bsim: Agent Based Modelling Software for Shared Memory Systems to Simulate Bats

Wissenschaftliche Arbeit zur Erlangung des Grades
Master of Science (M.Sc.) in Computational Science and Engineering
an der Max Planck Computing and Data Facility (MPCDF) Garching

Geprüft von
Direktor MPCDF
Professor Erwin Laure

Technische Entwicklung betreut von
Stellvertretender Direktor MPCDF
Dr. Markus Rampp

Modellentwicklung betreut von
Gruppenleiter Abteilung für Tierwanderungen
Max-Planck-Institut für Verhaltensbiologie Konstanz
Dr. Kamran Safi

Eingereicht von
Student der Technischen Universität München (TUM)
Mendelssohnstraße 20
81245 München
B.Sc. Andreas Reiner Laible

Eingereicht
München, 1. Juni 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Verwendung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, wurden als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

München, 1. Juni 2024; Andreas Reiner Laible

D

Abstract

While searching for food, bats form large collectives and exchange information about food sources by echolocation calls. A software tool was developed to experimentally investigate the influence of bat's information exchange on the foraging efficiency of the group. The tool implements an agent-based model of bat foraging behavior, where agents can exchange information about food sources. Written in pure C++, it is fast, works on normal desktop computers and provides a graphical user interface for easy parameterization and visualization of the simulation. It is seperated into a simulation and an animation framework, which can be used independently. The user can implement different agent behavior algorithms with few lines of code and can easily extend the tool with new features. While providing every feature necessary for the experiment, the tool is also designed such that it can be also used for other experiments in the field of agent-based modeling. First experiments with the tool show that the information exchange between agents can significantly increase the foraging efficiency of the group. While existing agent-based modeling tools are mainly written in Java, the new tool written in C++ is a fast and easy-to-use alternative for researchers in the field of agent-based modeling, providing new possibilities for experiments in this field.

Contents

List of Figures

List of Tables

Abbreviations

ABM	Agent-Based Model
API	Application Programming Interface
CSE	Computional Science and Engineering
GUI	Graphical User Interface
GPU	Graphical Processing Unit
HPC	High Performance Computing
IPC	Instrucions Per Cycle
MPCDF	Max Planck Computing and Data Facility
NUMA	Non-Uniform Memory Access
OMT	Optimal Movement Theory
SoA	Structure of Arrays

Introduction

This is a master thesis in Computational Science and Engineering (CSE) at the Technical University of Munich. CSE is a program that combines computer science and applied mathematics to solve complex problems in science and engineering [1]. Computational methods become of interest when real-world problems are too complex to be solved analytically or experimentally. In this thesis, a tool was developed to investigate in the effects of social information on swarm behavior of bats. The tool is designed to fit the specific requirements of Dr. Hannah Williams and Dr. Kamran Safi from the Max Planck Institute for Animal Behavior in Konstanz, Germany. Under the supervision of Dr. Markus Rampp from the Max Planck Computing and Data Facility (MPCDF) in Garching, Germany, the tool was implemented as an agent-based model in C++. The main work of this thesis was to develop and implement the model, and to create a user-friendly interface for biologists to use the tool. While the tool itself is designed to be used by biologists, the written part is aimed at computer scientists and engineers. The written part describes the theoretical background, the implementation of the model and software engineering aspects. Computational aspects such as parallelization and optimization are discussed in the written part as well, but it is not intended to be a documentation for the use of the tool.

Motivation

> "Do bats benefit from the information aquired about the location of food sources from echolocation calls of other bats so much, that this is a sufficient reason to build swarms?"

This is the scientific question. Not a computational problem yet, but one that could be addressed with a computational model. Dr. Kamran Safi from the Max-Planck Institute of Animal Behavior in Konstanz raised this question. Together with Dr. Hannah Williams, he is studying the social information transfer in bats. They are part of the Department of Migration, which is headed by Prof. Dr. Martin Wikelski and trying to understand the movement of animals. Of course, the best way to measure the importance of social information transfer in bats would be to inhibit it in some way and measure the consequences. This is not ethically desirable, so we need to rely on computational models to address this question. As bats form large swarms and communicate a lot, this model is expected to be complex and computationally expensive. For this reason, we started with a simple model and gradually increased its complexity. Finally, we ended up with a model that is still simple too use, but comprehensive enough to address the question. Further, the tool we created to simulate the model is flexible enough to be used for other questions as well. For the student this was an exiting project, comprising the entire simulation pipeline from the initial question to the final model, from the model to the simulation tool, from plain data to visualization and from theory to practice.

Chapter 1

Modelling the problem

1.1 The scientific question

Let's have a look at the initial question (see motivation) in more detail. The hypothesis to test is, that bats benefit from the information aquired about the location of food sources from echolocation calls of other bats. The question is, if this benefit is so large, that it is a sufficient reason to build swarms. This is a reasonable question, because up yet, the reasons for bats to swarm are not fully understood [2].

Bats localize food sources by echolocation calls. These calls emitted by the bats are reflected by the objects in the environment. The bats listen to the returning echoes and can determine the distance and direction of the objects. By the type of other bats' calls, bats can also determine whether other bats are searching for food or have found a food source. This information can be used by the bats to find food sources more efficiently.

In reality it is not possible to prevent this exchange of information between bats for various reasons. If they were either muted or deafened, they would lose their sense of orientation and would not be able to find any food sources at all. Furthermore, this would be an unethical experiment, as it would harm the bats. Therefore, Dr. Williams and Dr. Safi want to use a simulation to research the effect of the information on the bats' foraging success.

1.2 The model

To test this hypothesis, an agent-based model was developed. In an agent-based model (ABM), the interactions of autonomous agents are simulated to understand the behavior of a system [3]. In this case, the agents are the individual members of the population, and the system is the population as a whole. The agents are programmed to follow a set of rules that determine their behavior. The model is then run for a number of iterations, and the behavior of the agents is observed. By varying the rules that govern the agents, the effect of different factors on the behavior of the population can be studied. The choice of an ABM was made because it allows for the simulation of complex systems with many interacting parts.

1.3 Agent-based modelling

Simply speaking, an ABM can be compared to a role-playing game, each agent is defined by a set of characteristics and rules that govern its behavior, and each agent is able to interact with other agents and make individual decisions based on its own characteristics and the characteristics of the other agents. The system evolves over time as the agents interact with each other and the environment. The state of the system can only iteratively be determined and there are no closed-form solutions. In ecology, agent based models are a common approach, characterized by discrete representation of individuals, local interactions and a representation of how individuals interact with each other and their environment. Ecologists also call them individual-based models (IBMs) [4]. Or in other words, ABMs model systems as an ensemble of loosely coupled, goal-oriented autonomous entities, competing or collaborating by exchanging messages [5]. The strength of agent-based models is based on their ability to model decicion making processes and social interactions [6]. This is very important in the context of the current work, as the model is based on the interactions between the agents and the environment. One of the challenges in modelling ABMs is to choose the right degree of parameterization, it is important to include enough parameters to represent the essential mechanisms of the system, while on the other hand each extra paramter complicates the model and makes it harder to interpret [4]. Also every parameter which has to be processed increases the computational effort, so it is important to find the right balance between complexity and simplicity. The number of parameters in the model thus is the minimum necessary to represent the essential mechanisms of the system. And according to the principle of parsimony, the simplest model has to be preferred over more complex models, as long as it is able to represent the essential mechanisms of the system.

