles are placed. This is solved using an external shortest path method that calculates the shortest path towards a specific target. The returning path will not cross any obstacles or go outside the simulation boundaries and is guaranteed to contain the optimal path. There is also a complexity problem, because of the nature of shortest path algorithms, it can be very hard to get an optimal path and compute it in a limited amount of time.

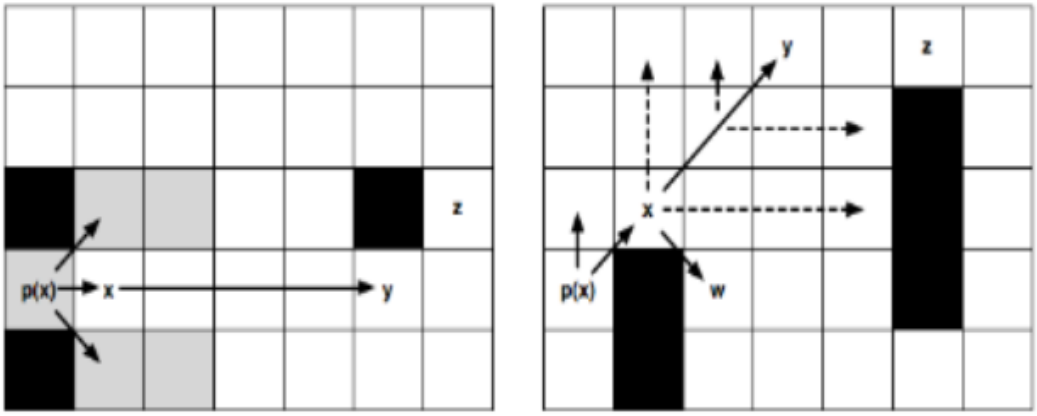


Figure A.8: *Showing a straight jump point to the left, and a diagonal jump point to the right.*

The algorithm used for finding the shortest path is called Jump Point Search [5]. Which is a new, very smart and fast algorithm. It finds specific points, "Jump Points", as targets towards the end position. From the jump points the algorithm expands the graph and searches for other jump points or if the current node is the goal. Selected jump points can be seen in Figure A.8.

A method used for performance improvements is upsampling of positions. Each position that the algorithm works with, is represented by 10 "real" positions, to make the computation much more faster. The downside of this is that the resulting path is a little more coarse than it would normally be if run with normal positions. The upsampling amount can be changed easily by changing a constant for decreasing or improving performance. The pathfinding is also implemented to be lazy. It first checks if theres any obstacle between the goal and start before it runs the shortest path algorithm, if there isn't any obstacle between the points, there is no need to do the shortest path algorithm, since the shortest way always is a straight line between the points.

A.6. File-System implementation

The Simulation program has support for saving and loading simulations to a simulation file. The simulation file is a custom made file type, of which has document object model [14] syntax similarities. The file consists of a header part that contains simulation specific settings, such as obstacle, simulation shape and dimensions. The rest consists of frame-by-frame data in the form of agents in each row, with coordinates, velocity, interaction range and colour. Every character in the saved files is encoded in UTF-8 to reduce the amount of space it takes storing it.

Different maps can also be saved to the disk using our map editor. The syntax of these

files is also simple. The first row in each file contains the name of the map, and the following rows contain one obstacle each. If any file is corrupt or in the wrong encoding, it ignores it.

A.7. Map Editor

The Map Editor is a semi-standalone program, that can be used to graphically create simulation maps. It features some basic functions regarding obstacles, such as adding, removing and changing obstacle properties. One other feature is exporting and importing maps to the editor, it allows the user to view them and edit them as normal maps.

When the user has finished one map, she may choose to save them. If she does, the map will then automatically be imported to the simulation system as a choosable map to run with as the other ones. The user interface of the map editor is shown in Figure A.9.

