BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PREDATOR-PREY SIMULATION IN JAVASCRIPT

SIMULÁCIA DRAVEC-KORISŤ V JAVASCRIPT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JOZEF MÉRY**
AUTOR PRÁCE

SUPERVISOR **Dr. Ing. PETR PERINGER**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Méry Jozef**
Programme: Information Technology
Title: **Predator-Prey Simulation in JavaScript**
Category: Modelling and Simulation

Assignment:

1. Analyze the simulation models of predator-prey dynamical systems and methods of their implementation. Focus on "boid" class of models.
2. Design the application for simulation of predator-prey models with suitable visualization of simulation results.
3. Implement the simulator in JavaScript language. Test the application in Electron environment and common WWW browsers. Create set of (at least 5) demonstration models suitable for educational purposes.
4. Evaluate the project results and propose possible future improvements.

Recommended literature:

- Reynolds C.: "Flocks, herds, and schools: a distributed behavioral model". In Proceedings of the SIGGRAPH'87 Conference, vol. 4, pp. 25-34, 1987.
- Macal C., North M.: "Agent-based modeling and simulation". In Proceedings of the 2009 Winter Simulation Conference, IEEE, 2009.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Peringer Petr, Dr. Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: May 13, 2020

Abstract

Studying the populations of organisms is a useful research field. It can help predict, understand, and possibly help preserve populations. This bachelor's thesis focuses on modeling and simulating an agent-based predator-prey model. The agents' essential traits include flocking based on C. Reynolds's boids model, reproduction, and mutation. This thesis's result is a configurable browser-based application, which can be used to simulate different scenarios and extract various statistical information. The application contains setting templates that were simulated. The results of the simulations are briefly evaluated.

Abstrakt

Študovanie populácie organizmov je veľmi užitočná oblasť výskumu. Dokáže pomôcť s predpovedaním, porozumením a možným zachovaním populácií. Táto bakalárska práca sa zameriava na modelovanie a simulovanie modelu dravec-korisť založeného na agentoch. Medzi najpodstatnejšie vlastnosti agentov patrí schopnosť zhlukovania založená na modeli boids vytvorený C. Reynoldsom, reprodukcia a mutácia. Výsledok tejto práce je konfigurovateľná prehliadačová aplikácia, ktorú možno použiť na simulovanie rôzných situácií a získanie štatistických údajov. Aplikácia obsahuje šablóny nastavení, ktoré boli simulované. Výsledky simulácií sú krátko zhodnotené.

Keywords

modeling, simulation, agent-based model, predators, preys, boids model, flocking, organism interaction, organism mutation, population study, web application, React.js, TypeScript, Electron.js

Kľúčové slová

modelovanie, simulácia, modelovanie založené na agentoch, dravce, koriste, model boids, zhlukovanie, interakcia organizmov, mutácia organizmov, študovanie populácií, webová aplikácia, React.js, TypeScript, Electron.js

Reference

MÉRY, Jozef. *Predator-Prey Simulation in JavaScript.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Dr. Ing. Petr Peringer

Rozšírený abstrakt

V posledných rokoch vedecké disciplíny čoraz viac využívajú modelovanie a simuláciu. Modelovanie a simulácia sú veľmi dôležité a užitočné nástroje na porozumenie nášho sveta. Umožňujú riešenie nespočetného množstva problémov. Príklady takýchto problémov môžu byť: optimalizácia dopravy, predpoveď počasia, analýza obchodného trhu alebo štúdie o populácií. Táto práca sa zameriava na študovanie dynamického systému, ktorého definujúce prvky sú: vzťah dravec-korisť a schopnosť zhromažďovania.

Hlavnými myšlienkami vzťahu dravec-korisť sú dve, protikladné bytosti: dravce a koristi. Vzťah týchto bytostí je, že dravce konzumujú koristi a naopak, koristi sú konzumované dravcami. Intuitívnym príkladom takýchto organizmov sú líšky a zajace. Avšak táto myšlienka platí aj na rastliny. Príkladom takéhoto prípadu sú medvede a bobuľovité plodiny. Z toho vyplýva, že vzťah dravec-korisť sa dá nazvať aj ako vzťah producent-spotrebiteľ. Jeden z najskorších a zároveň najvýznamnejších modelov, ktorý popísal tento vzťah je Lotka-Volterra model. Je to nelineárny, matematický model, ktorý sa vzťahuje na všeobecných dravcov a koristi. Taktiež tvorí základ pre mnoho ďalších výskumov aj mimo štúdie populácií.

Zhlukovanie alebo zhromažďovanie je prirodzený jav. Je to výsledok správania každého organizmu založeného na lokálnom vnímaní. Kým správanie celej skupiny sa javí ako organizované, jedince sa chovajú chaoticky a nepredvídateľne. C. Reynolds popísal takéto chovanie v jeho modely boids. Je založený na abstraktnom vnímaní a troch pravidlách chovania. Ich kombinácia spôsobí komplexné a výpočtovo prístupné zhromažďovacie správanie.

Prvým cieľom tejto práce je analýza systémov dravec-korisť. Z tohto dôvodu sa práca zaoberá dôkladnou diskusiou skoršie spomenutého Lotka-Volterra modelu. Jedná sa o nelineárný, matematický model, ktorý obsahuje všeobecné dravce a koristi, pričom predpokladá nekonečnú potravu pre koristi. Model predpokladá cyklický vzťah populácií, ktorý, avšak závisí len od prvotného stavu. Ak by nastal akýkoľvek externý podnet, cyklický vzťah by sa zmenil bez možnosti návratu k pôvodnému. Teda stabilita modelu je slabá a jeho výsledky sú otázne. Napriek tomu bol základom pre mnoho ďalších výskumov. Ďalej je spomenutá alternatíva modelu Lotka-Volterra s názvom Holling-Tanner. Tento model je opäť vyjadrený matematicky a taktiež sa zoberá s všeobecnými bytosťami, avšak obsahuje väčšie množstvo parametrov. Model je spomenutý len okrajovo s krátkym porovnaním.

Ďalším cieľom je diskusia rôznych spôsobov implementácie modelov dravec-korisť. Z tohto dôvodu práca obsahuje všeobecný postup modelovania, ktorý je prezentovaný ako množina krokov. Ďalej sú popísané tri špecifické modelovacie prístupy: matematické modelovanie, bunkové automaty a modelovanie založené na agentoch. Spomenuté kroky možno aplikovať na akýkoľvek prístup, keďže prístup sa vzťahuje na vyjadrenie a reprezentáciu myšlienok, prvkov a vzťahov v modeli.

Výsledkom práce je aplikácia spustiteľná v prehliadači. Vďaka nástroju Electron.js je spustiteľná aj priamo v prostredí operačného systému. Na implementáciu boli využité rôzne moderné nástroje ako Node.js, React.js alebo JavaScript s rozšírením TypeScript. Aplikácia je schopná simulovať a vizualizovať model dravec-korisť, ponúka ovládacie prvky simulácie, umožňuje parametrizáciu simulácií, obsahuje preddefinované šablóny parametrov a zbiera a prezentuje štatistické informácie. Relevantné informácie a dôsledky možno vyťažiť z aplikácie behom minút. Implementovaný model obsahuje tri bytosti: dravce, koristi a potravu koristí. Teda vzťah producent-spotrebiteľ sa v modeli nachádza dvakrát. Dôležité schopnosti dravcov a koristí sú starnutie, hladovanie, reprodukcia a mutácia. Koristi sú okrem týchto schopné aj zhlukovania založeného na modely boids. Bytosti existujú v dvojrozmernom priestore.

Preddefinované šablóny boli simulované a výsledky sú krátko zhrnuté. Dôležitý výsledok je, že zhlukovanie zlepšilo populáciu koristí, avšak nečakaným spôsobom. Zlepšnie možno nájsť v pomere dravcov a koristí. Bez zhlukovania mať stabilný stav znamená mať asi trojnásobne viac koristí ako dravcov. Zhlukovanie vylepšilo pomer na približne štvornásobok. Je to pravdepodobne spôsobené menším rozptylom koristí v priestore, a teda je nižšia šanca, že ich dravec nájde. Ďalší zaujímavý výsledok možno pozorovať s povolenou mutáciou. V tomto prípade boli populácie najviac chaotické. Cyklický vzťah možno pozorovať aj pri niektorých vlastnostiach bytostí.

Z aplikácie možno vyťažiť množstvo užitočných informácií a vyvodiť z nich dôsledky. Napriek tomu je veľký potenciál na ďalšiu prácu. Nakoniec, práca obsahuje navrhované vylepšenia, ako napríklad modelovanie pohlavia, zmena z dvoj na trojrozmerný priestor a ďalšie.

Predator-Prey Simulation in JavaScript

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Dr. Ing. Petr Peringer. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Jozef Méry
July 28, 2020

Acknowledgements

I would like to thank my supervisor Dr. Ing. Petr Peringer for his numerous ideas, advice, and help. They helped me greatly with the creation of this thesis. Next, I would like to thank all developers who make their software available free of charge for everyone. Without them, an application like this would be impossible.

Contents

Chapter 1

Introduction

In recent years, various scientific disciplines increasingly utilize modeling and simulation. Modeling and simulation are essential tools that allow studying a vast number of problems, allowing a better understanding of our world. Such problems include traffic optimization, weather forecast, analysis of market dynamics, or population studies. This thesis focuses on the population dynamics in a *predator-prey model* combined with *C. Reynolds's boids flocking model*.

The basic ideas of any *predator-prey model* are that it contains two distinct, opposing organisms: *predators* and *preys*, and that *predators consume preys*. One of the earliest predator-prey models is the *Lotka-Volterra model*. It is a simple *non-linear mathematical model*, which became the basis for many other studies in the field of population dynamics, and in others as well. [12]

Flocking or swarming is a common phenomenon in nature. It is the result of the behavior of every individual organism based on its local perception. While the group's behavior as a whole seems to be organized, individuals behave chaotically and unpredictably. *C. Reynolds* described such behavior in his *boids model*. The *boids model* is based on three simple *locomotion rules* and an *abstract perception model*. [17]

The result of this thesis will be a *browser-based* application capable of visualizing the simulation of a *predator-prey model*. Entities or organisms will be *generic* and their signature capabilities will include *flocking* and *mutation*. The application will allow loading and customizing *pre-loaded settings*, and will also gather and present *statistical information* about the entities. All the pre-loaded model settings will be simulated, and their results briefly evaluated.

The text of this thesis is divided into multiple logical units. Chapter 2 provides the *fundamental ideas* of modeling and simulation, defines the *process of modeling* and *model iteration*, provides *different methods of modeling* with example use cases, and discusses two specific mathematical *predator-prey models*. Continuing with the theoretical base, chapter 3 discusses the *boids model* in detail. Another model discussed in chapter 3 is a *physics and locomotion model*, which is required by the *boids model* for fulfilling the flocking rules and other objectives. The *initial design* for the resulting application can be found in chapter 4. It contains the *modeled problem formulation*, defines the *important elements* and their *connections*, proposes *element representation* using a *class diagram*, and provides a simple *user interface mockup*. Next, chapter 5 lists the most important tools utilized during development, contains a thorough description of the *implemented model*, and presents *visualization* results. *Results* of simulations with various settings, and proposals for possible *improvements* and *future work* are discussed in chapter 6.

Chapter 2

Modeling and simulation

The following chapter describes the basic idea of *modeling and simulation*, discusses why it is an essential tool, defines the *process of modeling* and *model iteration*, presents various *methods of modeling*, and lists *practical examples* of problems solvable using modeling and simulation. The *Lotka-Volterra* and the *Holling-Tanner* mathematical predator-prey models are also discussed.

2.1 Fundamental concepts

Let us discuss the *fundamental ideas* of modeling and simulation before we dive deeper into specific modeling methods.

The *fundamental idea of modeling* is creating a model that is an intentionally *simplified representation* of reality. This simplification is also referred to as *abstraction*. Because the world we live in is very complicated with many moving parts and variables, the goal of abstraction is to aid understanding a small piece of it by removing some of the variables, while keeping those, whose impact on a system we wish to study. In this context, a *system* is the abstraction of reality containing various entities that can act on the system, thus changing its state. [15]

After modeling comes *simulation*, which is the *execution* of a given model or „putting it into motion“. The simulation's goal is to *closely reproduce* a real system's behavior while possibly producing varying results for different inputs. Simulation results are *observations of impacts* on the system with changing inputs, or *predictions* about how the system's real counterpart will evolve. Acquired results can be compared with the real system to *improve the model* and its accuracy. The process of improving the model is referred to as *model iteration* or *calibration*, which can sometimes utilize historical data. Enabled by today's very powerful computer systems, simulation is a potent and essential tool that allows us to understand better the world we live in, anticipate the outcome of our actions, and discover connections in a complex system. [15]

Due to computers capable of simulations emerging only in the *20th* century, simulation is a relatively new field of research. The earliest examples of simulations come from technical fields, such as simulating a building's structural integrity. With the rapid growth of computing power in recent years, simulations are now applicable to *more complex* and *diverse* problems. More data is available, it is possible to process it better and more accurately, and better visualization techniques are available. Use cases include weather forecasts, traffic simulations, and *entity interactions*, which is an important part of this thesis. [15]

2.2 Modeling process

Modeling is the creation of a *simplified* and *inaccurate representation* of reality. The following section discusses the *process of modeling*. The process refers to steps or phases that result in a model.

Despite various existing techniques, the basic idea of modeling can be divided into six phases [15]:

1. *formulating the problem* (see 2.2.1)
2. *designing the model* (see 2.2.2)
3. *implementing the model* (see 2.2.3)
4. *verification and validation* (see 2.2.4)
5. *simulation and analysis* (see 2.2.5)
6. *processing the results and formulating answers* (see 2.2.6)

2.2.1 Formulating the problem

„The scientific mind does not so much provide the right answers as ask the right questions.“ (Claude Lévi-Strauss)

The basic principle of modeling is to model a *problem*, not a *system*. What ends up being modeled is, in fact, a *system*. This principle refers to the *usability of a model*. Therefore in this phase, we ask the question: „What problem is the model attempting to solve and what questions need to be answered?“. We attempt to *formulate the problem* as precisely as possible, choose the model *time domain*, and consider the *available resources*, such as time and finances. [15]

As an example, let us consider the problem of aircraft flight modeling. There are many possible approaches to formulating this problem, each resulting in drastically different models. One of the possibilities is to model a single specific flight scenario that would attempt to answer how a real aircraft would behave under various conditions. The essential elements of such a model could include the power of aircraft engines and the current weather conditions. The time-domain for this model could be minutes, hours, or days at most. A completely different possibility of formulation could be the modeling of expected aircraft lifespan. In this model, the essential elements could be usage and maintenance intervals. The time-domain could, in this case, be weeks, months, or even years.

2.2.2 Designing the model

In the early stages of model design, we attempt to keep the model very simple by *abstracting* away details and adding them later as necessary. We define the *most significant modeled elements*. These elements collectively define what the model is going to consider. Another consideration we have to make is in what *detail* we are going to model and represent each element. Next, we define the *connections* between individual elements focusing on the *relationship* rather than how exactly are said elements related. [15]

To continue with the aircraft flight model, let us assume we formulated the problem as a specific flight scenario modeling. Essential modeled elements could include modeling various physical forces, weather, fuel type, fuel quality, or engines' power. After defining

these elements, we have to decide on the details of their representations. Do we represent the engines as a single number, or do we consider its effectiveness, various types (jet, propeller), size, or other parameters? We proceed to define the individual element connections. In this example, the engine effectiveness could be related to the fuel type and quality.