As a matter of fact, translating a real-world system into an agent-based model can be summarized in the following steps:

1. Identify the minimal set of parameters to represent an agent.
2. Define a set of rules that govern the agent's behavior.
3. Develop an abstraction of perception containing all necessary information for the agent's decision-making process (an aggregation of state variables).
4. Elaborate an algorithm that allows the agents to act simultaneously and time-consistently.

The last point might sound trivial and obvious, but it is not. To ensure time-consistency and simultaneity, the agents must be able to act in parallel. Individual agent behaviour must not be influenced by the order in which agents are processed. Time-consistency is further important, as the model is designed such that information is exchanged between agents: social information must still be valid when the agents act, nor must the agents retrieve information from the future. So time-consistency means, that all agents jump from one state to the next at the same time, and that the state of the system is consistent at any given time. There is no textbook solution to this problem, as it depends on the specific model and the problem at hand. In the current model, this problem is solved by representing the agents as a part of

the system, representing the entire system as a consistent snapshot of the system at a given time, and updating the system state after all agents have acted. [1]

1.4 First approach: Prototype

At first, the problem was tackled in a straight-forward approach. A program was written in C++ that would read the input file, and then simulate the bats flying around searching food. Information exchange in between the bats could be switched on or off. A lot of effort was put in optimization for speed, and the program was able to simulate a large number of bats in a reasonable time. Talking back to Dr. Williams and Dr. Safi, it turned out that the performance of the prototype was sufficient for the current project. Thus it was not necessary to port the program to an HPC system, as we had initially thought. This had a major impact on the project, as the simulation could now be designed to run on a standard desktop computer.

Anyways, the first prototype had a major drawback: it was not flexible. The prototype was hard-coded for the specific simulation of bats searching for immobile food sources, and it was not possible to use it for any other kind of agent-based simulation. Due to the high degree of optimization for speed, it further was nearly impossible to change the program if new parameters were to be introduced or a different kind of agent behavior was to be simulated. Obviously, there seems to be some truth in the saying that premature optimization is the root of all evil. Thus, the decision was made to rewrite major parts of the program with optimized architecture and design, to not limit its lifespan to the current project. Instead of a specific hard-coded simulation, the new program should be a general simulation framework that could be used for a wide range of agent-based simulations. Instead of pure optimization for speed, the new program should be optimized for flexibility and maintainability.

Of course a lot of the code could be reused, and even more important, the experience gained from the first prototype was invaluable. Lots of software design aspects and necessities can hardly be anticipated right from the start, often they only become apparent during the development process. Only after having written the first prototype, all requirements were clear and the new program could be designed accordingly. Also as a lot of time was spent on performance optimization of the first prototype, computational bottlenecks were already identified and could be avoided in the new program, and the fastest solutions for specific problems were already known. Thus development of the new program was much faster than the first prototype.

[1] As explained later (see sections 3.1.2 and 3.1.3), the Agent class objects in our computational model have no memory members, and the state of the system is represented by the State class object. So to say, the agents are not complete objects, they have a 'physical' representation in the State object but all their cognitive decisions are made by the Agent class objects. This principle is called the separation of concerns, and it ensures that the agents are not influenced by the order in which they are processed.

1.5 Second approach: Generalization

One of the major insights from the first approach was, that the question to be answered was a particular case of a general problem. To allow for more flexibility of the software, the problem was generalized and the software was designed to be able to solve a wide range of problems. This way, the software and all the effort put into it would not be wasted, after the initial problem was solved. The general problem is the following:

- individuals are represented by a set of attributes
- individuals are grouped into classes
- individuals interact with each other
- individuals of different classes interact with each other in a different way
- individuals move in 3D space, based on a set of rules

Thus not only bats are agents, but food sources are nothing else than agents with a different set of attributes. By parameterizing the agent behavior too, the software can handle different kinds of agents, with different behavior without changing the code. Thus the software can be used to simulate a wide range of problems, from the behavior of a flock of birds to the spread of a disease in a population. It is even possible to simulate an entire eco-system, with predators and prey, plants and herbivores, and so on. All that is needed is to define the attributes of the agents, the rules of interaction and the rules of movement. The software therefore is designed in a way that it can be easily extended, by adding new rules of interaction or movement, or by adding new attributes to the agents.

Chapter 2

Software Design

We created a tool to simulate thousands of bats in their 3D movement. The bats search for food and exchange energy with the food sources, such that their foraging success can be evaluated. Different search strategies can be implemented and compared, including Levy flights and Brownian motion. Information flow between the bats can be regulated, to test the effect of social information on foraging success.

This chapter describes the implementation of the tool at a high-level and explains the reasons behind the software design choices. Furthermore, it provides an overview of the implemented system and the technologies used. Details of the implementation are provided in chapter 3.

2.1 Existing software

Of course there are existing agent-based modeling frameworks, such as NetLogo [7], MASON [8], Repast [9], and GAMA [10]. But these frameworks are not designed for the specific requirements of the current model. They are designed for general purpose agent-based modeling, and the current model is a very specific application. Even if our own implementation became capable of doing general purpose agent-based modeling, the generalization introduced no significant overhead. Furthermore, the named ABM frameworks are all written in Java, which is not the language of choice for the current work as it is not as fast as C++.

Also, a common basic assumption of ABM is, that agents only have access to local information and can obtain global information only through communication with other agents [11]. The existing frameworks are designed to support this assumption, even if they allow the use of global information. Our agents, due to their bat-nature, communicate non-stop with up to all other agents in the model. They permanently emit their echo-location signals and listen to the echoes of all other agents. To implement this information exchange by communication would be very inefficient as it increases the complexity of the algorithm by a factor of O(N^2), where N is the number of agents (an all-to-all communication scheme). Thus we decided to implement our own high-performance agent-based modeling framework in C++, tailored to the specific requirements of the current model. Instead of communicating all infor-

mation between agents, we use a shared memory model, where all agents have access to the same memory. This allows for a very efficient implementation of the echo-location algorithm, as agents can read the information of other agents directly from memory. By replacing communication with shared memory, information is not limited by what is communicated, but by which information the agents are permitted to use by the model.

With an existing framework, information exchange would have to be implemented by communication or the use of shared memory handled by the framework. Our own implementation in C++ allows us very low-level control over the memory management and is more suitable for our specific requirements than the existing, mainly Java-based, frameworks.

2.2 Programming language

The entire program is written in C++. C++ is an object-oriented programming language that is an extension of the C programming language [12]. It is well-suited to realize the intended design aspects (see section 2.3, p. 9) of the program. Furthermore, C++ is a widely used language in the field of computer science and is supported by many libraries and frameworks. And most importantly, it is a compiled language which allows very fine control over what happens below the hood. This is important for the performance of the program, as it allows to use the full potential of the hardware. Also, as C++ allows very low-level programming, it does not restrict the programmer in any way nor bloat the program with unnecessary overhead. This is also very important for the performance of the program, because we want all resources to be used as efficiently as possible.

2.3 Design aspects

To meet the requirements derieved from the generalization (see section 1.5, p. 6), the following design decisions were made:

- all agents are defined by the same set of parameters
- an initial set of parameters applied to a set of agents is defined as a class
- from the state of the system, a set of meta-information is provided to the agents
- each agent can do an individual transition to the next state based on the meta-information

Furthermore, based on the experience gathered in developing the first prototype, the following design decisions were also made:

- apply the Unix philosophy[13]:
  - each function should do one thing and do it well
  - don't clutter the output with unnecessary information

  - make the output machine readable
  - do not insist on interactive input, the program should be able to run without human interaction
- use a modular design to allow for easy extension
- every class should have a clear responsibility
- allow for logging the entire system state, such that the system can be analyzed afterwards
- allow for easy debugging, by a verbose debug mode
- choose readability and maintainability over maximum performance
- apply the KISS principle[14]: keep it simple, stupid

The importance of these design aspects have become clear during the development of the first prototype, and are essential to the development of the second prototype.