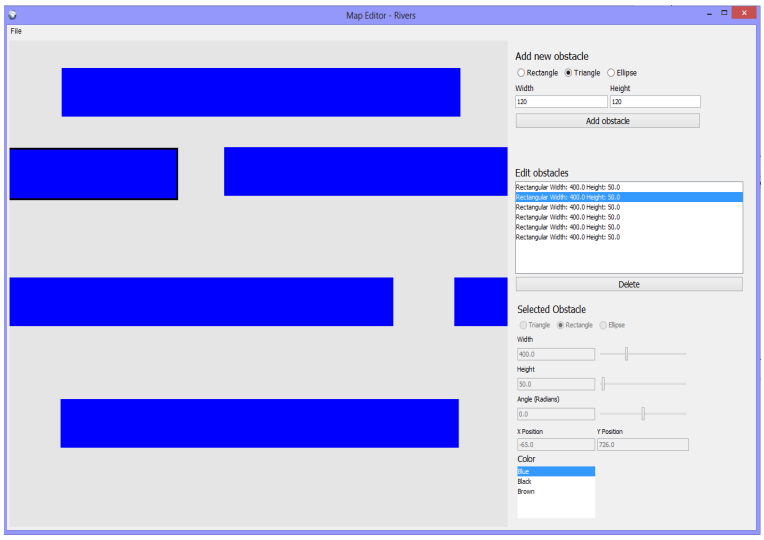


Figure A.9: *User interface of the Map Editor*

B. Sketches

B.1. GUI-Sketches

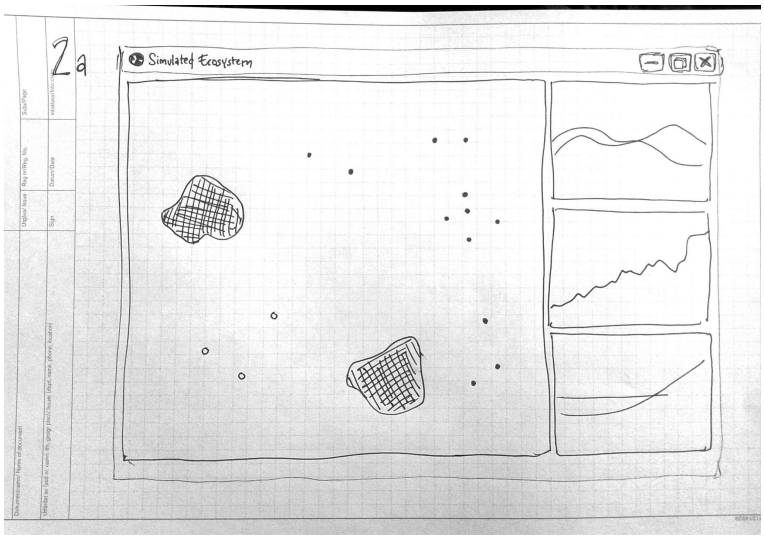


Figure B.1: *An early sketch of the simulation view.*

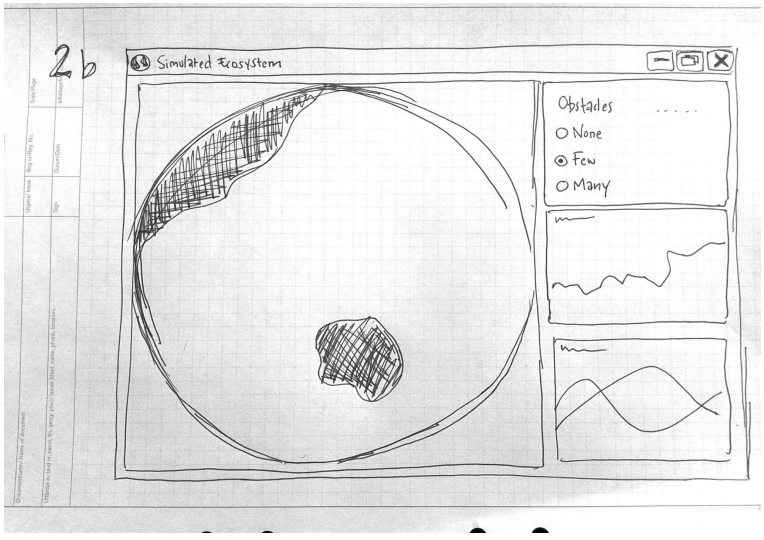


Figure B.2: *Another example of an early sketch of the simulation view.*

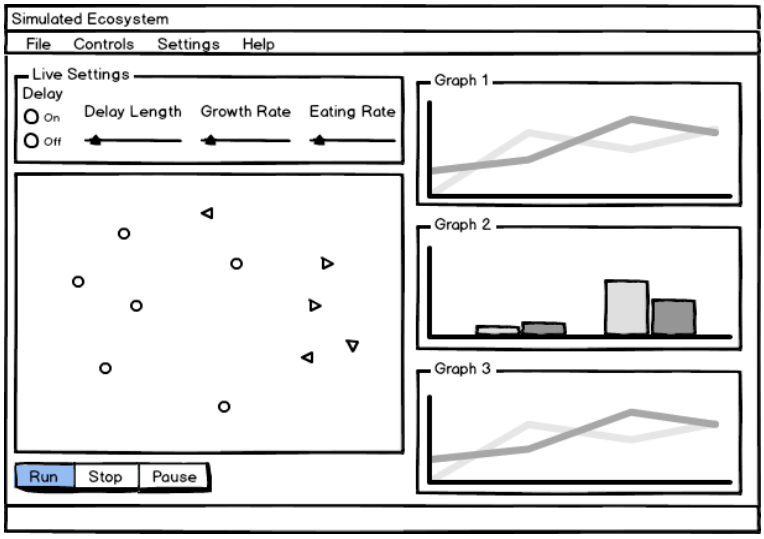


Figure B.3: *A more refined sketch of the application interface.*