2.2.3 Implementing the model

In this step, we choose the right *tool* for the job. Possibilities include existing applications like *MATLAB*[1] or writing a suitable program from scratch, possibly utilizing existing libraries or other tools. [15]

We then *define the representation* of individual modeled elements. For a *computational model* to be usable, we must define all the properties and connections of every element in the model, including the input parameters. These can be chosen based on *observations*, *historical data*, *estimating*, or even *randomly*. [15]

Continuing with the flight model example, in this step, we would accurately define the representation of each element we designed with all their connections. We could define the relation between engine effectiveness and fuel quality using an equation.

2.2.4 Verification and validation

When the implementation is finished, we have to determine whether the model is suitable for answering the original question. *Validation* is the process of confirming that the model reflects the behavior of the real system. [15]

Verification is the process of confirming that the implementation does what we expect to or confirming that the implementation reflects the abstract model's behavior [15]. In our example, verification could be an observation that engine effectiveness decreases with lower-quality fuel.

While *verification* can be done using well-defined steps, *validation* is much more difficult. The fundamental problem with models is that they are *always faulty* and can never entirely accurately reflect reality. Because of this, validation is often subjective and is based on *usability* rather than accuracy. [15]

2.2.5 Simulation and analysis

After we created a model we deem suitable, we *analyze* it. *Analyzing* is based on the specific problem, but it usually means finding the *role* of each element in the model, finding the *most impactful element*, and learning how the model *behavior changes* with varying inputs. [15]

We could analyze the aircraft flight simulation by asking the following questions:

- How does the weather affect flight stability?
- How does the traveling speed affect fuel consumption?
- How does the traveling speed affect the aircraft turning radius?
- What is the relation between air density and fuel consumption?

[1] https://www.mathworks.com/products/matlab.html

2.2.6 Processing the results and formulating answers

We study the *results* gained from the simulation analysis, and we attempt to *answer* the original questions. Modeling is an *iterative process* – if needed, we return to previous steps to possibly improve the model and its results. [15]

2.3 Different methods of modeling

Not all problems can be modeled well with a particular modeling method. Due to the popularity of modeling, many different modeling methods were invented, greatly increasing applicability. This section presents a non-exhaustive list of different *modeling methods* with their respective *properties* and possible *use cases*. The modeling steps mentioned in section 2.2 *can* be applied to any of the methods.

2.3.1 Mathematical modeling

Mathematical modeling is an example of the „top-to-bottom“ modeling approach that focuses on the whole system's structure and *quantity summaries* with *explicitly defined feedback* [14].

Mathematical modeling is based on the following *principles* [14]:

- The state of the system is expressed with *state variables*, whose summary represents certain information, such as the number of individual organisms, temperature, or currency exchange rate.
- The system behavior is defined using *equations* that contain the state variables. These equations define how the variables change in either *discrete* or *continuous* time.

There are two options for *solving* mathematical models [14]:

- *Exact mathematical analysis* leading to a generic solution or to values for which the behavior of the model stabilizes.
- *Numerical method* that begins with certain initial variable values and determines the behavior using numerical calculations (simulation).

Equations that define the variable value changes in discrete time are referred to as *recurrence relations*. They define the state of variables in time $t + 1$ based on previous variable states in the time interval 0 to t. If the variables states in time $t = 0$ are known, it is possible to calculate their state in time $t \in N$ using *substitution*. [14]

When it is not practically possible to divide time into discrete steps, *differential equations* express the change or difference (hence the name differential) of state in time t. Simpler differential equations allow *analytical solving*, resulting in an exact solution, while complex ones can be solved using *numerical methods* with reduced accuracy. Numerical methods are based on *time discretization*. [14]

Examples of problems solvable using mathematical modeling include *solidification and melting*, *population dynamics*, or even *tumor growth*. [14]

2.3.2 Cellular automaton based modeling

Cellular automata are *discrete* and *dynamic* systems based on a *regular grid*. In theoretical analysis, the grid is considered *infinite*, but it has to be *finite* in a computer simulation. Every unit in the grid is an *identical cell*, which is in *one* of a finite number of states. The dynamics in a cellular automaton are based on an individual cell's *neighborhood* (usual neighborhoods are illustrated in figure 2.1) and a finite set of *fixed rules* shared by all cells. [20]

In computer simulations, the set of rules has to include *edge-case rules* defining the behavior of cells on the edge of the grid [13].

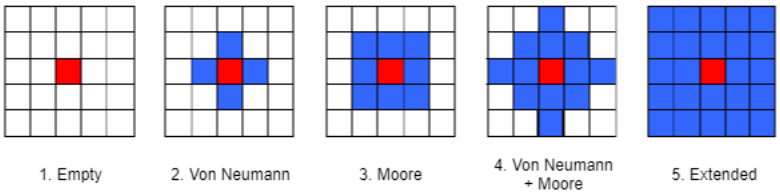

Figure 2.1: Different *neighborhood* options in cellular automata. Based on figure from article [3], edited.

The state of cells changes in *discrete* time steps meaning that updating is done on every cell *simultaneously*. In practice, this usually means that a *state snapshot* is required every update cycle, based on which every cell updates its state. [13]

Cellular automata are a case of *emergent behavior* systems because the cells themselves do not contain explicit information about the system's behavior. The system's behavior does not depend on its elements, but the element *relationships*. Because of this, the behavior of the system cannot be predicted by examining its elements. [9]

Example uses of cellular automatons include *traffic modeling*, *musical composition*, *spreading of a disease*, *crystallization*, *pattern emergence* modeling, and others [22][13]. Perhaps the most famous cellular automaton model is called the *Game of life* created by *John H. Conway*. Despite very simple rules[2], the model is able to produce exceptionally complex patters and behavior (examples of stable patters shown in figure 2.2).

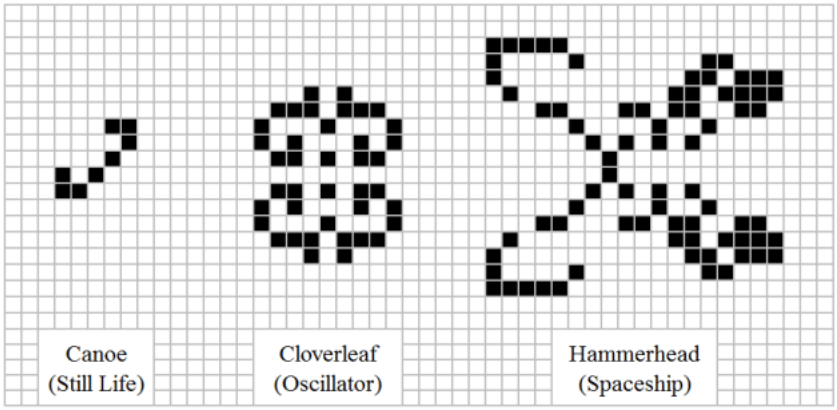

Figure 2.2: Examples of stable patterns in Conway's Game of life. Taken from article [8].

[2] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Rules

2.3.3 Agent-based modeling

Agent-based modeling and simulation (ABMS) is a new approach to modeling that has gained popularity in recent years. This increase in popularity is caused by the *increasing complexity* of problems that require analyzing and solving using modeling and simulation, its *broad spectrum of applicability*, and the rapid *growth of computational power*. ABMS also allows the modeling and simulation of systems that were previously too complex. [6]

An *agent-based model* (ABM) is most commonly made up of individual units - *agents*. While a strict definition for an agent does not exist, in the context of ABMS, agents are usually defined as *discrete*, *heterogeneous*, and *dynamic* subjects. Agents are also *autonomous* and *self-directed* - they can make decisions on their own based on their environment and interactions with other local agents. Agents usually store information about their *resources*, such as health, energy, or others. Agents have *behavior rules* or *goals* that define how they interact with each other and make decisions, possibly considering their resources. It may also be possible for agents to *learn*. They may be able to adapt and adjust their behavior based on previous experiences. [6]

A necessary part of an agent-based model is *randomness* and *heterogeneity*. Agent-based models require *agent diversity*, which can mean different types of agents or attribute diversity among individuals. An example of different types of agents is predators and preys, while attribute diversity may be a varying life expectancy value. An ABM usually contains a *large number* of agents with *simple behavior* rather than a small amount with complex behavior. [13]

Agents may exist in an *environment*. It can have its *parameters and behavior* or *interact* with agents. In a computational model, the environment must be *finite*. The environment is usually modeled as one of the following [13]:

- *Euclidean geometry*, which is usually up to 3D. In a finite *Euclidean geometry*, agents are usually wrapped around the edges.
- *Regular grid*, similar to cellular automata (see 2.3.2). The important difference is that ABMs are *heterogeneous* as opposed to homogeneous.
- *Chart*, which is useful when the emphasis is on the *connection* between agents and their physical location is insignificant.
- *Point*, when all agents are in a *single location*, possibly interacting with any other agent.

Similar to cellular automata (see 2.3.2), agent-based models (ABMs) may also exhibit *emergent behavior*. Simple, deterministic rules may cause the emergence of various *sustainable patterns*. Emergent behavior may have implications on the *development* and *interpretation* of ABMs, because it may be present in more complex models representing *real-world phenomena*. [6]

Practical examples of ABM applications include *analyzing the spread of epidemics*, *understanding the human body*, *studying consumer behavior*, or *improving supply chains* [11]. As another example of an ABM, figure 2.3 presents the visualization of *Thomas Schelling's (1971) segregation model*[3]. The top figure shows the initial configuration, and the bottom figures are different results based on the *similarity threshold*.

[3] http://nifty.stanford.edu/2014/mccown-schelling-model-segregation/

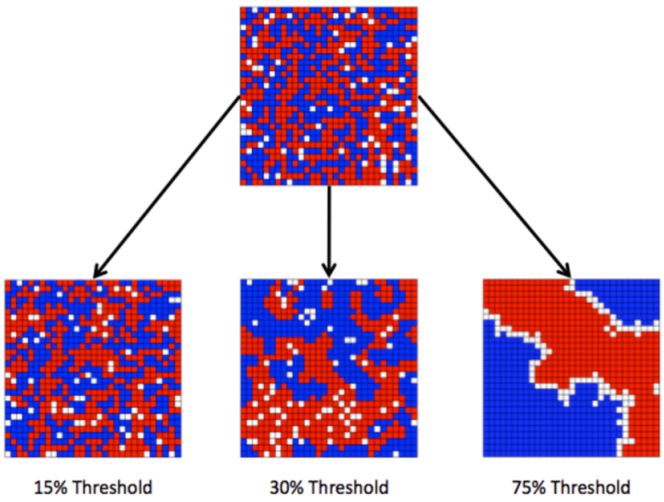

Figure 2.3: Example of an ABM application. Taken from article [4].

2.4 Predator-Prey models

The essential elements in a *predator-prey model* are two *opposing* organisms: *predators* and *preys*. The relationship between these two organisms is that they exist in the *same environment*, the *predators consume preys*, and opposingly, the *preys are consumed by predators*. This relationship forms the behavioral basis of predator-prey models. *Foxes and rabbits* are an intuitive example of animal predators and preys, but the same concept also applies to plants: *bears and berries*. Predator-prey models often model *evolution* or *mutation*. The predators are evolving their properties that are increasing their prey consumption rate. On the other hand, preys are evolving whatever is necessary to prevent being eaten by the predators. The main goal of predator-prey models is to understand the *fluctuations in the respective populations*. [12]

In the following section, the *Lotka-Volterra model* is described in detail, and a more sophisticated alternative is briefly discussed, the *Holling-Tanner model*.

2.4.1 Lotka-Volterra model

An early and basic yet significant predator-prey model is called the *Lotka-Volterra model*, named after Italian mathematician *Vito Volterra* and American chemist *Alfred Lotka*, who worked on the model simultaneously, but independently in the second decade of the *20th* century. It is said that Volterra developed the model idea based on the observations of *Adriatic fishing fleets*. [12]

Let $N(t)$ be the *prey population* and $P(t)$ be the *predator population* in time t. Assuming the preys have *infinite* food, the Lotka-Volterra population model is described with the following *differential equations* [2]:

$$\frac{dN(t)}{dt} = N(t)\,(r - aP(t)), \tag{2.1}$$

$$\frac{dP(t)}{dt} = P(t)\,(caN(t) - q) \tag{2.2}$$

Equations 2.1 and 2.2 have the following *parameters* [2]:

- r - prey growth rate
- a - predator hunting/attack rate
- c - predator efficiency at turning food into offspring
- q - predator mortality rate

$\frac{dN(t)}{dt}$ and $\frac{dP(t)}{dt}$ represent the *instantaneous growth rates* of populations.

Equations 2.1 and 2.2 have the following *implications* [2]:

- With the absence of preys, the population of predators is expected to *reduce exponentially* as described in equation 2.2:

$$\frac{dP(t)}{dt} = -P(t)q$$

- With the absence of predators, the population of preys is expected to *grow exponentially* as described in equation 2.1:

$$\frac{dN(t)}{dt} = N(t)r$$

- Predators cause prey population *reduction* as described by the $-N(t)aP(t)$ member in equation 2.1.
- Preys cause predator population *promotion* as described by the $P(t)caN(t)$ member in equation 2.2.

As illustrated in figure 2.4, the Lotka-Volterra model predicts a *cyclical population relationship*. When the predators have *abundant* food resources (there is a large number of preys in their environment), their population *thrives* - they can survive and reproduce; thus, their numbers grow. With the growth of the predator population comes the *reduction* of the prey population lowering predator sustainability, causing their *starvation* - the predator population *diminishes*. Low predator population numbers allow the *thriving of preys*, and with that, the cycle is *complete and repeats*.

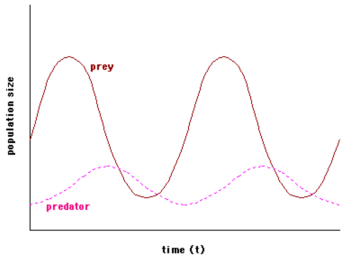

Figure 2.4: Hypothetical cyclical population relationship predicted by the Lotka-Volterra model. Taken from [2].

The *cyclical equilibria* are present in the *continuous-time* version of the model. The respective population numbers at any given time are based on the initial state (assuming

no external stimuli). They oscillate within a closed range *indefinitely*, centered around the *interior steady-state*, creating a closed orbit. These closed orbits, however, are only *weakly stable*. Any external stimulus changes the internal steady state without the possibility of returning to the previous one on its own; thus, the Lotka-Volterra model is *structurally instable*. Weak stability questions the robustness of predictions made by the model. The cyclical equilibria are not present in the *discrete-time* version of the model in which the orbits *spiral outward* instead of being closed. [10]

A more complex yet similar pattern to the one in figure 2.4 is observable in figure 2.5. The population numbers present an approximation of a real predator-prey relationship, as they are based on *pelt purchases*, which is an *indirect* measure. An *assumption* can be made according to which the pelt purchase numbers are directly related to the actual respective populations.

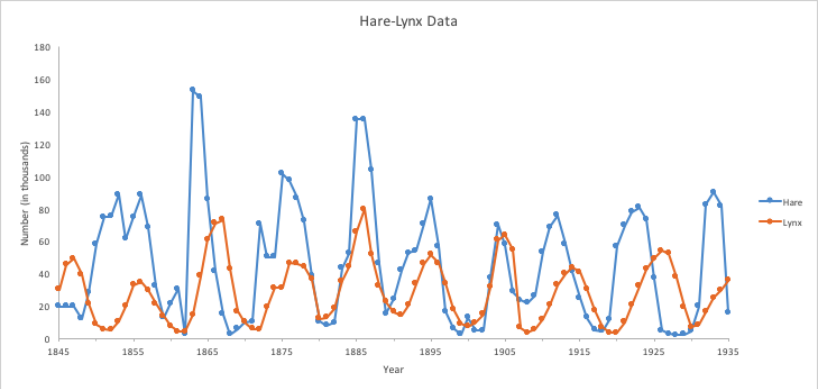

Figure 2.5: An indirect measure of predation based on pelt purchases. Taken from [7].