2.4 Third-party libraries

It's a waste of time to reinvent the wheel, so third party libraries were used as much as possible. These include libraries for command line and configuration file parsing and several libraries for the graphical user interface. For the implementation of Levy flights, the inverse cumulative distribution function from the asa241 library was used. All libraries are open source and have a permissive license. The libraries were selected such that they are easy to use, have a good documentation, are actively maintained and do not pull in further dependencies. Here is the list:

- TOML file parser: TOML++ [15], MIT license
- Command line parser: cxxopts [16], MIT license
- Inverse cumulative distribution function: asa241 [17], GNU LGPL license
- OpenGL library: GLFW [18], zlib/libpng license
- OpenGL mathematics library: GLM [19], Happy Bunny License
- Graphical user interface: Dear ImGui [20], MIT license
- ImGui file Dialog: ImGuiFileDialog [21], MIT license
- Image codec: libwebp [22], BSD license

These are great libraries, they saved a lot of time in the development of the simulation part and without them the animation part would not have been possible to the same extent. Let the authors of these libraries be thanked for their work at this point.

2.5 AI assisted coding

To the time of coding, a new and highly fascinating technology has become publicly available: artificial intelligence (AI). AI is a technology that has been around for a long time, but only recently has it become so advanced that it can be used in everyday applications. The most well-known example of AI is probably the chatbot, which is a program that can simulate a conversation with a human. Less known, but more important for programmers, are AI coding assistants. Both technologies have been used in this project.

Chatbots Chatbots were used to ask questions like: "How to animate a 3D point cloud?". The chatbot would then provide an explanation or a code snippet. Code provided by the chatbot is still not perfect and often buggy, but it can be a good starting point, to learn how to solve a problem, find new libraries or functions, or to get a general idea of how to solve a problem. Unfortunately, chatbots tend to make up stuff if they don't know the answer, and being told that a solution does not work, they will often stubbornly provide the same solution again, or creatively make up another faulty solution that doesn't exist.

AI coding assistants AI coding assistants are more advanced version of chatbots. They integrate into the code editor and can provide suggestions for code completion, code refactoring, and code generation. The AI coding assistant used in this project was GitHub Copilot[23]. It was trained on all of GitHub's public code repositories, which is an incredible amount of code. In the preferences, it is possible to block suggestions matching public code, to prevent accidental plagiarism. For this project, of course suggestions matching public code were disabled.

Using AI reviewed The AI coding assistant was a great help in this project, mainly by speeding up the coding process, thus increasing productivity enormously. It has to be imagined like a very skilled assistent, who can write code for you, but only if you tell him exactly what to do. The thinking still has to be done by the programmer, but the typing is done by the AI assistant. One evil detail is, that the AI is trained on human code including human errors, especially such that are hard to spot at first glance. So one of the lessons learned is to always double-check the code generated by the AI assistant. And sometimes, the AI assistant is annoying, like any assistant, who always provides suggestions, even if you don't need them. After all, soon no programmer will be able to afford not using such powerful tools.

2.6 Build system

With the project and codebase evolving, so did the need to enhance the build system. The prototype used a starting simple - at last large - Makefile. This was replaced by a more sophisticated build system based on CMake [24]. This was necessary for there are more than 60 source files in the final codebase, not counting third-party libraries. The codebase is split into several libraries, each

with its own CMakeLists.txt file. By splitting the codebase into libraries, the build system can be more modular and flexible. Also, changes in one library do not require a full rebuild of the entire project. Another advantage is the simplification of the build configuration. The project can be built with different configurations, such as debug or release, and with different build options. Further, it is possible to build the project with the GUI or as a headless version. In the headless version, parallelization can be disabled, or the project can be run on a cluster. To test performance, the project can be built with PAPI support, an advanced low-level performance measuring library [25].

2.7 Program structure

The software, on a high-level, is structured in two main parts: the simulation part and the animation part. The simulation part realizes the agent-based model. The animation part visualizes the simulation and provides a graphical user interface(GUI) to interact with the simulation. Both parts are completely independent of each other. The simulation can be run without animation and GUI, and the animation part can be used to visualize any 2D or 3D point data. In chapter 3 the architecture of the software is described in detail.

2.8 Software distribution

The tool is distributed under a MIT license. It is called 'bsim' for bat simulation. You can find the source code on GitHub:
`https://github.com/de-arl/bsim`.

Chapter 3

Architecture and Functionality

This chapter describes the architecture, functionality and implementation details of the program. As the codebase is quite large, we will focus on the most important parts of the program. As mentioned in section 2.7, the tool can be split in two independent parts, these are the simulation and the animation part. While the simulation can be run headless, that means in the terminal without graphical user interface (GUI), the animation part provides a GUI and can be used to animate any 3D point cloud data and easily be adopted to a different simulation.

3.1 Simulation

The simulation part of the software implements the agent-based model. It comprises the necessary code to represent the agents and the interactions between them, as well as the code to run the simulation, and collect the data. The data can be logged to a file to be analyzed later. Additionally, it includes code to save and restore the state of the simulation at any point. Figure 3.1 shows the corresponding UML diagram. There are three major classes: Agent (see subsection 3.1.4), State (see subsection 3.1.2) and StateMachine (see subsection 3.1.3). These are responsible for the representation and evolution of the modelled system. The other classes can be summarized as parameter classes, helper classes, result aggregation classes and user interface classes. In the following sections, the classes which constitute the simulation part of the software are described in detail. How the classes interact is explained in the corresponding sections. The UML diagram is not further referenced, but recommended to be consulted for a better understanding of the software structure. All class-interactions mentioned in the following sections are also visualized in the diagram and all interactions in the diagram are explained in this chapter.

3.1.1 World

The world in which the agents act is a uniform continuous symmetric cuboid space around the origin. Space is limited, and agents cannot move outside of it. In the prototype periodic boundary conditions

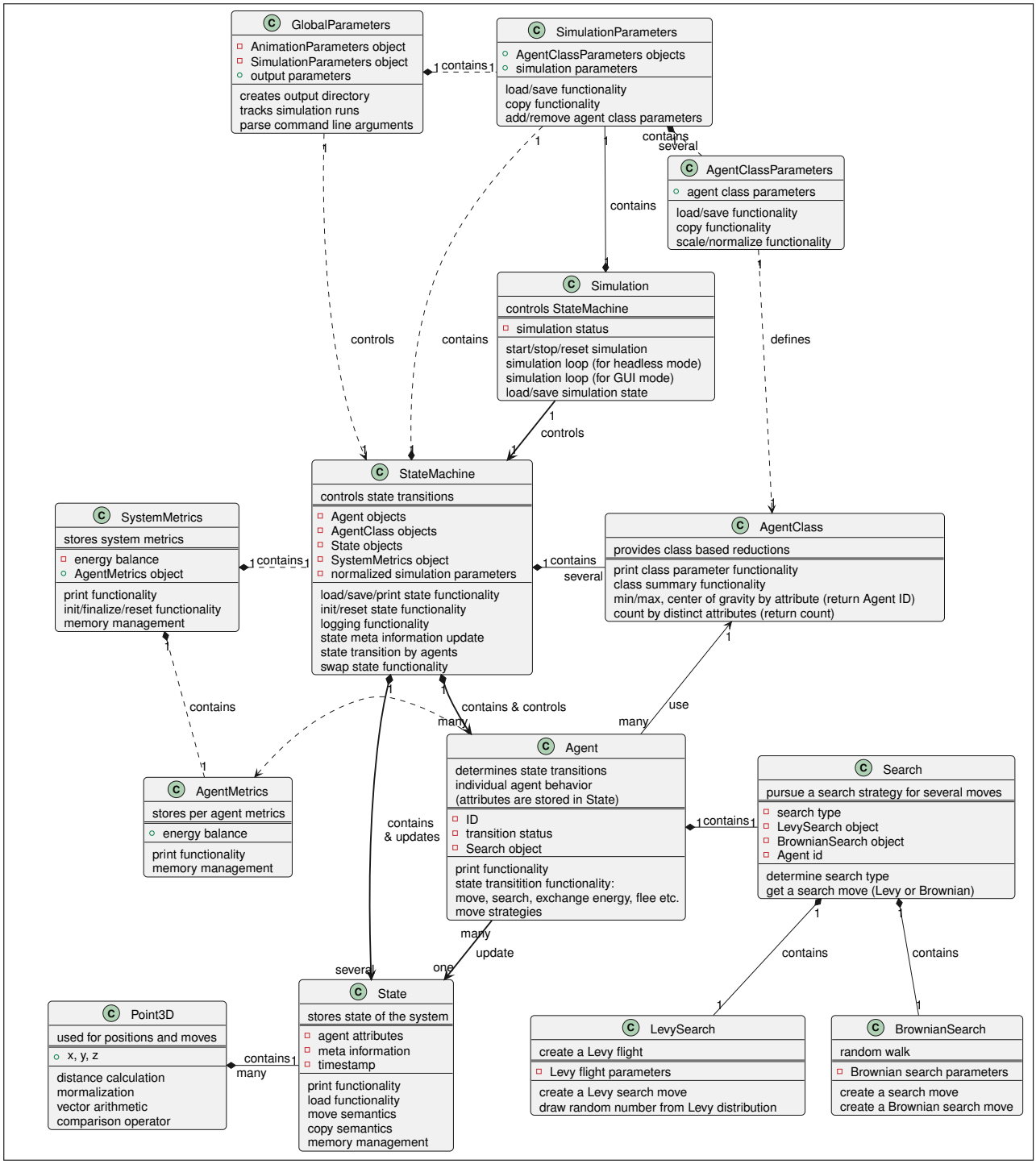

Figure 3.1: Simplified UML class diagram of the simulation

were tested, but they do unnecessarily complicate the model and were not included in the final version. According to Dr. Safi, they are not necessary for the experiments he wants to run either. As the real world habitat of bats is not infinite, the limited space is a more realistic representation of the environment. Thus the walls are simply repulsive, and agents cannot move through them. Also the user can choose worldsize so large that the agents will never reach the walls, so the walls will never have any effect on the agents. The world can be given any size, but internally all length and distance units are always normalized, such that the longest world side ranges from -1 to 1. The world is fully defined and represented by its dimensions in the simulation parameters.

3.1.2 System State

The state of the system is represented by the State class. Several objects of the State class exist, each representing a different state of the system. State objects never change, in each iteration of the simulation a new State object is created. The program can store an arbitrary number of State objects. These objects form the memory of the system and allow for complex agent behavior. Agents can access the memory and use it to make decisions.

For the program to work, at least two State objects must be stored. One stores the current state of the system, the other is created in each iteration and will become the new current state. In every iteration, the oldest State is deleted and the memory deallocated. The object which held the oldest State then receives the contents of the second oldest State. Swapping states is done using move semantics, so the memory is not copied, only the resource ownership is transferred. This way, there is no performance penalty for using multiple State objects and the only factor limiting the number of simultaneously stored States is the available memory. The State objects are governed by the StateMachine class.

3.1.3 State Machine

The entire simulation is based on the concept of a state machine. As one can see in the UML diagram (p. 13), the StateMachine class is the central class of the simulation. It is responsible for managing the state transitions and the state data. If a new simulation is started, the StateMachine initializes the first state based on SimulationParameters and AgentClassParameters objects. Responsible for the transitions of the states are the Agent class objects - the agents of our model. The StateMachine handles all the agents and passes the current state to the agents transition functions. Further it provides the agents with a new state object, into which they can write the new state data. After the StateMachine has called all the agents transition functions, it calculates the meta information of the new state. Meta data includes for example the distances between the agents or the number of agents of a class X within a certain distance to an agent Y. This meta information is needed for the agents to calculate their transitions. Further the StateMachine brands the states with a timestamp in simulated time. The meta information is also stored in the state object. After the next State of the system is calculated, the StateMachine swaps all the states, forgetting the oldest and creating a new empty state. As the State objects define the state of the system entirely, store and load functions of the StateMachine allow for

stopping and resuming the simulation at any time. The StateMachine is controlled by the Simulation class.

3.1.4 Agents and Agent Classes

Together with the State class and the StateMachine class, the Agent class is one of the three main classes of the model. In the UML diagram, this relationship is emphasized by the bold arrows between these classes. The representation of the agents of our model is split into two parts. Imagine body and soul. The body is the part of the system which interacts with the other agents. It is the representation of the agent in the State object, which for every agent contains the set of variables that fully define the state of the agent. The soul is the part of the agent, which makes decisions. It is represented by the Agent object. The agent objects have only two memory members: the agent's id and the agents transition state. The transition state is the state the agent is in between two time steps. For example, if it dies by one transition function, a second will know from the transition state that it is dead. The agents have a seperate transition function for every state variable of their representation in the State object. Each transition function is responsible for a single state variable only. The transition functions are passed the old state of the agent as a constant reference and the new state as a reference. This means, they can never change the current state, such that it is guaranteed for every agent that the state is consistent during the whole time step. Passing the states per reference further avoids copy operations and is important for the performance of the model. The agents provide a lot of convenience functions to access their state variables and the meta information related to them, and to interact with other agents. These convenience functions have been written with the user in mind. They are thought to be easy to use and to understand. Using the convenience functions, the user can easily implement the desired behavior of the agents.

AgentClass The sum of agents of a common class form an AgentClass object. The AgentClass object provides the user with the possibility to access all agents of a class at once. Though an AgentClass object could be used to force all agents of a class to have the same behavior, this is not the intended use. AgentClass objects are meant to provide class reduction operations. They are aware of the state variables of all agents in the class, similar to a shared consciousness. Such reductions include for example summing up the number of agents in the class with a common attribute or finding the agent with the highest value of a state variable. This information can be used to implement the decision algorithms of the agents. The reduction operations provided do not operate on a single state variable, but they combine a reduced variable with one or two masks, such that complex behavior can be implemented easily. This way, for example requests like "find the closest of class A with property A and B" are easy to implement. Last but not least, one of the most important aspects of splitting the agents into body and soul is, that the user can try all kinds of different transition functions without changing the representation of the agent or having to worry about the system as a whole to malfunction. The StateMachine does not care about the transition functions, it only cares about the State objects. Thus the agents are modelled

in a way that the user can easily experiment with different transition functions and find the one that fits best to the desired behavior of the agents.

3.1.5 Search

As shown in the UML diagram (p. 13), the Agent class objects have a Search class object as member. This Search class object enables the Agents to pursue a search strategy over unlimited time. That means it can be imagined as part of the agents' memory.