2.4.2 Holling-Tanner model

The earliest model describing the population dynamics in a predator-prey relationship is the *Lotka-Volterra model* (see 2.4.1). This model pioneered the research of interacting species. However, its usability is questionable. A modification or an improvement over the Lotka-Volterra model is the *Holling-Tanner model*. It also models predators and preys as generic entities, but it has additional parameters. For example, the populations are affected by the *environment* they live in or the *prey's quality*.

Let $N(t)$ be the *prey population* and $P(t)$ be the *predator population* in time t. The Holling-Tanner model is described with the following *differential equations* [21]:

$$\frac{dN(t)}{dt} = rN(t)\ (1 - \frac{N(t)}{K}) - \frac{mN(t)P(t)}{A + N(t)}, \tag{2.3}$$

$$\frac{dP(t)}{dt} = sP(t)\ (1 - \frac{P(t)}{hN(t)}), \tag{2.4}$$

where r is the *prey growth rate*, K represents the *carrying capacity of prey* or the maximum number of prey that can be sustained by the environment, m is the *maximum number of prey consumed by individual predators* in a time unit, A is the *number of preys required to achieve half of the maximum rate* m, s represents the *predator growth rate*, and h is the *measure of food quality* provided to the predators (abstract quality of prey, which may refer to size, nutritional quality, or other properties). [21]

Chapter 3

Boids and entity physics models

Flocks, schools, herds, or other *group behaviors* are made up of *discrete entities*, and yet the overall group motion seems *fluid* and *synchronized*, creating a beautiful and puzzling visual result. Despite the seemingly complex behavior, it is believed that it is the *aggregate result* of each entity interacting with its own perceived part of the world. [17]

Flocking is used throughout this thesis to describe a grouping behavior. Other terms exist that describe the same idea for different species.

This chapter groups together the *boids model* and a simple *entity physics and locomotion model*. Despite sharing the author, they are usable independently. They are grouped in this chapter as the boids model requires entity physics and goal achieving ability. The *boids model*, discussed in section 3.1, aims to efficiently describe flocking behavior with a *set of simple rules* or goals, and an *abstract perception model*. *Entity physics*, *locomotion*, and *achieving goals* are discussed in section 3.2.

3.1 Boids model

The following section is devoted to the *boids model*. It discusses its *requirements*, three fundamental *behavior rules*, and an *abstract perception model*.

The *boids model* was invented by *Craig Reynolds* in *1987*. It was inspired by birds' social behavior, hence the name *boids* (*bird-oids* or *bird-like* objects). The base for the boids model is a *geometric flight ability*, which refers to motion along a path: a *dynamic*, *incremental*, *rigid geometrical transformation* in space. While in real flight, turning and movement happen continuously and simultaneously, an incremental geometric flight is only a *discrete approximation* of real flight. Animating discrete geometric flight requires incrementing the motion at least once every frame. [17]

Natural flocks consist of two opposing, balanced behaviors: *collision avoidance* and *staying close* to the flock. Collision avoidance is a reasonably obvious desire to *prevent damage* within the flock. However, the desire to stay close to the flock seems to be the result of the following factors: *protection* from predators and *survival*, *social behavior* within the flock, and better chances of *feeding*. These factors could be further condensed to *survival* and the *ability to reproduce*. Natural flocks have seemingly *no upper limit* to the number of members due to each flock member basing its behavior on a local flock around itself, or in other words, a limited number of other *closest flockmates*. [17]

Simulated flocks require a model that supports *geometric flight* and the following simple *rules*, whose combination results in a *complex flocking behavior* listed in order with decreasing precedence [17]:

- *separation* (3.1.1) - avoiding collision with local flockmates
- *alignment* (3.1.2) - adjusting velocity to match average of local flockmates
- *cohesion* (3.1.3) - steering to average position of local flockmates

Every rule mentions *local flockmates* that are discussed in subsection 3.1.4.

3.1.1 Separation behavior rule

Collision avoidance or *separation* urges the boids to steer away from each other to prevent impact creating a natural, safe distance between individual boids. Separation and alignment (see 3.1.2) are *complementary*, and together they enable safe flight within the flock's interior. Separation is based only on the *position* of other local flockmates (see 3.1.4), ignoring the velocity. The resulting separation force magnitude is *inversely proportional* to the distance of other boids, urging them to steer away from each other more, the closer they are. [17]

Algorithm 1 describes how to calculate the desired velocity for separation. It takes data structures as inputs that allow retrieving various information about the boid, such as position and velocity. *Self* is the data structure of the boid for which the separation force is being calculated and *PerceivedBoids* (see 3.1.4) is a set of perceived boid data structures. Steering towards the desired calculated velocity ($\vec{separation}$) is described in subsection 3.2.2.

Algorithm 1: Desired velocity for separation

Input: $Self, PerceivedBoids$
Output: $\vec{separation}$
1: $\vec{total} = \vec{0}$
2: $\vec{ownPosition} = position(Self)$
3: **foreach** *boid in PerceivedBoids* **do**
4: $\quad \vec{otherPosition} = position(boid)$
5: $\quad dist = euclidianDist(\vec{otherPosition}, \vec{ownPosition})$
6: $\quad \vec{diff} = \vec{ownPosition} - \vec{otherPosition}$
7: $\quad \vec{scaled} = \frac{\vec{diff}}{dist}$
8: $\quad \vec{total} = \vec{total} + \vec{scaled}$
9: **end foreach**
10: $N = count(PerceivedBoids)$
11: **if** $N > 0$ **then**
12: $\quad \vec{average} = \frac{\vec{total}}{N}$
13: $\quad$ **return** $\vec{average}$
14: **end if**
15: **return** $\vec{0}$

Figure 3.1 illustrates the *separation* behavior rule in *two dimensions*. The grey area represents the *green* (focused) boid's *perception* in a circle centered on its origin. In this

illustration, the perception angle is not considered, equivalent to it being *360* degrees. The green lines can be interpreted as vectors (facing the green boid), which the boid *considers* for separation calculations. Other (blue) boids outside the perception circle are missing the green lines; therefore, they are *not* considered. The result of this rule is illustrated as a red arrow, which represents the *desired velocity* vector. Using the desired velocity vector is described in subsection 3.2.2.

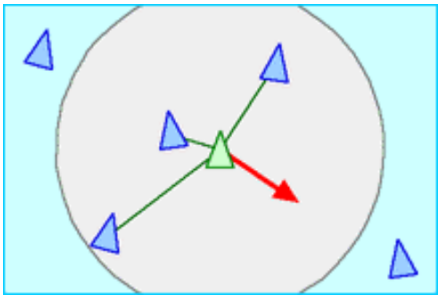

Figure 3.1: Separation based on the position of other perceived boids in two dimensions. Taken from [16].

3.1.2 Alignment behavior rule

Thanks to *alignment*, individual boids tend to have similar velocity angles and magnitudes, enabling the flock to travel in the same direction. As mentioned in subsection 3.1.1, alignment and separation are *complementary*. While separation forces a safe distance between individual boids, alignment helps *maintain* it, because if flockmates have a similar speed and direction, collision is less likely. Calculating the alignment force is based on the *arithmetic mean velocity* of other local flockmates (see 3.1.4). [17]

Algorithm 2 describes how to calculate the desired velocity for alignment with *PerceivedBoids* (see 3.1.4) being a set of perceived boid data structures that allow retrieving information about individual boids. Steering towards the desired calculated velocity ($\vec{alignment}$) is described in subsection 3.2.2.

Algorithm 2: Desired velocity for alignment

Input: $PerceivedBoids$
Output: $\vec{alignment}$
1: $\vec{total} = \vec{0}$
2: **foreach** *boid in PerceivedBoids* **do**
3: $\quad otherV\vec{elocity} = velocity(boid)$
4: $\quad \vec{total} = \vec{total} + otherV\vec{elocity}$
5: **end foreach**
6: $N = count(PerceivedBoids)$
7: **if** $N > 0$ **then**
8: $\quad \vec{average} = \frac{\vec{total}}{N}$
9: $\quad$ **return** $\vec{average}$
10: **end if**
11: **return** $\vec{0}$

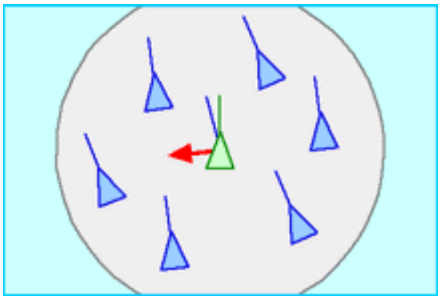

Figure 3.2: Alignment rule illustration in two dimensions. Taken from [16].

Figure 3.2 depicts the *alignment* behavior in *two-dimensional* space. The *focused boid*, for which the rule is illustrated, is colored green. The grey circle represents the focused boid's *perception*. In this example, perception is based only on a radius, while the angle is not considered. Each perceived boid, colored blue, has a blue line - their respective velocities. The focused boid has a green and a blue line representing the *current* and *desired* velocity, respectively. The red arrow represents the *acceleration* vector. Acquiring and using the acceleration vector is described in subsection 3.2.2.

3.1.3 Cohesion behavior rule

Cohesion causes individual boids to steer towards the center of the flock. Due to limited boid perception, it refers to the local center. While deep inside a flock with a mostly *homogeneous* population around it, this force becomes very small, because the boid is already very close to the local center. On the other hand, when a boid is on the *boundary* of the flock, the cohesion force becomes much more prominent and deflects the boid's path towards the local center. Cohesion also enables *flock splitting* to prevent collision with an object in the environment. Splitting is enabled by *limited boid perception* and *absence of an explicitly designated flock leader*. If the boid can stay close to the local flock, it does not care about the rest steering away. For determining the local flock center, it is required to calculate the *arithmetic mean position* of other local flockmates. [17]

Algorithm 3 describes how to calculate the center of the local flock for cohesion. It takes data structures as inputs that allow retrieving various information about the boid, such as position and velocity. *Self* is the data structure of the boid for which the center position is being calculated and *PerceivedBoids* (see 3.1.4) is a set of perceived boid data structures. Steering towards the calculated position ($\vec{cohesion}$) is described in subsection 3.2.3.

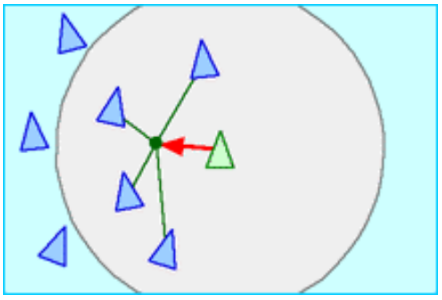

Figure 3.3: Cohesion rule causing steering towards the local center. Taken from [16].

Algorithm 3: AVERAGE POSITION FOR COHESION

Input: $Self, PerceivedBoids$
Output: $\vec{cohesion}$
1: $\vec{total} = 0$
2: **foreach** *boid in PerceivedBoids* **do**
3: $\quad \vec{otherPosition} = position(boid)$
4: $\quad \vec{total} = \vec{total} + \vec{otherPosition}$
5: **end foreach**
6: $N = count(PerceivedBoids)$
7: **if** $N > 0$ **then**
8: $\quad \vec{average} = \frac{\vec{total}}{N}$
9: $\quad$ **return** $\vec{average}$
10: **end if**
11: **return** $position(Self)$

An illustration of the *cohesion* behavior rule is visible in figure 3.3. It is illustrated for the *green* boid. The *perception* area for the green boid is represented as the grey circle. The essential element is the green point, the *average location* of other perceived boids (colored blue, inside the grey circle). The red arrow represents the *desired velocity* vector. Using the desired velocity vector is discussed in subsection 3.2.2.

3.1.4 Boid perception

Rather than simulating real animal senses, the boids model tries to expose a *similar amount* of information to the behavioral model as is available to real animals based on their senses. Giving every boid complete information about its surroundings is *unrealistic*, and it would lead to *mistakes* in the behavior of the model. In an early version of the model, a *central force model* was used, which led to undesired behavior. It caused all members of a flock to converge towards the center even when scattered far apart simultaneously. This early model led to the conclusion that flocking is based on a *limited, local area.* [17]

The *neighborhood* or the local flock of each boid is defined as a spherical zone centered around the boid, inside of which it can sense other boids. The *magnitude of sensitivity* is the *inverse exponential* of the distance because real animals tend to be more influenced more by other animals that are closer. A possible improvement is to *exaggerate* the field of sensitivity in the boid's *forward direction* by an amount proportional to the boid's speed. While in motion, increased awareness of what is ahead is required. [17]

In a software program that implements the boids model, a boid has available information about every boid, including itself. Thus simulating perception is merely *filtering* out redundant information based on some rules for every individual boid. One of the rules should usually be removing itself from the perceived set because the flocking behavior is based on others. [17]

The *perception* of a boid in *two dimensions* is illustrated in figure 3.4. A boid can sense other entities within the area outlined by the *dashed lines* and *curves.* An *angle* and a *radius* define the perception area. The *radius* is calculated from the boid's *origin point*, and the perception *angle* represents the whole field of view. For correct angle interpretation, it is required to add half of it in both directions starting from the angle of velocity.

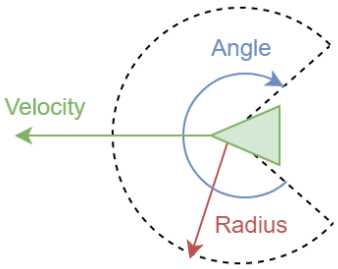

Figure 3.4: Abstract symmetrical perception of a boid in two dimensions.

3.2 Physics and locomotion

The behavior rules listed in section 3.1 listed the „what“, while this section describes the „how“. Specifically, a *basic physics model* enabling *locomotion*, *steering* or applying steering requests to entities, making entities *steer* towards a target (seek a target), and *combining multiple steering requests*.

Despite sharing the author, these models are not directly related to the boids model and are usable on their own.

3.2.1 Basic physics model

For allowing any motion in a model, a *basic physics model* is required. The following model is based on *point mass approximation*. It is intended to encompass a wide range of moving entities on which it can be applied. Thanks to its simplicity and generality, it is also *computationally feasible*. [18]

Each physicalized entity in this model is defined by a *position vector*, a *velocity vector*, a *mass scalar*, a *maximum force scalar*, and a *maximum speed scalar* property. The maximum force and speed are abstract summaries of various *locomotion limitations*. The number of vector components must be greater than or equal to the number of dimensions of the environment entities exist in. [18]

The *physics* in this model is based on *forward Euler integration*. At each simulation step, physics is calculated using the following *formula* for each physicalized entity [18]:

$$
\begin{aligned}
steering\_force &= limit(steering\_force,\ max\_force) \\
acceleration &= steering\_force\ /\ mass \\
velocity &= limit(velocity + acceleration,\ max\_speed) \\
position &= position + velocity
\end{aligned}
$$

The behavioral model may request *steering* represented as a $steering\_force$ vector. This force has its magnitude *limited* by the max_force scalar value. *Acceleration* is calculated by *dividing* the $steering\_force$ with the *mass*. Next, the *velocity* is updated based on the *acceleration* and *limited* by the max_speed. Finally, the *position* is updated based on the new *velocity*.