In our model, a key factor is the search strategy of the agents. The agents search for targets conducting a so-called random walk. Random walks are a mathematical model for a path that consists of a succession of random steps. They can be classified into two categories: Brownian motion and Levy flight [26]. The two subclasses of the Search class, BrownianMotion and LevyFlight, implement these two types of random walks.

Brownian motion is a random walk in which the individual steps are independent and identically distributed with mean and variance. Levy flight is a random walk in which the individual steps are independent and identically distributed, but the variance is infinite. This means that the steps are not normally distributed, but follow a heavy-tailed distribution, i.e. most of the steps are small, but some are very large [26].

The Search class coordinates the search strategy of the agents. Because Levy steps are often very long, agents cannot do Levy steps in single time steps. Therefore, the agents do a Levy step over multiple time steps. Because of this, the agents need the special search memory.

Agents alternate between the two types of random walks. They do a Levy step, then Brownian steps for a given time, then another Levy step, and so on. This is implemented by the Search class. It enables the agent to pursue a Levy-step over several time-steps and knows the remaining time for the Brownian steps. All of this is handled under the hood, the user only has to set the parameters for the search strategy and call the getDirection function.

Figure 3.2 on page 17 shows the difference between Brownian motion and Levy flight and the mixed search strategy. One can see that the area covered by the agent strongly depends on the search strategy. Brownian motion covers a small area, while Levy flight covers a much larger area.

3.1.6 Controlling the Simulation

Simulation Class

The Simulation class controls the StateMachine. It provides an endless loop intended for use with the graphical interface and a loop that runs for a specified number of iterations intended for use with the command line interface. If the simulation program is to be included in a larger program, the Simulation class could be used as a simple interface to the StateMachine. A simulation object can initialize the StateMachine with a set of SimulationParameters, start the simulation, pause the simulation, reset the

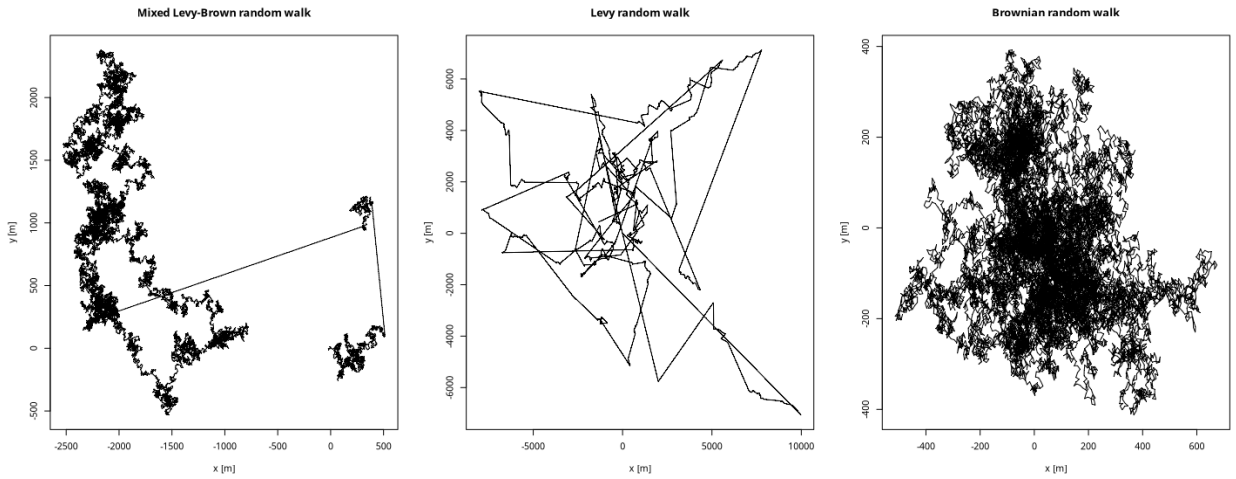

Figure 3.2: Random walks of different types (Mixed, Levy, Brownian), test conducted with one agent and the configuration in listing 2, page 57. The graphs show the x and y coordinates of the agent over 21600 time steps (6 hours). Mind the different labels on the axes (Mixed: x [-2500,500], y [-500,2000]; Levy: x [-5000,10.000], y [-6000,6000]; Brownian: x [-400,600], y [-400,400]). In the mixed type, after each Levy step the agent did Brownian steps for 120 time steps. One can see that the agent moves in a rather small area in the Brownian case compared to the other two cases. Also well visible are the long jumps in the Levy case. In the mixed case, one can easily see the Brownian steps as the agent moves in a small area for a while after each long Levy step jump.

simulation and stop the simulation. It can also store the state of the simulation to a file and restore the state of the simulation from a file.

Parameters

There are three types of hierarchically structured parameters relevant to the simulation. The global parameters, which are shared with the animation, include for example the output directory. They also include a SimulationParamters and an AnimationParameters object, to ensure that the simulation and animation are in sync. There exists a single GlobalParameters object only, which is created at the beginning of the program and passed to all other objects. It is also responsible for reading the configuration file and parsing the command line arguments (with the help of third party libraries). Of the SimulationParameters exist multiple instances, one for the running simulation and one for the parameter editor in the GUI. This way parameters can be edited and stored during runtime, while the simulation is running. The simulation parameters include for example the worldsize and the timestep. Time related parameters are automatiaclly scaled to the timestep, so the user does not need to worry about the units and only take care of a consitent relation of time and space in the units he uses. The agents are parameterized by the AgentClassParameters object. For every agent class, there is a corresponding AgentClassParameters object, which includes the parameters for the agents of that class. To allow for an unlimited amount of classes which must not be known at compile time, the SimulationParameters

object can store an arbitrary amount of AgentClassParameters objects in a vector. The number of agent classes is then defined by the configuration file or at runtime from the GUI. Every time the simulation state is saved, of course the parameters are saved as well. A key simulation parameter to be mentioned last is the random seed, which is used to initialize the random number generator. The agent movement includes a lot of randomness, because in our model the agents (bats) search for food using random directions, so the user can set the seed to reproduce the same simulation multiple times, and guarantee a reproducible result of his experiments. Of course, the seed can be set to a random value as well, to get different results every time and do a statistical analysis of the simulation.

3.1.7 Collecting Data

To allow for maximum flexibility and not restrict the user in evaluating the simulation results, it is possible to log the full state to a file in every iteration. Thus, evaluation of the simulation results can be done at any time after the simulation has finished. The simulation provides five predefined log levels, which can be used to control the amount of information that is logged. With the highest log level, the full state of the simulation is logged in every iteration, including the meta information of the simulation state. To calculate the meta information is of O(N^2) complexity, such as the distances agent to agent. As it can be calculated from the state variables, it is not necessary to log it. But as this information is already precomputed in a fast and efficient way, it might save a lot of time to log it, if it is needed later for statistical analysis, compared to calculating it again with R, Python or Excel. The software also provides the necessary functionality for data aggregation on system and agent level at runtime. This is provided by the AgentMetrics and SystemMetrics classes. But as this is use-specific, this functionality is intended for advanced users who want to implement their own data aggregation, to save time compared to first logging the full state and then aggregating the data.

3.2 Animation

This section describes the part of the software responsible for the 3D visualization. It is based on the OpenGL graphics library and the Dear ImGui library for the graphical user interface. The UML-diagram 3.3 on page 19 shows the involved classes and their relationships. Central to the visualization is the Animation class, it coordinates the visualization, keeps it in sync with the simulation and provides the user interface. The Shader class is responsible for rendering the 3D scene and the transformations of the scene done by the graphical processing unit (GPU). Graphical objects are splitted in the Point-Cloud objects which display the agents and Grid, Axes and BoundingBox objects, which display what their names suggest. All of these objects and their visual representation can be controlled by the user through the graphical user interface (GUI). The GUI is based on the open source library Dear ImGui [20]. This library provides a simple and easy to use interface for creating GUIs in C++. To keep the code clean and maintainable, for every object that can be controlled by the user, a separate ImGuiHandle class exists, named after the object it controls. The user can also control the simulation through the

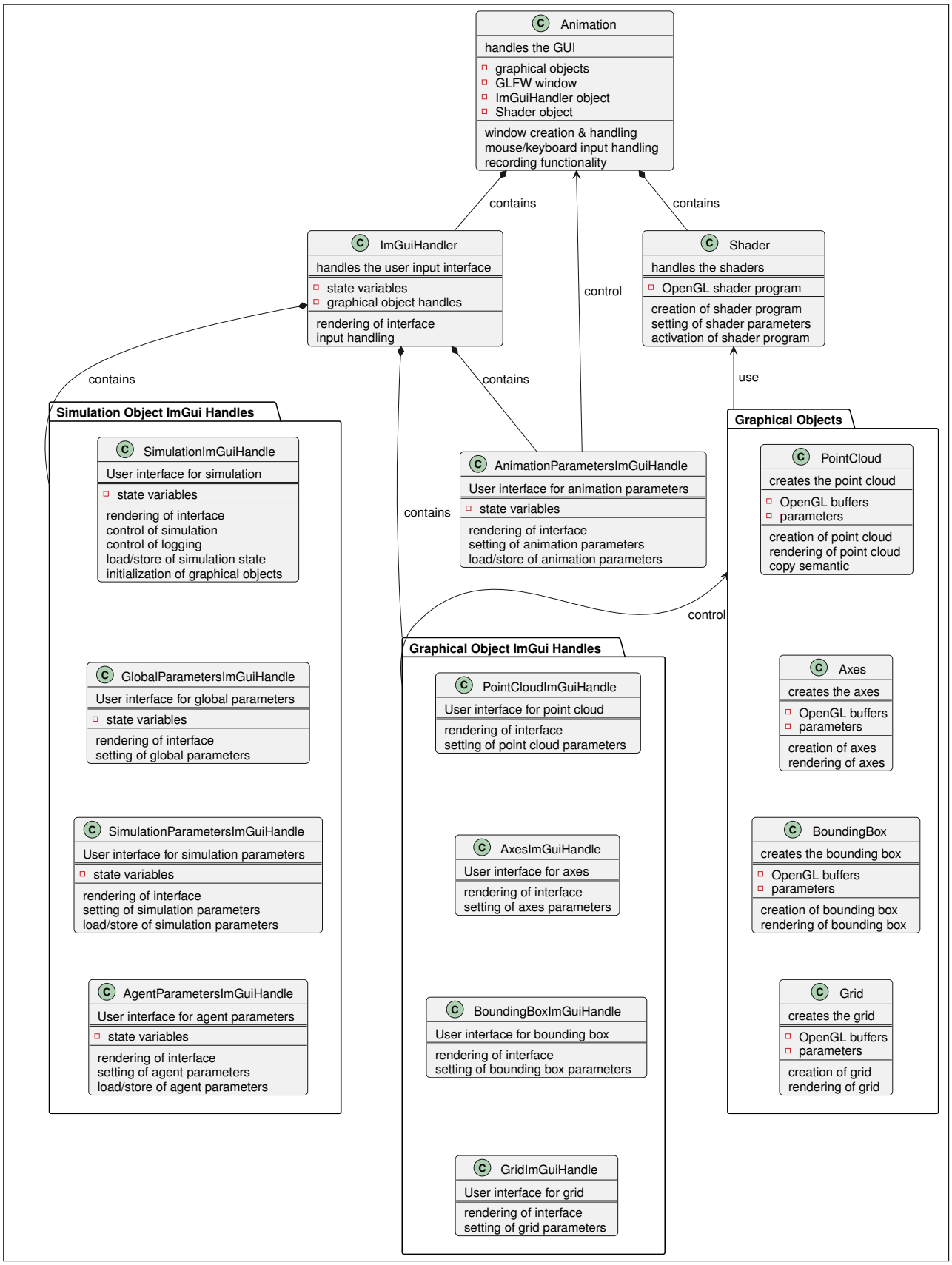

Figure 3.3: Simplified UML class diagram of the animation

GUI, thus there exists a second group of ImGuiHandle classes, which control the simulation. Please refer to the UML-diagram 3.3 on 19 for a detailed overview of the classes and their relationships. Analogously to the simulation section, this section explains all the classes shown in the diagram in detail, which are responsible for the visualization. The visualization is completely independent of the simulation, so it can be used without the simulation and vice versa. It can be used as a standalone tool for visualizing 3D data, as long as the data is provided in the correct format. Even the simulation control user interface can be used with any simulation, as long as the simulation provides basic functions like start, stop, and reset. As visualizing 3D data is often necessary in scientific research, this tool can be reused in a wide range of applications to save a lot of time and effort.

3.2.1 The Setup

The animation setup is based on OpenGL, GLAD, GLFW, GLM and Dear ImGui [27, 28, 18, 19, 20]. While GLM is responsible for camera transformations and ImGui for the GUI, OpenGL, GLAD and GLFW are used for rendering and window management. OpenGL is the main graphics API used in this project, GLAD is used to load OpenGL functions and GLFW is used to create windows and handle input. This major setup is handled by the Animation class, which is responsible for initializing the window, setting up the OpenGL context and handling the main loop. Details of the major setup are not explained any further in this document, because the setup was done according to the GLFW documentation and the OpenGL tutorials on learnopengl.com. These contain a step-by-step guide on how to set up a window, an OpenGL context and how to render objects. The Shader class completes the setup by loading and compiling shaders, which are used to render objects on the screen. It compiles the shaders, which are written in GLSL, and links them to a program on the GPU. Later the shaders can be used to render objects on the screen and to apply transformations to them or change their appearance. If the program is run with animations enabled, and endless loop in the Animation class will call the render function of the Animation class, which will render the scene and update the GUI. To keep the animation responsive, the animation including the GUI is updated at a fixed rate of 60 frames per second. This is possible because animation and simulation are decoupled, so the simulation can run at a different rate than the animation. Both run in separate threads using OpenMP, which is a library for parallel programming in C++ [29]. Without the decoupling, the simulation would have to run at the same rate as the animation, which would slow down the simulation and make the animation less responsive.