3.2.2 Steering model

A *steering request* is usually the result of a behavioral model. A steering request can be defined as a request to *change* velocity to a different velocity. An instantaneous change in

velocity would result in an unrealistic looking motion. It would also introduce mistakes to the entity locomotion model. Therefore, change in velocity is modeled as a *steering force*, allowing a smooth transition from current to the desired velocity. Such steering force can be expressed as a vector and may be passed to the *physics model* (see 3.2.1), which transforms it into *acceleration*. The behavioral model may often produce *multiple* steering requests, which would need to be *combined* into one (see 3.2.4), before passing to the physics model. Calculating the steering force is based on the following *formula* (illustrated in figure 3.5) [18]:

$$steering\_force = desired\_velocity - current\_velocity$$

This formula allows an unrealistic *infinite* steering force, possibly belittling any other steering/acceleration request. An improved formula *limits* every individual steering force, which can *improve merging* multiple steering forces [19]:

$$steering\_force = limit(desired\_velocity - current\_velocity,\ max\_force),$$

where max_force is a scalar defining the maximum magnitude of each $steering\_force$ vector.

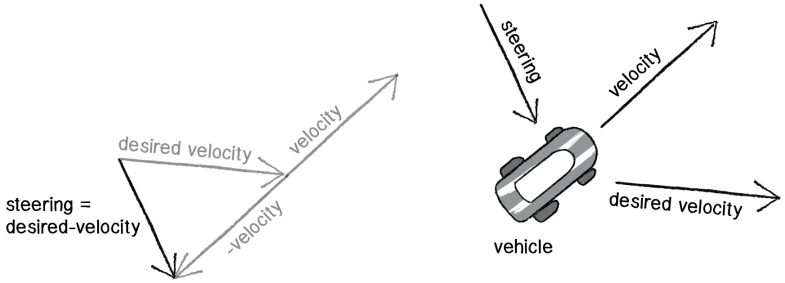

Figure 3.5: Visual representation of steering calculation without limitation. Taken from [19].

3.2.3 Seeking model

Seeking urges an entity to steer towards a particular target or point in space by adjusting its velocity to *align* with the target. The seeking force is *not diminished* even when the target is close, meaning that the entity will fly over it at some point. [18]

The *seeking* ability is based on *steering*. The desired velocity, in this case, is a vector pointing from the boid to the target. It can be calculated using the following formula [18]:

$$desired\_velocity = target - current\_position$$

The result can be used to *steer* towards the desired velocity using the formula described in subsection 3.2.2.

Seeking with a limited desired velocity magnitude is depicted in figure 3.6. The effect of varying *maximum force magnitudes* on a trajectory is shown in figure 3.7. The left figure shows the effect of a *higher* maximum force, aligning the entity's velocity towards the target quicker. The figure on the right depicts the result of a *lower* maximum force. The entity is able to align its velocity with the target after significantly slowing down, resulting in a

spiral trajectory. A significant slow down happens when the target is almost *behind* the entity.

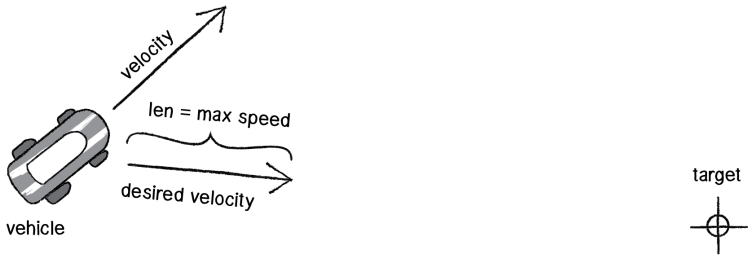

Figure 3.6: Illustration of seeking with a limited desired velocity. Taken from [19].

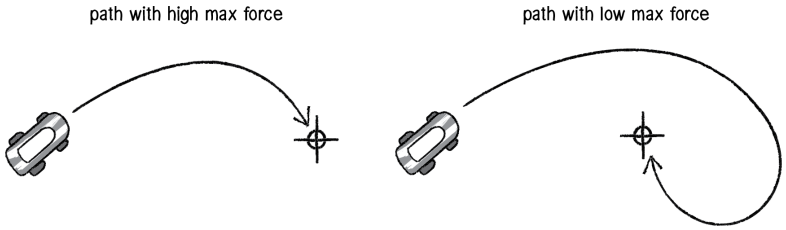

Figure 3.7: Effect of various maximum force magnitudes on steering. Taken from [19].

3.2.4 Combining a set of steering forces

The result of every entity movement rule is a different isolated behavior expressed as a *steering request*. The navigational module of each entity has to *combine* these requests, which includes *resolving conflicts*. *C. Reynolds* proposes two different options for combining forces [17]:

1. *averaging* - the easiest way to combine multiple acceleration forces is to average them. Because every request has a certain magnitude, the result is a *weighted average*. This approach works reasonably well but might not be sufficient during *critical situations*. The problem is with the lack of decision making about which force is to be *accepted* and *discarded*. Some forces might be *canceled out* when they are in approximately *opposite* directions. This indecision can lead to critical errors.

2. *force accumulation* - this approach is based on strict acceleration request *ordering*. Every request is considered in *priority order* and is added into an *accumulator* until the sum of magnitudes is less than a maximum acceleration value. Thanks to this in critical situations, only the *most essential* requests are considered. For example, this would allow a boid to temporarily ignore the urge to flock and leave the flock to prevent collision with an obstacle.

There are several other, more sophisticated techniques for decision making, such as *expert systems*[1] in artificial intelligence [17]. However, discussing such techniques is beyond the scope of this thesis.

[1] https://en.wikipedia.org/wiki/Expert_system

Chapter 4

Designing the application

The following chapter is an overview of the *application design*. I present the *initial*, minimal design based on the first two steps of the modeling process (see 2.2). In section 4.1, I define the *purpose of the model*, and the *questions* that the model is meant to answer (*1.* step of the modeling process). Next, in section 4.2, I list the bare minimum, *most essential elements* that are meant to express the idea of the model. I also define the *element representations* using a *class diagram* and describe the connections between them (*2.* step of the modeling process). Finally, in section 4.3, I present a basic, simple *user interface mockup* that contains the required elements and can show everything mentioned in previous sections.

4.1 Formulating the problem

In the following section, I attempt to capture the model's *fundamental idea*, define the *questions* that are meant to be answered by the model, and *formulate the problem* as precisely as possible.

The assignment has two main requirements for the implementation. One of them is *visualization*, which is discussed in section 4.3. The other is the *predator-prey model* simulation (see 2.4) combined with the *boids model* (see 3). The model also needs to be flexible enough to allow simulating *different scenarios* and produce different results. What problems are to be solved or what observations are to be made are *not* specifically defined in the assignment. The idea behind *predator-prey models* is usually studying the respective *populations*. If we mix in the *boid model's flocking behavior*, the idea of the model becomes clear. The questions that the model is supposed to answer could be:

- *How does flocking affect the respective populations?*
- *How does the lack of flocking affect the respective populations?*
- *How is one population affected with the complete lack of the other?*
- *How are the populations affected with varying perception parameters?*

To summarize, this list is *not exhaustive*. The questions are usually, but not necessarily, related to the effect of *flocking* on the respective populations. Other questions include how a certain property *affects* the populations. The important element found in every question is the *population*. Therefore a more *general question* that captures the essence of the model is:

How does X affect the respective populations?.

Because we wish to study populations, the time domain for this model is *many multiples* of the expected life duration of a single entity. We are not modeling any specific real organisms; therefore, the time domain cannot be expressed with specific time units.

The spatial domain is a *plane*, meaning that the world is defined by *two dimensions*. A two-dimensional domain is more of a design decision than a requirement. It also makes implementation and visualization simpler.

4.2 Designing the model

In this section, I present a *model design* that should be able to answer the questions mentioned in section 4.1. In subsection 4.2.1, I list the *most significant elements* and their *behavior*. I describe the proposed *representation* of these elements in subsection 4.2.2 using a *class diagram*.

4.2.1 Defining the model elements

The most viable approach to modeling the ideas discussed in section 4.1 is an *agent-based model* (see 2.3.3). Thus, the most important elements in the model are *agents* (entities), *agents' behavior*, and the *agent environment*. In an agent-based predator-prey model, there are two important entities: *predators* and *prey*. The *entity behavior* is the result of two separate models: the *predator-prey model* and *the boids model*. This behavior can also be expressed as simpler behavior models (sub-models). The following list defines the *sub-models* for the *boids model*:

- A *2D geometric flight model* (see 3.1) that allows the entities to move in the virtual space. For enabling physics and geometric flight, every individual entity needs to know its *position*, and *velocity* represented by vectors (see 3.2.1). Entities also need to be able to respond to the passing of time, which means they need to be able to update these vector values based on *elapsed time*, creating the illusion of *continuous movement*.
- *Steering* (see 3.2.2) and *seeking* (see 3.2.3) enabling entities to achieve goals, such as flocking.
- Behavior rules: *separation* (see 3.1.1), *alignment* (see 3.1.2), and *cohesion* (see 3.1.3). Combining these rules give the entities the ability to form a flock.
- Because flocking rules are based on other entities, a fundamental requirement becomes *perception* (see 3.1.4). A simple *2D* perception model requires a *perception radius* and a *perception angle*.

The requirements for a *predator-prey behavioral model* can be broken down into the following *sub-models*:

- One of the basic ideas of a predator-prey model is that predators *consume* preys. For enabling this behavior, a simple *hunger model* is required, including keeping track of the *current hunger* and *change in hunger* for every individual entity.

- For a hunger model to make sense, a *health model* is required. When a predator consumes prey, the prey is *killed*. The model assumes that the virtual environment contains *some food* for the prey. Thanks to this, the prey are fed *randomly* based on an interval. Using this model also enables *starvation* of entities when there is a lack of food. For modeling health, individual entities need to keep track of their *current health* and *health delta*. The decrease in health by the health delta value occurs when an entity goes below a certain *hunger threshold* representing *starvation*. In contrast, entities regain health when above a different *hunger threshold* representing a *healing ability* when adequately fed. A health model also enables determining whether an entity is *alive* or *dead*. When dead, it is *removed* from the virtual environment.

- One of the fundamental goals of life is *reproduction*. Because predators consume preys, they require prey reproduction. Otherwise, they would starve to death after eating all the initial prey. A straightforward *reproduction model* requires a *reproduction interval* and a *current progress* property. This model is quite abstracted because it does not involve a *mating process* or the idea of *gender*.

- When a predator wants to consume prey, it has to seek it and then „touch it“ or *collide* with it. For enabling *collision*, an entity has to take up some *space*. A simple collision model only requires a *position* and a *collision radius* resulting in a *circular area* occupied by the entity. *Determining* collision in this model is simply a matter of *comparing* the distance between the two positions with the sum of the two *collision radii*. If the distance is *less than* or *equal* to the sum of radii, the entities *collided*.

Most of the listed *sub-models* are depending on the *passing of time*. In a computer simulation, this can be *approximated* by repeatedly re-calculating (updating) the necessary values as rapidly as possible. Thus in each update cycle, *delta* values (health delta, hunger delta, and others) need to be *adjusted* with respect to the elapsed time. With this approach, the delta values are *independent* of the application frame rate.

The *environment* in this model is rather simple. It has the following *behavior*:

- *Feeding prey* on a *random* basis, which is the abstraction of *containing prey food*.

- *Removing dead entities* from the environment, which is the abstraction of *natural decomposition*.

The environment also has to be limited to a *finite area*. For intuitive and simple visualization, a *square area* should be sufficient.

4.2.2 Element representation and connection design

Based on the individual models discussed in subsection 4.2.1, I designed *classes* that contain the required *information* and *functionality*. Because the environment is modeled as a *plane*, vector types in both classes are appended with *2D*, meaning they contain *two* components.

The *Context* class (as seen in figure 4.1) holds all entities and information about the virtual environment. It allows *adding entities*, *clearing all entities* to possibly set up a new simulation, *modifying the virtual area*, and also provides *statistical data* about their respective counts. It also contains two *update* methods:

- The *drawAll* method that simply displays all the entities registered in the context using the *draw* method of the *Entity* class.

- The *updateAll* method that updates every entity in the context by delegating the updating to individual entities and calling their *update* method. This method provides the *random prey feeding* mechanism too. Because entities do not know the context they belong to, *updateAll* is also responsible for *filtering* out *dead* entities. The updating is based on a *time delta* to make sure updating is based on actual *elapsed time* rather than the application frame rate.

The application would also require a system that creates the *Context* instance, provide an *update loop*, calls the *Context* update methods, and provide a *timing system*. This system is an implementation detail and is not relevant to the model.

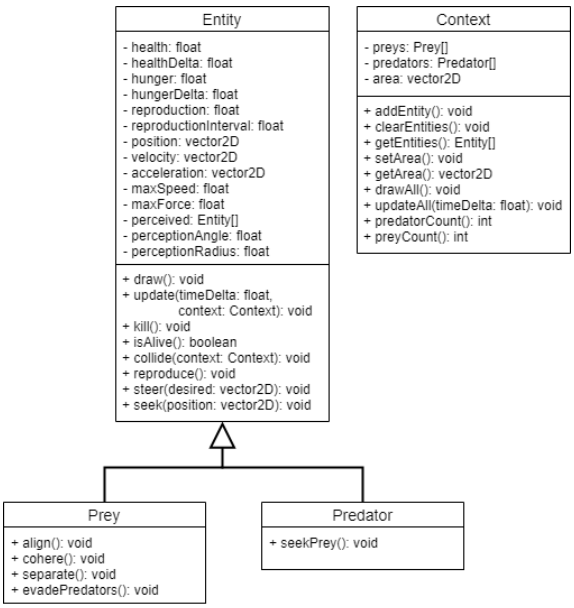

Figure 4.1: Class diagram proposing the representation and functionality of entities and their environment (context).

The *Entity* class (illustrated in figure 4.1) represents a *physicalized*, movable being. With a *basic physics model* in mind, it holds a *position* vector, a *velocity* vector, and *maxSpeed* and *maxForce* scalars. In my model, the *mass* of entities is not modeled (it is always equal to one), and therefore there is no mass class member. An *acceleration* vector is also present to make it available in multiple methods if needed. Besides being physicalized, the entities are also modeled to be *alive*. Therefore, they hold information about their *health*, *hunger*, and their respective *deltas*. Furthermore, they are able to *reproduce* and thus hold information about the current *reproduction* progress and the *reproduction interval*. Entities also need to be able to *sense* other entities. The sensing area is defined by the *perception radius* and *angle* properties. The *perceived* property represents a *cache* of perceived entities for a given update cycle. The most important methods of the *Entity* class are the *update*, *draw*, and *isAlive* methods. They are called by the *Context* in every update cycle. The other methods are present for expressing *functionality* and improving *decomposition*. The *update* method is responsible for invoking other update methods (such as *collide*) and changing the state of the entity or context based on the elapsed time and the context. The *draw* method is straightforward - it *visualizes* the entity. The *isAlive* method simply states whether an entity instance should remain in the context.

The *Predator* and *Prey* classes inherit from *Entity* class, adding their own required functionality, such as *flocking* present only in the *Prey* class.

4.3 User interface design

In this section, I present the application *user interface design*. The purpose of the interface is to *visualize* the simulation in real-time, *set up* a simulation with different settings, *control* the simulation, and display gathered *statistical results*.