3.2.2 Camera

The so called camera is a mathematical transformation on the scene, which consists of objects defined by coordinates in a 3D space. It is a transformation that maps the 3D scene to a 2D plane, which is the image plane. For the animation a museum view is used, which is a perspective projection. It does not allow the user to move freely in the scene, but it is possible to rotate the scene around the axes, zoom in and out and move the scene in the x and y direction. This means, the camera is fixed in the scene and the scene is moved in front of the camera. Internally, the coordinates of an object are loaded to

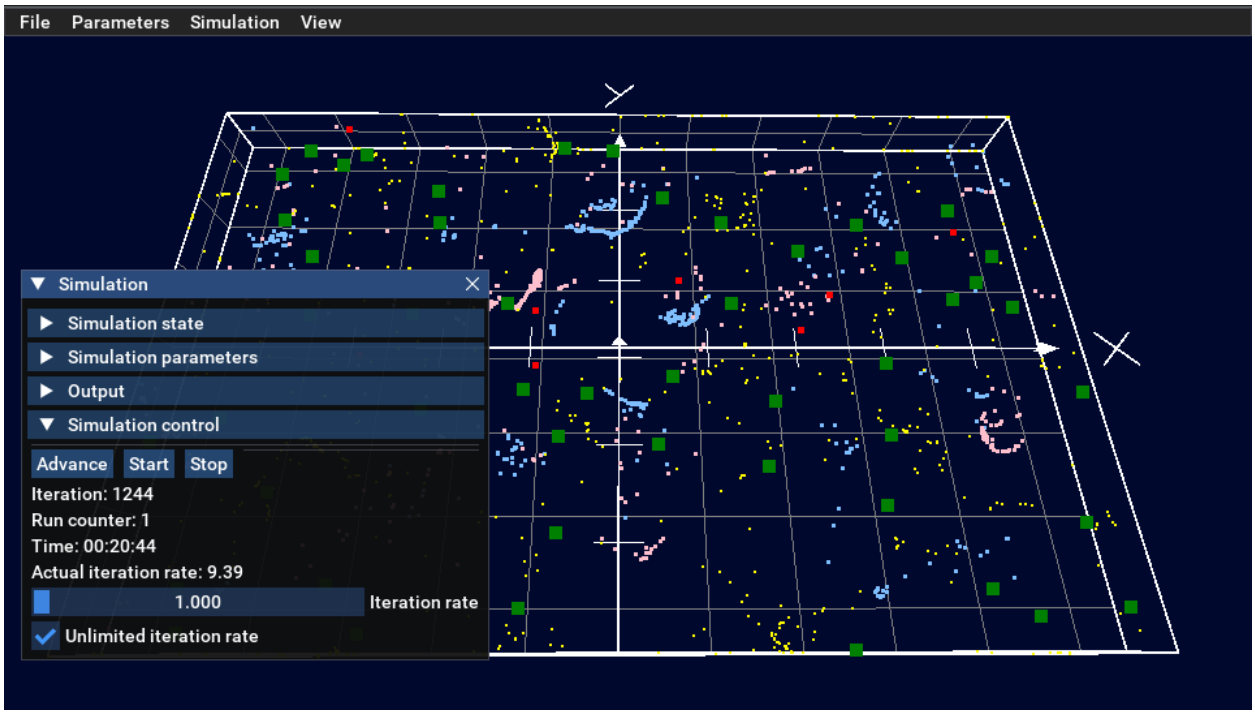

Figure 3.4: The tool simulating a predator prey system of 2060 agents in five classes. Green (plants), yellow (insects), pink (female bats), blue (male bats) and red (owls) dots show the agents, white lines form the bounding box, grid and axes, in the bottom left the simulation control dialog is visible.

a GPU buffer, they do not change, but the transformation matrix of the camera is changed. With the help of GLM, a projection view matrix is created, which is then uploaded to the shader program on the GPU. This transformation matrix is multiplied with the object coordinates in the vertex shader, which results in the final coordinates on the screen. The GLM documentation provides a good explanation of the projection view matrix and how it is created [19].

3.2.3 Graphical objects

In the context of the animation, graphical objects refers to objects of the PointCloud, Axes, Grid and BoundingBox class. All these are represented by a set of 3D coordinates which are individually animated. The point clouds are used to represent the agents. The bounding box shows the boundaries of the simulation space. Axes and grid help to orient the viewer in the 3D space and give a sense of scale. All graphical objects provide an inititialization and a draw method. While axes, grid and bounding box are static, the point cloud also provides an update method to animate the agents. Figure 3.4 on page 21 shows an example of a predator-prey simulation with a bounding box, axes, grid and five point clouds representing the agents.

Point Clouds The animation can handle an arbitrary number of point clouds. For each class of agents, a point cloud is created. These can be rendered in different colors and with different sizes.

Each point cloud can be switched on and off individually. Point clouds are not limited to agents, they can be used to represent any set of 3D coordinates. All that is needed is a pointer to the coordinates and the number of points. Coordinates must be of type float and ordered in the following way: $x_1, y_1, z_1, x_2, y_2, z_2, ...$

Axes, Grid and Bounding Box The axes, grid and bounding box are static objects. They are drawn once and do not change during the animation. Of course they are rotated by the camera with the rest of the scene. At initialization, they need to be provided with the size of the simulation space. The axes also provide tick marks and labels. Number of tick marks and number of grid lines can be set at runtime by the user. Depending on the number, the Axes and Grid class will calculate the end points of the lines and draw them. The grid can be drawn on any outer face of the bounding box. By default, the sides of the bounding box at which grids are drawn is based on the orientation of the camera. This is achieved by calculating the dot product of the camera direction and the normal vectors of the bounding box faces. Based on the sign of the dot product, the grid is drawn on the corresponding face.

3.2.4 Graphical User Interface

The graphical user interface is a very important part of the software. It would not be necessary, but without it, all parameters would have to be set in the config files before starting the program. While this is still possible in the tool, all parameters can be set in the GUI. This includes the parameters for the simulation, the parameters for the visualization and the parameters for the data output. Furthermore, the GUI allows the user to start, stop or reset the simulation. And last but not least, it allows saving and loading of the simulation state and the parameters. The GUI is implemented using the Dear ImGui library [20]. This library is very easy to use and can be integrated into any OpenGL application. It is fast, lightweight, has no dependencies and is easy to use. ImGui does not provide any classes, it is just a namespace with functions. Therefore different parts of the GUI were bundled into classes called ImGuiHandles. These can be grouped into handles for the graphical objects and handles for the simulation.

Graphical Object Handles Each graphical object has its own ImGuiHandle. This handle is responsible for the GUI elements that are specific to the object. These graphical object handles allow the user to change visual properties of the objects, or to switch the object visualization on or off. For the point clouds, the user can change the point size and the color. In case of the axes the user can change the number of ticks and the tick length. For the grid, the user can change the number of lines and switch autoselection based on view off and choose the sides of the bounding box which should be drawn as a grid manually.

Simulation Handles The simulation handles interact with simulation class objects. They also correspond to single classes mainly. Most important is the handle for the Simulation class. It allows the user

to start, stop and reset the simulation; and to save and load the simulation state. Also the log level and the verbosity of the simulation can be changed. Simulation speed can be changed and the user can choose to run the simulation as fast as possible. Second most important is the handle for the simulation parameters. It allows the user to change the parameters for the simulation. The tool includes a parameter editor, which is independent of the simulation. Parameters, once set to the user's liking, can be saved, loaded and used in the simulation. Agent classes can be added, removed, saved and loaded separately. An example session of editing parameters via GUI is shown by figure 3.5 on page 24. The simulation control handle can be seen in figure 3.4 on page 21. The simulation handles complete the tool and make it a fully graphical application.