The user interface mockup consists of the following *elements* (visible in figure 4.2):

1. A *simulation real-time rendering* displaying entities (rendering figure is based on [5]).
2. *Simulation control and display elements* that enable pausing/resuming the simulation, allow controlling the simulation speed, and display the current speed.
3. *Setup elements* that enable loading specific pre-loaded scenarios and their customization for both entity types individually.
4. A *chart* displaying the number of predators and preys respectively over a certain period. This element also allows *exporting/saving* this chart, making it utilizable outside the application. Chart figure is taken from [2].

To summarize, there are no elements for *closing* or *minimizing* the application as those are provided by the browser the application runs in. The interface is also designed to take up all the *available space* in the browser window. Unless the browser window is too small, vertical or horizontal scrolling is unnecessary to reveal parts of the interface. The placement of individual elements is simple and straightforward. As there are not too many elements, all are visible at all times. During the mockup design, *function* was chosen over form.

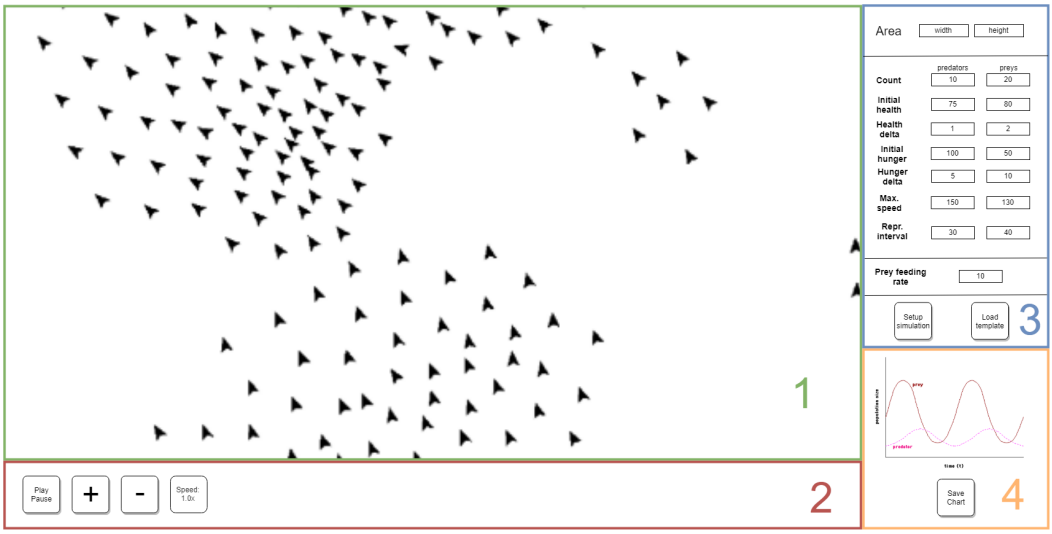

Figure 4.2: A simple and minimalistic application user interface mockup.

Chapter 5

Implementation description

In this chapter, I provide an overview of the *utilized technologies* and a selection of the most *significant libraries*, a *thorough implemented model description*, and *visualization results*.

5.1 Utilized technologies and tools

This section contains a non-exhaustive list of the *most significant tools* that I utilized to develop the application. I provide a short *introduction* to every mentioned technology, and how or why I decided to use it. It is meant to viewed as a list of the most important building blocks that can be used in a *similar application* or an introduction to the *application architecture*.

5.1.1 Node.js and NPM tools

Node.js[1] is a *JavaScript runtime environment* that enables execution of JavaScript code without a browser. Node.js itself has a very simple role in this implementation: *running* a simple server that *serves* the client. Node.js also provides a package manager tool called *Node Package Manager* (NPM), which, on the other hand, was very significant for the development of this application. NPM provides a huge *repository*[2] of various libraries, scripts, and tools that make developing JavaScript applications very compelling.

I utilized NPM for *managing* all *project dependencies* and *running various scripts*, such as building the application (building and running the application is described in appendix A.2), or bootstrapping a React.js application from a template (see 5.1.2). To create a Node.js project managed by NPM, use the following *command* in the desired project directory, and fill out the requested information:

npm init

An alternative to NPM is *Yarn*[3], which can be *faster* in some situations enabled by better utilization of available system resources.

[1] https://www.nodejs.org/
[2] https://www.npmjs.com/
[3] https://www.yarnpkg.com/

5.1.2 React.js component library

React.js[4] is a library for creating *isolated*, *reusable*, *dynamic* components. React.js enables creating dynamic *single-page* applications that do not require reloading to display changes. It includes *component state management*, which is immutable, meaning that state updates are done using a *callback function* that creates an updated copy of the original state. React.js detects changes in state, and applies them to the *DOM*[5] (document object model) only if needed.

I used React to create *state components* that manage and provide their state to visualization components, and *visualization components* that display a certain state by modifying the DOM. I *bootstrapped* the client application using the *create-react-app*[6] command line tool with the following *command*:

npx create-react-app client --template typescript

where the *client* argument represents the name of the project directory. The resulting application supports type checking provided by *Typescript*, and *JSX*[7]. JSX is a language extension that enables *HTML-like* declarative syntax for component creation. To create an application without typescript support, omit the *--template typescript* arguments.

Note: The command is calling a program called *npx*[8], which is an NPM package for *executing package binaries*. When using *npx*, always the latest version of *create-react-app* is being used.

I also utilized a React-based component library called *MaterialUI*[9], which provided most of the user interface components, such as buttons, icons, and menus.

5.1.3 TypeScript language extension

JavaScript is a *dynamically typed*, *interpreted* language. While this allows great code architecture flexibility, it also means there is massive potential for *type-based errors* during runtime.

Typescript[10] is an essential development tool that extends the language by adding *type information* allowing a *static code analysis* preventing a vast number of potential crashes during runtime. The Typescript compiler provides the *type checking mechanism* and also *removes* the type information by *transpiling*[11] it to plain JavaScript. Therefore, the resulting code does *not* contain any *computational overhead* added by TypeScript. Another benefit is that an integrated development environment with TypeScript support can provide *better suggestions* based on types.

5.1.4 Redux.js, Redux Toolkit, and state management

In short, *Redux.js*[12] is a state management library. In contrast with React, it is responsible for managing one, *application-wide* state. Redux by itself requires a lot of *boilerplate code*[13].

[4] https://www.reactjs.org/
[5] https://en.wikipedia.org/wiki/Document_Object_Model
[6] https://www.create-react-app.dev
[7] https://www.w3schools.com/react/react_jsx.asp
[8] https://www.npmjs.com/package/npx
[9] https://www.material-ui.com/
[10] https://www.typescriptlang.org/
[11] https://en.wikipedia.org/wiki/Source-to-source_compiler
[12] https://redux.js.org/
[13] https://en.wikipedia.org/wiki/Boilerplate_code

Instead of plain Redux, I used an opinionated library built on top of Redux called *Redux Toolkit*[14]. It provides utilities for usual Redux *use-cases*, significantly reducing *boilerplate code* and making the resulting code *easier to read* and much more *maintainable*. Redux Toolkit also allows updating the state in a seemingly *mutable-like* fashion by utilizing *Immer.js*[15] under the hood.

In Redux, any *state changes* or updates are defined by *actions* (Flux Standard Action[16]). An action defines *what* should happen, not how. Initiating an *action* is done by *dispatching*, which is usually the result of a *user action* or an *interval callback*. Requested actions are then processed by the *reducer*, which has to *create a copy* of the state, *modify* the copy, and *return* the modified copy. Finally, all subscribed components are *notified* about the state changes and are *updated*. The described flow is illustrated in figure 5.1.

The state in my application can be divided into two categories:

- *Global application state* containing various settings that do not change very often, such as the area dimensions, current simulation speed, hotkey sequences, or the currently selected language. For the management of this state, I used *Redux Toolkit*.
- *Simulation state* containing all the entity instances. This state is managed using *React's context API*[17], and because it changes very often, updating is done in a *mutable* fashion for performance reasons. For the *P5.js* renderer, this is acceptable because it redraws the scene every frame, and thus it queries the current state every frame and does not require to be notified of changes. On the other hand, React.js components that display parts of the simulation state require a *force updating mechanism* that forces them to query the current state and display it without being notified of changes based on *state immutability*.

To summarize, I utilized *Redux Toolkit* to manage the *global application state* that does not change too often.

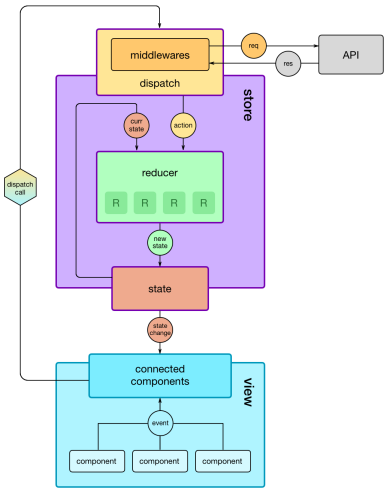

Figure 5.1: Redux.js state update flowchart. Taken from [1].

[14] https://redux-toolkit.js.org/
[15] https://immerjs.github.io/immer
[16] https://www.github.com/redux-utilities/flux-standard-action
[17] https://www.reactjs.org/docs/context.html

5.1.5 P5.js multi-media library

P5.js[18] is a creative multi-media library. It includes drawing tools, transformations, color manipulation, DOM manipulation, input handling, various utility functions, and more[19]. It focuses on *ease-of-use* and *accessibility*. It also provides an *interactive web editor*[20].

I utilized P5.js for the simulation *rendering* in real-time. The simulation is drawn on an *HTML canvas* element. P5.js provides the *loop mechanism*, which is the basis for all drawing and updating within the application. It also provides a *Vector* class, which I used for physics and locomotion calculations.

5.2 Implemented model description

In this section, I describe the *implemented model*. As a whole, the model is complex and best described in terms of its parts (sub-models) and connections between them.

The implemented model is based on the *agent-based modeling* approach (see 2.3.3). Therefore, the essential elements in the model are *agents* and their *environment*. In the design phase, I propose having two distinct generic entities: predators and preys. The final model, however, contains *generic predators*, *preys*, and also *prey food*.

Prey food is a very simple entity. Its *own* behavior is *aging* only. It is able to respond to the *passing of time* and *die* when a specified maximum age has been reached. Its *related* behavior is *feeding prey* and *collision with prey*. When a prey wants to eat and collides with a food entity, the food entity is *killed*. This behavior is specified as *related* because the preys provide the collision and feeding mechanisms. The food entity does not do any collision calculations, nor does it directly interact with other entities. I added the food entity to make the visualization of the simulation more *interesting*, and the movement of prey more *complex*. The food entity is also a direct *replacement* for the random prey feeding mechanism provided by the environment proposed in design.

Before discussing the specifics of predators and preys, I describe their *shared* properties. Both entity types need to ability to *move*. Therefore, I implemented a *custom physics and locomotion model* based on the models discussed in section 3.2. Entities store their *position*, *velocity*, and *acceleration* vectors. The locomotion is limited by three properties: *maximum magnitude* for each individual acceleration request, *maximum acceleration angle*, and *target speed*. An important part of the implemented physics and locomotion model is the ability to *steer* and *seek*. Therefore in every update iteration, the behavioral model of entities may ask the physics and locomotion model to steer or seek a target. Each of these requests has its magnitude *limited* by the *maximum force magnitude*. Multiple requests are combined using the *averaging method* discussed in subsection 3.2.4. This method is simpler to implement, and it is sufficient for this model, as there are no obstacles. Next, the combined result has its *angle limited* to a *maximum acceleration angle* relative to the velocity. Angle limitation is one of the *custom* features of the physics model. It improves flocking and fleeing, especially when entities are heading *towards each other*. In that case, the acceleration request is in the opposite direction of the velocity and would cause a *slow-down* only and no turning. After the angle limitation, comes a custom acceleration request to *regain speed*. It is based on the following *formula*:

[18] https://www.p5js.org
[19] https://www.p5js.org/reference/
[20] https://editor.p5js.org/

$$target\_magnitude = target\_speed - magnitude(velocity)$$
$$adjustment = normalize(velocity) * target\_magnitude$$
$$adjustment\_factor = 1 - (abs(angle(adjustment,\ acceleration))\ /\ PI)$$
$$adjustment = adjustment * adjustment\_factor$$
$$acceleration = acceleration + adjustment$$

The *speed adjustment acceleration* request has a direction based on the *current velocity* and its magnitude based on the *difference* between the target and current speed. For enabling steering, the speed adjustment needs to happen only when *no steering is required* represented by the *adjustment factor*. The adjustment factor is *inversely proportional* to the angle between the adjustment and acceleration vectors. This speed adjustment ability is required because behavioral rules may often cause a *slow-down*. In an early version of the model, entities would often slow down or stop entirely. Occasionally, the entities may also *exceed* the target speed. In that case, the formula causes a *slow-down*. With this custom request, the acceleration calculation is *finished*. For finishing the physics calculation, the *velocity* vector is updated based on the *acceleration* and has its magnitude *limited* to a *maximum speed* scalar. The maximum speed is *not* an explicit property, and it is based on the *target speed*. The maximum speed calculation is currently *hard-coded* and is *10%* higher than the *target speed*. Differentiating between a target and a maximum speed allows the entities to travel *slightly faster* than usual introducing *diversity* and *complexion*. Finally, the *position* is updated based on the *limited velocity*. All the mentioned calculations happen *every update cycle* for both predators and preys based on the elapsed time. In addition, the entities store the calculation results in temporary *draft properties*. With this approach, each entity bases its update on other entities from the *same time step*. After each entity is updated, the draft properties are written to the actual properties. This approach comes with a *computational overhead*, but it is necessary to *prevent mistakes* in the physics and locomotion model. All physics calculations are *floating-point* based. Therefore, *floating-point errors* are possible, but they are *ignored* in this implementation. There are available techniques to prevent such errors, but they come with an overhead, and no such techniques were implemented. The *mass* of entities is not modeled in the current implementation (the physics model assumes that the mass of all entities is normalized to *1*).

Next, both entity types are modeled as *living* organisms. Therefore, I modeled their *health*, *hunger*, *age*, *reproduction*, and *mutation*. The *hunger model* adds *current hunger*, *hunger delta*, and *eating threshold* properties to each entity. All values are expressed as *percentages* instead of specific values. With time, the current hunger simply *decreases* by the hunger delta value. The eating threshold defines the percentage of current hunger, below which the entity attempts to eat, preventing *excessive overeating*. When the entity wants to eat, it asks the locomotion model to *steer* towards the closest perceived food entity. The current hunger property *modifies* the force of steering towards the food entity. Increasing the current hunger after eating is specific to each type and is discussed later. The health model introduces a *current health* property and a *health delta* property. Again, both of them are expressed in terms of *percentages* rather than specific values. An entity is *alive* when its health percentage is greater than zero. When zero, the entity is considered *dead*. Changes in current health happen in three situations:

1. *starvation* - health decreases by the health delta value if the entity's hunger percentage is *0%*

2. *healing* - health increases by the health delta value if the hunger percentage of the entity is greater than *50%*

3. *getting eaten* - when an entity eats its food, the food is *killed* without the possibility of healing (applies to predators eating preys, and preys eating food)

The healing and starvation thresholds are *hard-coded* in the current implementation. One of the obvious missing models in the design is the *age model*. It requires keeping track of the *current age* and the *maximum age* for each entity expressed as properties. The aging is *not* influenced by any other property. When the current age reaches the maximum age, the entity *dies*. In nature, to be alive means to have a specific set of *functioning parts* or *organs*. Therefore, a living organism's maximum age is *not* defined by an arbitrary value. As organs or similar concepts are not modeled, the *maximum age* property is an *abstraction* of said concepts. Finally, the *reproduction model* is crucial for the virtual ecosystem balance. In the current implementation, reproduction is quite *abstract*, meaning that *gender* or *mating* is not modeled. It is based on two properties: the *current reproduction progress* and the *maximum reproduction interval*. First, the entity needs to be in the *required age*. The required age for reproduction is between *25%* and *75%* of the maximum age. Next, the progress counter is incremented based on the following *formula*:

$$reproduction\_progress\_delta = time\_delta * (1 + (hunger \ / \ 50\%))$$

When an entity is *starving*, it is still able to reproduce, hence the maximum reproduction interval property. When well-fed, it may reproduce up to *three times faster*. The mentioned formula is quite abstract and somewhat arbitrary, but it introduces much-needed variety. Finally, when the progress *reaches* the maximum interval, and both health and hunger percentages are over *80%*, exactly one new entity of the same type is added to the environment close to the parent. When the new entity is generated, most of its properties are based on the parent with *intentional mistakes*. These mistakes are referred to as *mutations*. The mutated properties need to be *clamped* to reasonable values. For example, angles are always between *0* and *360* degrees. During testing without the clamping, the entities eventually developed the ability of *negative hunger delta*. Therefore, they „ate“ only with the passing of time. Even though this model is very simple, its impact on the model behavior is *immense* and *subjectively exciting*. The model does *not* contain an explicit *scoring system* for various properties of any kind. „Good“ and „bad“ property values are defined only by the entity's ability to *survive* and *reproduce*, and thus, *pass its properties* to the next generation. This process is referred to as *natural selection*, also described by the *Darwinism theory*[21].

Another component of the entity model is the *collision model*. The collision model is made up of two parts. Part one is the *collision* with the *area boundary*. It is based on the *origin point* of the entity - the position property. When an entity would leave the area (or its origin point is outside the area), it is instead simply *wrapped around* to the opposite end. Part two of the collision model is the *collision of entities*. It is based on the *origin point* and an *arbitrary hard-coded collision radius*. Thus, entities take up a *circular collision area*. Predators and preys have a collision radius of *20* units, and the entity food has *10*. It is somewhat based on the actual drawings of the entities. *Querying collision* in this model is merely finding the distance between the two entity origin points in question and comparing it to the sum of the two collision radii. If the distance is smaller or equal, the entities *touched* or *collided*. The collision is considered only in the *producer-consumer* relationship

[21] https://en.wikipedia.org/wiki/Darwinism

(predators collide with prey and prey collide with prey food). Furthermore, the collision is used only *during eating*. Therefore, when an entity has its current hunger percentage above the eating threshold, collision is *not* considered. Entities of the same type could also collide, but it could make the flocking behavior more of a problem than a benefit. Furthermore, real entities exist in three spatial dimensions and can *partially overlap* when projected to two dimensions.

The final shared component of the two entity types is the *vicinity* and *perception model*. The vicinity is based on the *origin point* and the higher value between the *perception* and *collision radius*. The vicinity serves as an entity *subset* for any calculations involving other entities. While the perception radius value is higher in most cases, it is possible to have an entity *incapable* of perceiving beyond its collision. Calculating the vicinity subset is based on the same principle as collision - *comparing distances*. Perception is based on the entity *origin point*, *radius*, and *angle*. Perceiving based on the radius is, again, based on the same principle as the collision. The perception angle defines the *whole* angle of perception. Half of the perception angle spans in both directions relative to the velocity angle. Thus, entities have *symmetrical perception* relative to their heading (visible in figure 3.4).

The only specific behavior of the *predators* is the *feeding model*. When a predator eats prey, its current hunger percentage is set to *100%*.

Preys have *three* additional behaviors. When feeding, they consider flock mates in their vicinity. When a prey eats and its vicinity has *no other preys*, its hunger percentage is set to *100%*. On the other hand, when its vicinity *has preys*, the eating prey has its hunger percentage increased by *50%*. The remaining *50%* is *equally shared* between all the other preys in the vicinity. This behavior is the abstraction of *improved odds of feeding* in a flock. As the hunger model is directly tied to the reproduction model, food sharing also improves the odds of reproduction in a flock. Without this abstraction, the preys at the back of the flock would be at a *disadvantage*, as preys that are in front of the flock have a much higher likelihood of colliding with the food entities, effectively causing *starvation* at the back. Thanks to this behavior, flocking is more appealing.

Next, preys have the ability to *form flocks*. The flocking behavior is based on *C. Reynolds's boids model* (see 3.1). Each of the three behavior rules has an associated *modifier* value. The modifier values may *mutate*, enabling the *observation of flocking importance* over time, caused by natural selection.

Finally, preys are able to *flee* from perceived predators. The implementation of this behavior is based on the separation rule defined in the boids model.

The last component of the model is the entity *environment*. It is modeled as a *plane* and therefore defined by a *width* and a *height* property. It also keeps track of all the entities in the model by storing their *instances*. It also contains all entities in the sense of *logical space*. As it stores instances, it can provide *statistical information* about their respective properties. The environment enables *setting up a simulation*. Setup involves clearing all previous entities, adding new ones, and resetting statistics. The environment is also responsible for *spawning prey food* entities in a random location. Furthermore, it is responsible for *clearing dead entities*. A unique feature of the environment absent in the design is the entity *regeneration ability*. When the number of entities of a particular type drop below a given threshold, the environment *spawns new entities* of said type based on the initial parameters (all mutations are lost). As the area is finite and relatively small, the *extinction* of an entity type is very likely at some point. Furthermore, as the parts of the simulation are based on randomness, finding *population equilibrium* is challenging and not the focus of the application. With this *optional* abstraction, the simulation may go on

indefinitely in a reasonable manner. The regeneration ability may also be used to spawn the initial entities *gradually*.

5.3 Quad-tree optimization

One of the model's properties is that every entity may access any other entity, as all of them are stored in the environment instance. Entities that base their behavior on other entities have to consider every entity in the environment. Such behavior has *quadratic complexity*: $O(n^2)$. In my model, when an entity considers another entity, it has to calculate their *distance* and *angle*. Calculating the distance is based on the *Pythagorean theorem*[22] containing the *square root* operation, which is relatively slow. In this section, I briefly describe the *quad-tree* data structure used in the implementation, and I present the *results*.

A *quad-tree* is a data structure whose internal nodes have precisely *four* children. It allows the division of two-dimensional space by *recursively subdividing* it into quadrants. The position of entities in my model is defined by a *point*. Therefore, I used a *point quad-tree* that bases the quadrants on points. The divided area causes a performance improvement. When looking up points in a circle (defined by a point of interest and a radius), only the points in intersected quadrants need to be considered, instead of all points. I used an *existing* implementation[23] that allows finding the closest node to a point within a radius. I modified the library by adding the ability to *find all* nodes within a radius. The modification was not implemented by me either, as it is a part of an *unmerged pull request* on the original repository.

The impact of a quad-tree on the simulation performance is illustrated in figure 5.2. Right and left figures show the *FPS* (frames per second) of simulations with and without the quad-tree optimization, respectively. In both cases, there are *500* entities, as the optimization is observable only with a *higher number* of entities. With a deficient number of entities, the simulation may be faster without the quad-tree, as it comes with a *slight overhead*. The quad-tree needs to be recreated every iteration as the entities are moving. The current *FPS*, shown with red, may *fluctuate* due to implementation details and the *single-threaded nature* of JavaScript. The vital information is the *average* FPS. In this specific case, the average FPS is more than *doubled*. Specific results can be influenced by many factors, including hardware configurations, operating systems, browsers, and others.

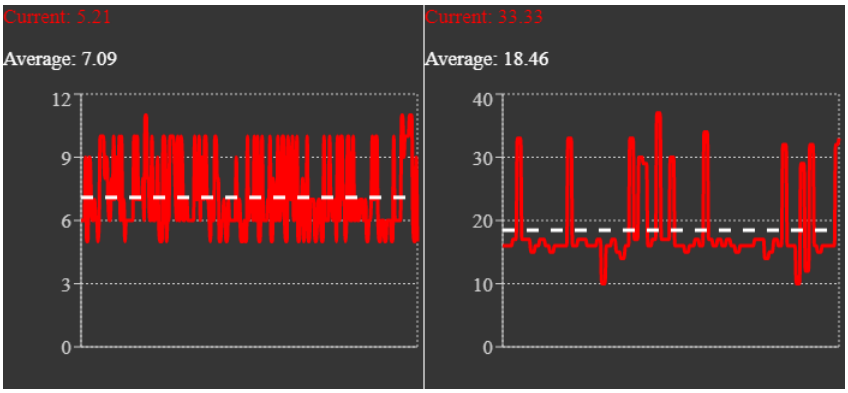

Figure 5.2: Performance comparison without and with a quad-tree.

[22] https://en.wikipedia.org/wiki/Pythagorean_theorem
[23] Quad-tree implementation repository: https://github.com/d3/d3-quadtree

5.4 Simulation visualization

As the *simulation visualization* is one of the main goals of the application, I present figures with the visualization results in this section.

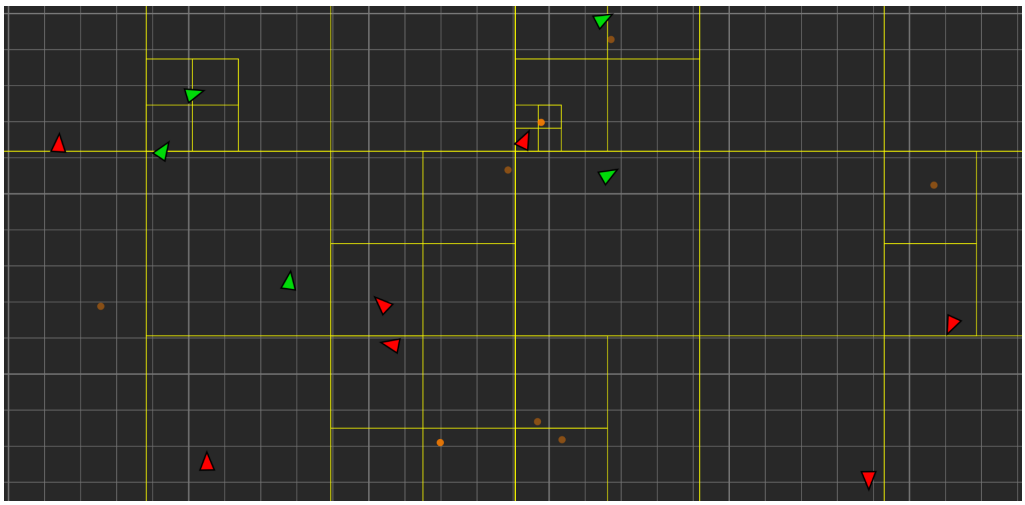

Figure 5.3: Example of a simulation visualization.

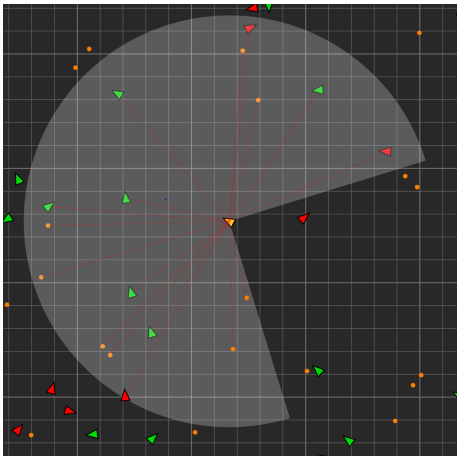

Figure 5.4: Visualization of a selected entity's perception.

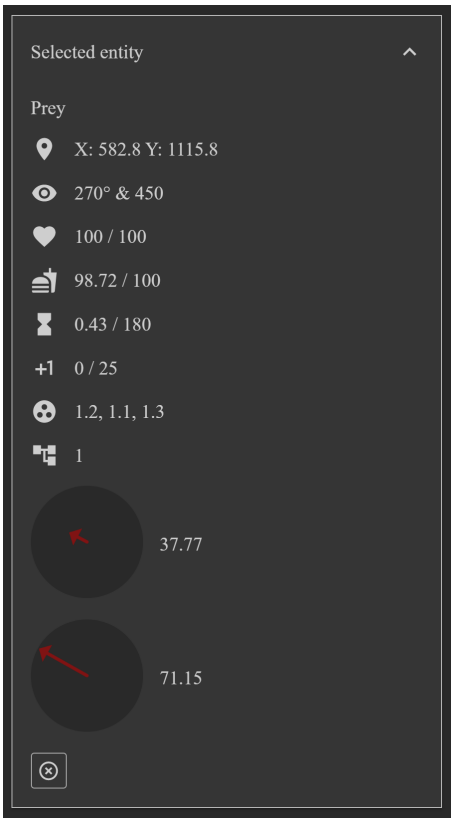

Figure 5.5: Part of the user interface showing various properties of a selected entity.

Visualization of the *entities* is visible in figure 5.3. The background of the figure represents a part of the entity *environment*. Next, a *regular grid* is drawn to provide a sense of scale. *Note*: The grid cells are purely visual and are *not* part of the model. Prey food entities are drawn as *orange circles*. The drawing *opacity* represents a visual indicator of the food entity *age*. As they age, their opacity *decreases*. Finally, the predators and preys are drawn as *red* and *green triangles*, respectively. An optional visualization of the *quad-tree quadrants* is also present.

It is also possible to *select* a predator or prey (food entities are intentionally not selectable) by clicking on them. Figure 5.4 depicts a highlighted entity shown in *yellow* together with its *perception area*. *Various properties* of a selected entity are part of the user interface (visible in figure 5.5). Properties include position, health, hunger, velocity, acceleration, and others. Hovering over any of the properties brings up a tooltip with their name.

Chapter 6

Results of simulations and suggested improvements

The implemented application allows the loading of multiple setting templates. Every template can be further customized. In the following chapter, a setting template is referred to as a *scenario*. I *simulated* each of the pre-loaded scenarios, and in this chapter, I present their *results*. The results are expressed with *charts* directly extracted from the application and brief *text descriptions*. Multiple figures tied to a specific scenario are always from the same simulation. In the end, I provide a *summary* of all the results, and a list of *suggested improvements* with brief descriptions.

6.1 Scenario 1 - The need for predation

The following scenario presents the importance of *predation*. In this scenario, there are no predators, and the preys have seemingly infinite food. The infinite food is achieved with *0% hunger delta* and not with abundant food entities.

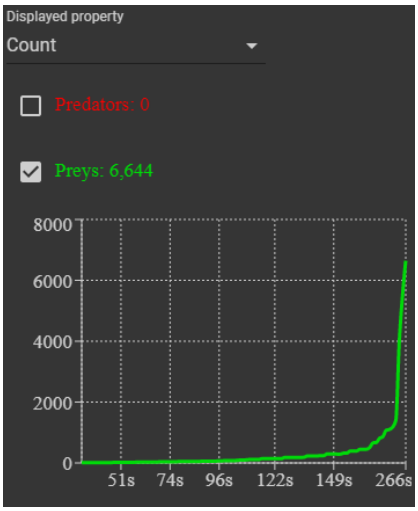

Figure 6.1: Exponential prey population growth caused by infinite food resources and the lack of predators.

The initial number of preys is *10*. After only *260* seconds, the number of preys rose to nearly *7000* (visible in figure 6.1). A small, temporary decrease in the population may occur due to the age model. With this number of entities, the simulation gets very slow, and it is not reasonable to continue. While this scenario is not very realistic, it quickly presents the *need for predators* in an environment. In a real situation, food can never be unlimited, but it can be so abundant, we can consider it as such. A similar *problem*[1] can be found in Australia caused by rabbits and lack of their natural predators. Assuming unlimited food resources, exponential prey population growth is also one of the predictions/implications of the *Lotka-Volterra model* (see 2.4.1).

6.2 Scenario 2 - Controlling prey population with food

In my model, the predator-prey, or more generally, the *producer-consumer* relationship is present *twice*. Predators consume preys, and preys consume prey food. The following scenario demonstrates the latter.

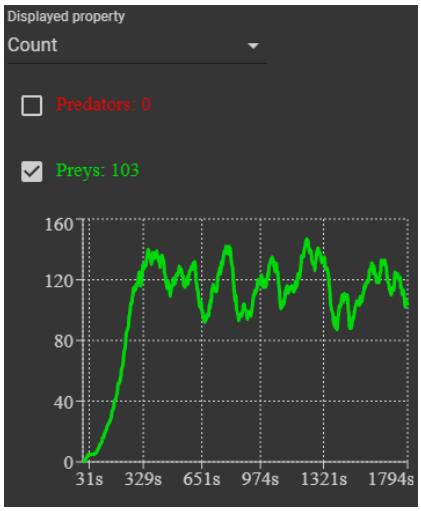

Figure 6.2: Stable prey population caused by limited food resources only.

The result is presented in figure 6.2. Despite the complete lack of predators, the prey population stabilized after approximately *5* minutes. The stabilization is caused purely by the limited input of food entities. The population can be considered *stable* despite the slight oscillation. The number of preys oscillates between *~90* and *~150* caused by the age model and the random nature of the simulation. As mutation is disabled in this scenario, the population is expected remain in this *closed range* indefinitely.

6.3 Scenario 3 - Absence of flocking

In this scenario, both predators and preys are present. Flocking and mutation are entirely *disabled*. The initial number of predators and preys is *5* and *30*, respectively.

In the beginning, the prey population grows exponentially, and after about three minutes, it reaches an *all-time high* (presented in figure 6.3). At this point, the prey population

[1] https://www.nationalgeographic.org/article/how-european-rabbits-took-over-australia/

is controlled mostly by the limited food. Initially, the predators are struggling because they have a hard time finding the preys as they are very scattered. At this stage, the *environment regeneration* is beneficial, as the simulation's initial state is usually the most challenging to get right. After the preys reached a decent population, predators can survive and reproduce. In this specific case, a visible predator population *promotion*, and simultaneously prey population *reduction*, happened after about *5* minutes. The cyclical population *equilibria* are visible in figure 6.4. Even without flocking, the populations can *stabilize*. In this scenario, the *steady-state* is to have approximately *2* to *3* times more preys than predators.

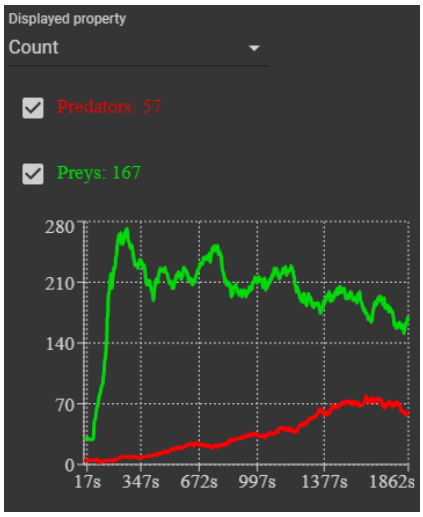

Figure 6.3: Populations without flocking at the beginning.

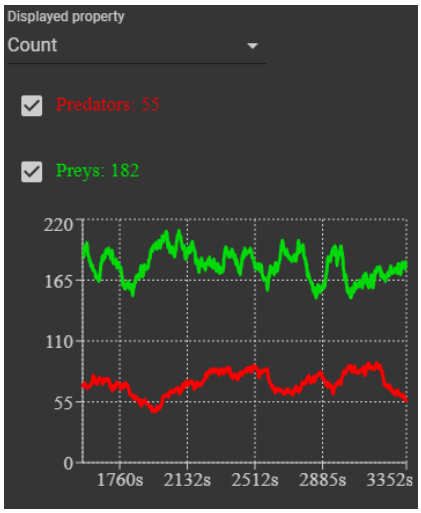

Figure 6.4: Cyclical relationship without flocking.

6.4 Scenario 4 - Presence of flocking

The following scenario is nearly identical in terms of its parameters to *scenario 3* (see 6.3), and the description of this scenario often references it for comparison. The critical difference is the *presence of flocking* and its impact on the populations. Each of the flocking behavior modifiers is set to *1*.

The effect of flocking on the populations is somewhat *unexpected*. The first difference is that the population did *not* reach an all-time high initially, but after the predator population promotion (visible in figure 6.5). Major predator population *promotion* happens slightly *later*, as preys are less scattered around the environment, and perceiving them is less likely. Another effect of flocking is that the cyclical population equilibria are more *pronounced* (visible in figure 6.6). At one point, the predator population even managed to *exceed* the prey population. Perhaps to most interesting effect can be found in the specific numbers of entities. With flocking enabled, the number of individuals in both respective populations is *slightly lower*. At first sight, the flocking seems like bad or undesired behavior. However, the improvement can be found in the *ratio*. Without flocking, the steady-state is to have about *2* to *3* times more preys than predators. In this case, it can be up to *4* times. This improvement is likely the result of decreased odds of predators finding prey. In other words, the gap between the red and green lines is bigger in figure 6.6 than in figure 6.4.

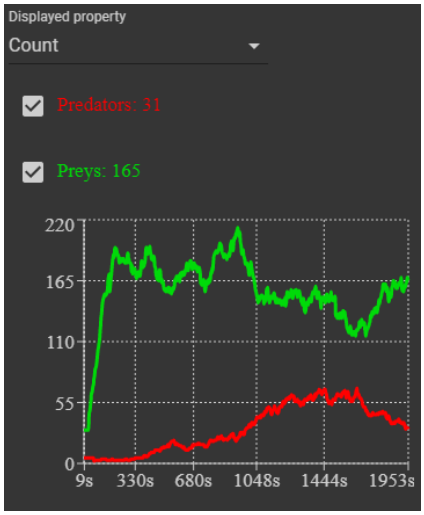

Figure 6.5: Populations with flocking at the beginning.

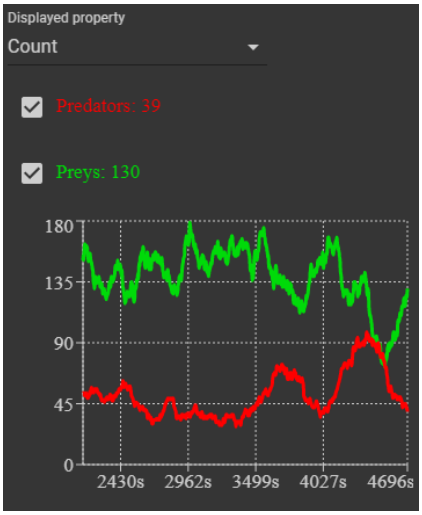

Figure 6.6: Cyclical relationship with flocking.

6.5 Scenario 5 - Mutating towards ideal properties

The final scenario utilizes all the available abstractions present in the model. The most crucial parameter in this scenario is the *5% mutation* for both predators and preys.

In this scenario, the population numbers behaved the most *chaotically* (observable in figure 6.7). As predators and preys rapidly evolve or mutate *together*, the only steady property in this scenario seems to be *chaos* and *change*. An indirect scoring system of all properties in my model is the natural selection. Figures 6.8 and 6.9 present a dramatic *drop* in average *hunger delta* and average *separation modifier* prey properties. These *jumps* happen when the prey population also significantly *decreases*. Therefore, the survivors shared a common property that natural selection deemed better and more important for survival. I present the average separation modifier specifically, as the other flocking modifiers did *not* change significantly and remained close to *1* throughout the simulation. Separation is the force that prevents very dense groups, but it seems that *dense grouping* results in better survival chances. Next, figures 6.10 and 6.11 display the *mutation of speed* and *maximum age* over time, respectively. The cyclical relationship is present in the mutation of both properties, likely caused by the *simultaneous* mutation of both types. Predators are evolving the ability to hunt better, while preys are evolving to escape better. While the speed tends to increase for both types, the maximum age seems to go down. Longevity seems to be *unimportant*, likely caused by *delayed reproduction*. Finally, figure 6.12 presents the mutation of *perception radius*. While the perception radius values are *always* different, the difference itself is almost *constant*. In other words, the two curves have very similar shapes.

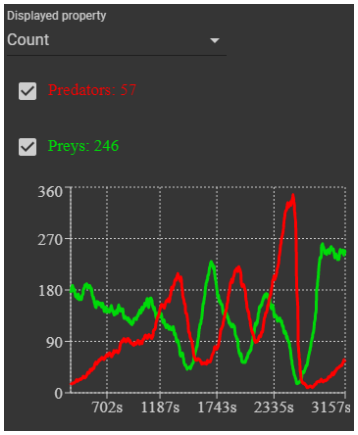

Figure 6.7: Population relationship with 5% mutation.

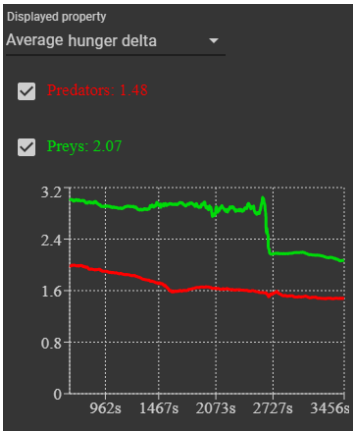

Figure 6.8: Mutation of average hunger delta over time.

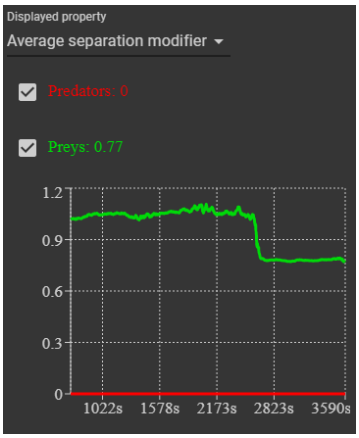

Figure 6.9: Mutation of average separation modifier over time.

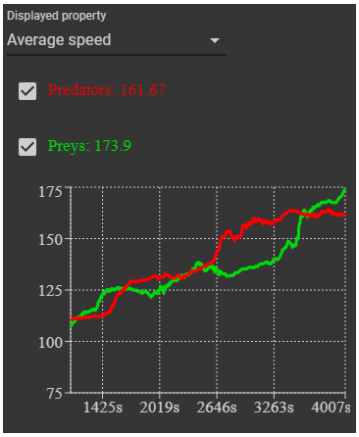

Figure 6.10: Mutation of average speed over time.

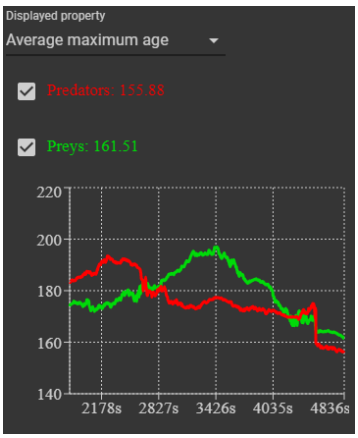

Figure 6.11: Mutation of maximum age over time.

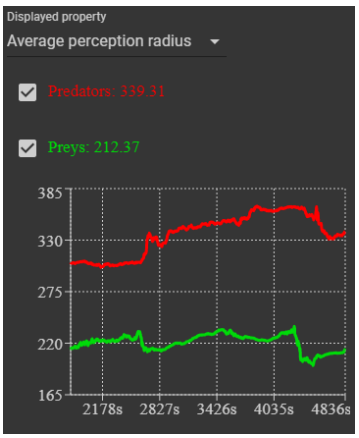

Figure 6.12: Mutation of perception radius over time.

6.6 Summary of results

To summarize, in this chapter, I presented the results of different scenarios. Due to the random nature of the simulation, obtaining similar results may *not* succeed on the first try. I added the most relevant and interesting charts extracted directly from the application to each scenario. However, the application gathers much more statistical information allowing *additional studies*.

In the design phase, I defined the problems that the model attempts to solve (see 4.1). Studying any specific property is not part of the assignment, and therefore, it is neither the focus of this thesis. I defined the problems to give the model a *purpose* and make the results of simulations *sensible* and *meaningful*. As the *boids model* is one of the focus points of this thesis, I simulated scenarios in which the flocking is *absent* (see 6.3) and *present* (see 6.4), respectively. In short, the flocking behavior did *improve* the prey population, but unexpectedly. The improvement is visible in the *ratio* of predators and preys. Without flocking, the steady-state is about *1:3* ratio of predators and preys. With flocking, the ratio grows to roughly *1:4*.

6.7 Suggested improvements and future work

In this section, I propose a set of *improvements* to various aspects of the currently implemented application with varying degrees of complexity.

- *Modeling of obstacles.* Obstacles could provide a more *diversified environment*, resulting in even more complex behavior. Thanks to obstacles, the preys might sometimes be forced to split up the flock to prevent a collision. Obstacle avoidance could also enable steering away from the simulated area boundary, while in the current implementation, entities that collide with the area boundary are wrapped around. The *challenge* with obstacles is that they should be considered in the *perception model* and the *collision model*. *Improvements* to the *locomotion model* would also be necessary.

- *Improved reproduction.* The current reproduction model is based on a maximum reproduction interval, the passing of time, amplification based on hunger status, and strictly defined age requirements. A possible improvement is to model *genders* and *mating*. Another possible improvement to reproduction could be the modeling of *pregnancy* and *congenital disabilities* making pregnant entities possibly more *vulnerable*, and congenital disabilities could introduce *high priority* targets for predators. Another improvement to reproduction could be *varying* the number of newly born entities. In the current implementation, always *one* entity is added when reproducing.

- *Mass modeling.* The implemented physics model assumes that all entities have *equal mass* regardless of other traits. Entity mass could partially be based on the *hunger model*, making overeating entities slower and a *higher priority* target.

- *Adjusting an ongoing simulation.* Currently, after a simulation is started, it is impossible to influence the entities or the environment in any way. In this improvement, I propose the ability to *edit* existing entity properties, *adding* or *removing* entities, *changing* environment properties such as prey food spawning rate or prey food expected maximum age. Such an adjustment could make the simulation *more interactive* and allow the creation of *unexpected events*.

- *Saving custom template settings.* It is currently possible to load pre-defined simulation scenarios and edit them, but it is impossible to save them as custom user template settings. I propose the ability to *save custom template settings*, possibly tied to a user account, which I propose next.

- *Server-side services.* The currently implemented server has a single role: to serve the client-side of the application. There is great potential for creating server-side services. As an example, I propose the addition of *user accounts* that could store certain information about the users. The stored information could mean storing *custom user templates*, *simulation states*, *user preferences*, or *statistics*.

- *Improving the user experience.* To make the application *more flexible* and *user friendly*, I propose adding the ability to change various settings currently *hard-coded*. The changeable settings could include: *hotkey mapping*, *area grid size*, *grid highlight interval*, *toggling the drawing of the grid*, or *camera movement speed*.

- *Adding the third spatial dimension.* My implementation models and simulates the virtual environment in two dimensions. Therefore, I propose an improvement to transform the model into *3D*. While some implemented behaviors could easily work in *3D* with minimal or no changes, others would require a complete re-design. The *PVector* class in the *P5.js* library supports three dimensions out of the box. Therefore the geometric physics model could easily be transformed into *3D*. One of the most challenging aspects of this transformation would be the boid *perception model*, which would require perception inside a *cone*. Another aspect of this improvement would be the *visualization* of the *3D* world.

- *Modeling eyes.* Improving the perception model by considering the entity *eye position*. In nature, predators often have eyes facing forward, while preys have their eyes positioned on the side of their heads to aid predator spotting.

- *Improved statistics presentation and saving.* Currently, it is possible to choose a single property that is drawn on the chart. Therefore, I propose the ability to choose *multiple properties*. For helping *clarity*, different properties would require different colors or line types. Also, it is currently possible to save a given chart as an image. I propose the ability to save the chart as *raw data* (using *JSON*[2] or *CSV*[3], for example). This improvement could aid in comparing multiple charts, possibly using other software.

- *Additional simulation parameters.* Increasing the number of parameters could enable simulating more varied situations. Examples of such parameters include *enabling/disabling* food sharing among preys, *changing the ratio* of shared food among preys, setting a *maximum speed*, fine-tuning *mutation*, defining the *collision parameters*, or setting the *age requirements* for reproduction.

[2] https://en.wikipedia.org/wiki/JSON
[3] https://en.wikipedia.org/wiki/Comma-separated_values

Chapter 7

Conclusions

The first goal of this thesis is the *analysis* of predator-prey systems. I discussed the most significant predator-prey model in detail, the *Lotka-Volterra mathematical model*. I also provided a more advanced alternative, the *Holling-Tanner model* containing more parameters. The next important goal is the discussion of implementation methods. Therefore, I first discussed the generic *process of modeling*, using a set of steps. Next, I discussed three specific approaches to modeling with their respective properties and possible use-cases: *mathematical modeling*, *cellular-automaton based modeling*, and *agent-based modeling*. The mentioned modeling steps can be applied to any modeling approach, as the approach itself refers to entity, state, behavior, or connection representation. Any of the approaches can be used to model a predator-prey system, but in this case, I utilized the *agent-based* approach. The main reason is the next focus point of the thesis, *C. Reynolds's boids flocking model*. I discussed the *perception model* and *behavior rules* defined in the boids model in detail. In addition, I discussed a *physics and locomotion model*, also created by C. Reynolds. While this was not an explicit requirement, I included it, as the boids model requires the ability of geometric flight and goal achieving.

Next, I designed and implemented a browser-based application. Thanks to *Electron.js*, the application can also be executed in desktop environments. For the implementation, I utilized modern open-source technologies, such as *React.js*, or *JavaScript* extended with *TypeScript*. The resulting application is capable of *controlling* and *visualizing* a predator-prey model simulation, allows setting simulation *parameters*, contains pre-defined *setting templates*, and gathers *statistical information*. Meaningful results and conclusions can be extracted from the application within *minutes* of simulation run-time. The implemented model contains three generic entities: *predators*, *preys*, and *prey food*. Therefore, the *producer-consumer* relationship is present *twice*. Predators and preys have the ability to *mutate*, while preys have an additional ability to *flock* based on the boids model.

I simulated each of the pre-defined setting templates and evaluated their results. The important conclusion from the simulations is that flocking did *improve* the prey population but in an indirect, unexpected manner. The improvement is present in the population *ratio*. While flocking is enabled, the ratio of predators and preys rises from *1:3* to *1:4*. Another conclusion is that entity mutation causes constant *chaos* and that the *cyclical relationship* is observable in specific entity properties as well. While the application can provide sensible results, it may be seen as a *proof of concept* and has great potential for future work. Therefore to finalize the thesis, I proposed a set of *improvements*.

Bibliography

[1] Arroyo, Y. *Docs need one or more diagrams #653* [online], 25. february 2018 [cit. 2020-06-09]. Available at: https://www.github.com/reduxjs/redux/issues/653.

[2] Beals, M., Gross, L. and Harrell, S. *PREDATOR-PREY DYNAMICS: LOTKA-VOLTERRA* [online]. 1999 [cit. 2020-05-05]. Available at: http://www.tiem.utk.edu/~gross/bioed/bealsmodules/predator-prey.html.

[3] Hassani, B. and Tavakkoli, M. A multi-objective structural optimization using optimality criteria and cellular automata. *ASIAN JOURNAL OF CIVIL ENGINEERING (BUILDING AND HOUSING)* [online]. january 2007, vol. 8, p. 77–88, [cit. 2020-06-05]. Available at: https://www.researchgate.net/publication/228522976_A_multi-objective_structural_optimization_using_optimality_criteria_and_cellular_automata.

[4] Jackson, J., Lewis, K., Norton, M. et al. Agent-Based Modeling: A Guide for Social Psychologists. *Social Psychological and Personality Science* [online]. december 2016, vol. 8, [cit. 2020-06-05]. DOI: 10.1177/1948550617691100. Available at: https://www.researchgate.net/publication/311425820_Agent-Based_Modeling_A_Guide_for_Social_Psychologists.

[5] Kübler, R. README.md. *Unity3D Flocking using Craig Reynolds' Boids* [online], 19. september 2018 [cit. 2020-02-19]. Available at: https://www.github.com/RafaelKuebler/Flocking/blob/master/README.md.

[6] Macal, C. and North, M. Agent-based modeling and simulation. In: [online]. December 2009 [cit. 2020-06-07]. DOI: 10.1109/WSC.2009.5429318. Available at: https://www.researchgate.net/publication/216813135_Agent-based_modeling_and_simulation.

[7] MacPhee, L. *Predator-Prey Interaction* [online]. [cit. 2020-05-05]. Available at: http://www2.nau.edu/lrm22/lessons/predator_prey/predator_prey.html.

[8] Marolt, D., Scheible, J., Jerke, G. et al. SWARM: A Self-Organization Approach for Layout Automation in Analog IC Design. *International Journal of Electronics and Electrical Engineering* [online]. january 2016, p. 374–385, [cit. 2020-05-05]. DOI: 10.18178/ijeee.4.5.374-385. Available at: https://www.researchgate.net/publication/320019435_SWARM_A_Self-Organization_Approach_for_Layout_Automation_in_Analog_IC_Design.

[9] Merritt, J. *Cellular automata as emergent systems and models of physical behavior* [online]. December 2012 [cit. 2020-06-07]. Available at: https://guava.physics.uiuc.edu/~nigel/courses/569/Essays_Fall2012/Files/merritt.pdf.

[10] Montgomery, J. *Predator-Prey Models* [online]. 1180 Observatory Drive, Madison, WI 53706: Department of Sociology, University of Wisconsin - Madison, 2009-10-22 [cit. 2020-06-06]. Available at: https://www.ssc.wisc.edu/~jmontgom/predatorprey.pdf.

[11] Mosimtec blog. Notable Agent Based Modeling Examples. *4 Agent Based Modeling Examples* [online], 10. may 2019 [cit. 2020-06-05]. Available at: https://www.mosimtec.com/agent-based-modeling-examples/.

[12] Obaid, T. The Predator-Prey Model Simulation. *Basrah Journal of Science* [online]. january 2013, vol. 31, p. 103–109, [cit. 2020-02-18]. Available at: https://www.researchgate.net/publication/308633701_The_Predator-Prey_Model_Simulation.

[13] Pelánek, R. Buněčné automaty a modelování založené na agentech. In: *Modelování a simulace komplexních systémů* [online]. Nakladatelství Masarykovy univerzity, 2011, chap. 2, p. 71–82 [cit. 2020-05-04]. ISBN 978-80-210-5318-2. Available at: http://www.radekpelanek.cz/dokumenty/ms-web.pdf.

[14] Pelánek, R. Matematické modelování a systémová dynamika. In: *Modelování a simulace komplexních systémů* [online]. Nakladatelství Masarykovy univerzity, 2011, chap. 2, p. 59–70 [cit. 2020-05-04]. ISBN 978-80-210-5318-2. Available at: http://www.radekpelanek.cz/dokumenty/ms-web.pdf.

[15] Pelánek, R. Modelování a simulace. In: *Modelování a simulace komplexních systémů* [online]. Nakladatelství Masarykovy univerzity, 2011, chap. 1, p. 43–55 [cit. 2020-05-04]. ISBN 978-80-210-5318-2. Available at: http://www.radekpelanek.cz/dokumenty/ms-web.pdf.

[16] Reynolds, C. W. *Boids Background and Update* [online]. 2001-09-06 [cit. 2020-05-10]. Available at: http://www.red3d.com/cwr/boids/.

[17] Reynolds, C. W. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics* [online]. july 1987, vol. 4, no. 21, p. 25–34, [cit. 2020-05-10]. Available at: http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/.

[18] Reynolds, C. W. Steering Behaviors For Autonomous Characters. In: *Proceedings of the Game Developers Conference 1999* [online]. San Jose, California: [b.n.], 1999, p. 763–782 [cit. 2020-05-15]. Available at: https://www.red3d.com/cwr/steer/gdc99/.

[19] Shiffman, D. Autonomous Agents. In: *The Nature of Code* [online]. 2012, chap. 6 [cit. 2020-05-20]. ISBN 978-0985930806. Available at: https://www.natureofcode.com/book/.

[20] Wolfram, S. Cellular Automata. *Los Alamos Science* [online]. Fall 1983, vol. 9, p. 2–21, [cit. 2020-05-04]. Available at: https://www.stephenwolfram.com/publications/cellular-automata-complexity/pdfs/cellular-automata.pdf.

[21] Yang, J., Tang, G. and Tang, S. Holling-Tanner Predator-Prey Model with State-Dependent Feedback Control. *Discrete Dynamics in Nature and Society* [online]. october 2018, vol. 2018, p. 1–18, [cit. 2020-05-10]. DOI: 10.1155/2018/3467405. Available at: https://www.researchgate.net/publication/328376981_Holling-Tanner_Predator-Prey_Model_with_State-Dependent_Feedback_Control.

[22] YUEN, A. and KAY, R. *Applications of Cellular Automata* [online]. 2010 [cit. 2020-05-05]. Available at: https://www.cs.bham.ac.uk/~rjh/courses/NatureInspiredDesign/2009-10/StudentWork/Group2/design-report.pdf.

Appendix A

Project overview and useful guides

In this appendix, I provide an overview of the *project structure*, a guide for *building* and *running* the application, and I discuss the application *compatibility* and *availability*. This appendix may be useful, especially for readers that wish to work on this or a similar application.

A.1 Project structure overview

The project is organized into *logical directories* (visible in figure A.1, rendered by the primary editor used during development - *Visual Studio Code*[1]). The whole project is in the *Predator-Prey-Simulation* directory. This directory is made up of two distinct directories and a special *.gitignore* file used by the version control system. The *local_packages* directory contains a single local, modified version of the *d3-quadtree library* (modification discussed in section 5.3).

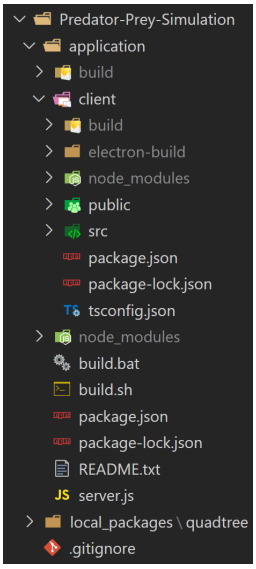

Figure A.1: Project file structure visualization.

[1] https://code.visualstudio.com

All of my own work can be found in the *application* directory. This directory contains a very simple *server*, *NPM* configuration files, *builder scripts*, a *README* text file containing information about building/running the application, and the *client* directory. The *client* directory again contains *NPM* configuration files, a *TypeScript* configuration file (not present in the application folder as the server is written in JavaScript). Finally, the client directory contains the *public* and *src* directories. The *public* directory contains static files, such as an *HTML* file, into which the *React.js* application is rendered. Some items in this directory were generated by the *create-react-app* (see 5.1.2) command line tool. All the application code resides in the *src* directory, which is further structured into various directories and modules. The *build*, *electron-build*, and *node_modules* directories are generated by various actions and may not always be present.

A.2 Building and running the application

The text of the following section is dedicated to *building* and *running* the application. The process of building is described as a *sequence of command-line commands*.

Prerequisites (other version are likely to work, but were not tested):

- *node.js* - v12.11.0
- *npm* - v6.11.3 (usually installed together with node.js)

All the following command line *commands* and *directory descriptions* assume the working directory to be the following directory (directory structure is discussed in section A.1):

Predator-Prey-Simulation/application.

Before the first build, *external package dependencies* need to be *installed* from local/remote locations with the following command:

npm run install:all

Running the *client development server*:

npm run client

When the application is under development, this server automatically *reloads* when changes are detected. With additional configuration, it is possible to reload specific parts only to retain the application state, further improving the development process. Such a technique is referred to as *hot reloading*.

Building the *client* only:

- cd client
- npm run build

Now, it is possible to run the client by opening the *index.html* inside the *client/build* directory with a browser. *Note*: A *React.js* application created by the *create-react-app* tool (see 5.1.2) does *not* support this by default. It is required to add the following *key-value* pair to the *package.json* configuration file, to make all the resource paths relative:

homepage: ./

Building the *client* and *desktop application* (using *Electron.js*):

- cd client
- npm run build:both

Note: For building the desktop application, it is necessary to build the standalone client, hence the „:both“ in the command. Now the client is available inside the *client/build* directory, same as in the previous guide, and the portable executable desktop application is inside *client/electron-build*. Electron.js might add other files in this directory but they are not needed to run the application.

Building the *server*, *client* and *desktop application*:

- OS *Windows* using a script:
  - .\build.bat
- *Linux* based OS using a script:
  - ./build.sh
- Generic steps:
  - run „npm run build:both“ inside the *client* directory
  - create a *build* directory
  - create a *build/client* directory
  - copy *server.js* to the *build* directory
  - copy contents of *client/build* to *build/client*
  - the built portable *desktop application* is inside *client/electron-build* and is ready to be used, moved or renamed

The *server.js* cannot be moved out of the development directory because its dependencies are located in the *node_modules* directory. The desktop application is also placed in the build directory for *convenience* but is not required by the server or client.

Running the *server*:

- OS *Windows*:
  - node build\server.js *PORT*
- *Linux* based OS:
  - node build/server.js *PORT*

where *PORT* is an optional parameter and defaults to *5000*. The application is now accessible at *http://localhost:PORT*.

A.3 Application compatibility and availability

Tested desktop browsers (on OS *Windows 10, x64*):

- Google Chrome v83.0
- Opera v68.0
- Mozilla Firefox v76.0.1

In general, any reasonably modern browser should work, which supports newer *CSS* properties, such as the *CSS Grid*. Outdated browsers such as *Internet Explorer* are not supported. *Mobile browsers* were not tested as the application is not suited for mobile devices. Due to the nature of the application, it should work on various operating systems, but the proper building and running are guaranteed on OS *Windows 10, x64*. Other operating systems may require *additional software* or *configuration*.

At the time of writing, the application is available here:

http://www.stud.fit.vutbr.cz/~xmeryj00/

Appendix B

Contents of attached DVD

- Archive *Jozef-Mery-Predator-Prey-Simulation.zip* containing:
  - *Predator-Prey-Simulation* directory - scripts, project configuration and source files, and others
  - *build* directory - built desktop application (executable on OS *Windows 10, x64*), and standalone client
  - *thesis-latex-src* directory - thesis source files (extracted from Overleaf[1])
  - *predator-prey-simulation.pdf* - thesis in PDF format

[1] https://www.overleaf.com