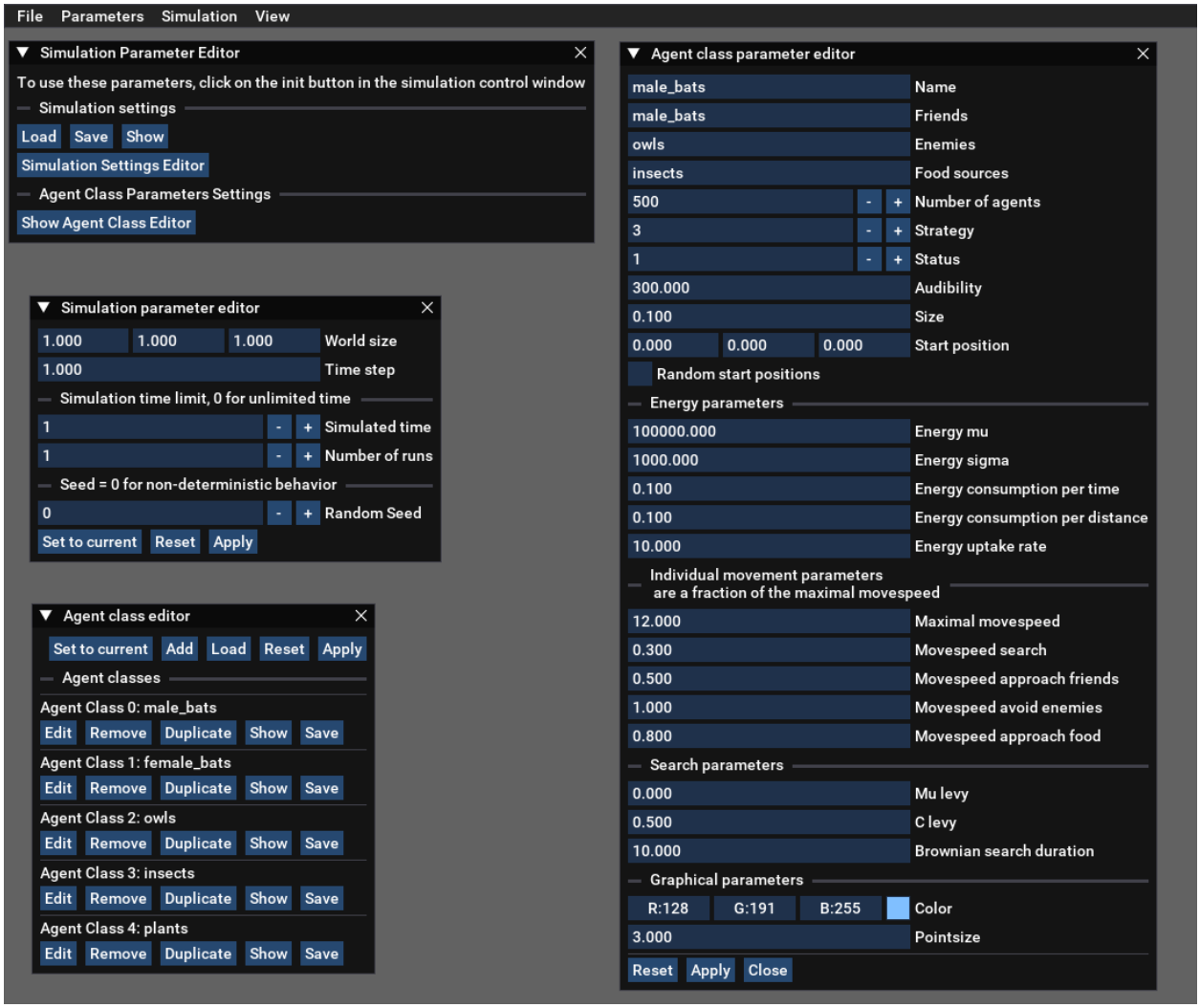

Figure 3.5: Screenshot of parameter editing with the tool, general simulation parameters can be edited with the 'Simulation parameter editor' (top and middle left), agent classes can be added, removed loaded and saved with the 'Agent class editor' (bottom left), and agent class parameters can be edited with the 'Agent class parameter editor' (right).

Chapter 4

Illustrative Examples

In this chapter some illustrative examples of the framework are presented. They show the effect of different search strategies and the effect of communication. The parameters used are set to more or less realistic values, but the tests are not biological experiments. The purpose of these examples is to show the potential of the framework and to give an idea of the kind of results that can be obtained.

4.1 Search strategies

This example shows the effect of the 'brownian_search_duration' parameter in a simulation of 1000 bats and 42 moving insect swarms over 21600 time steps, equivalent to 6 hours of simulated time. The runtime of the three scenarios was about a minute each. The example compares three scenarios in which only this parameter is varied. The first scenario uses the default value of 0, the second of 120, and the third of 21600. A value of 0 means that the brownian search is disabled and the agents will only use Levy flights. A value of 120 means that the agents will use brownian search for 120 seconds after each Levy flight. A value of 21600 means that the agents will use brownian search for the entire simulation. The search parameters correspond to the resulting random walks displayed in figure 3.2 on page 17, which have been discussed in section 3.1.5 on page 16. The bats had to find the insect swarms in an area of 400 square kilometers, and communicated if they found one for a distance of 300 m. The world had 400 m in height and the bats were able to fly at a maximum speed of 9.6 m/s to approach food and searched at a speed of 3.6 m/s. All bats started in the center of the area. Insect swarms were randomly distributed over the area and audible to the bats in a radius of 200 m and performed a slow Levy random walk. The number of insect swarms is kept constant over time.

Table 4.1 on page 26 shows the success rates of the three scenarios. All bats started with an initial energy of 1, energy consumption per time and distance were set to 0. The results show the sum of energy gained by the bats by catching insects. As one can see, the Brownian search strategy, which covers rather small areas (see figure 3.2), is the least successful. In this scenario the bats in average were able to gain 915 energy units. This sums up to the energy contained in 18 swarms of insects. The mixed scenario with a duration of 120 time steps of Brownian search after each Levy flight was

more successful. The bats in average were able to gain 3048 energy units. This sums up to the energy contained in 61 swarms of insects. Most successful were the bats using Levy flights only. They were able to gain 6492 energy units in average. This sums up to the energy contained in 130 swarms of insects.

Figure 4.1 on page 27 shows the mixed search scenario after 10 minutes of simulation time. One can see that the bats (blue) randomly spread over the area starting from the center (roost). The insect swarms (yellow) are sparsely distributed over the area and are moving slowly. One grid cell corresponds to 1 square kilometer. Figure 4.2 on page 28 shows the same scenario after 6 hours of simulation time (21600 time steps). One can see the scenery from a different angle. Bats have spread all over the area, but big clusters in the center are still visible. The two screenshots also illustrate well the 3D animation capabilities of the simulation.

Listing 4.1 on page 29 shows the behaviour algorithm of the bats. It illustrates the ease of use of the simulation framework. One can see that it is implemented using simple if-else statements and the provided application programming interface (API). These convenience functions can be used to implement complex behaviours in a few lines of code. Not counting blank lines and comments, the behaviour algorithm consists of 18 lines of code, compared to the more than 130,000 lines of code of the total framework (see section 5.9 on page 44). Behaviour algorithms are intended to be implemented by the user and can be easily exchanged. Arbitrary numbers of behaviours can be implemented and combined to complex move strategies which can be selected with the 'move_strategy' parameter.

4.2 Communication

This example shall demonstrate the effect of communication between agents on their foreaging success. The setup is the same as before, but the bats are now split into two groups, each with a different foraging strategy. The first group of bats will use the same strategy as before (see listing 4.1), while the second group will not communicate information about food sources with each other. Listing 4.2 shows the algorithm used by the second group of bats. This one is even more simple to implement, it only needs 11 lines of code. Agents don't move if they are at a food source, otherwise they move towards the nearest audible food source, else they search for food.

Generating the configuration for this example was done using the parameter editor of the framework by using the duplicate feature for agent classes and changing only the strategy parameter. This

Strategy	Average energy uptake
Brownian search	915
Mixed search	3048
Levy search	6490

Table 4.1: Average predator energy uptake for different search strategies. Setup with 1000 predators and 42 prey. Predators start with 1 unit of energy. One can see that the Levy search strategy is the most effective. Configuration see listing 5, p. 62, random seed set to 1.

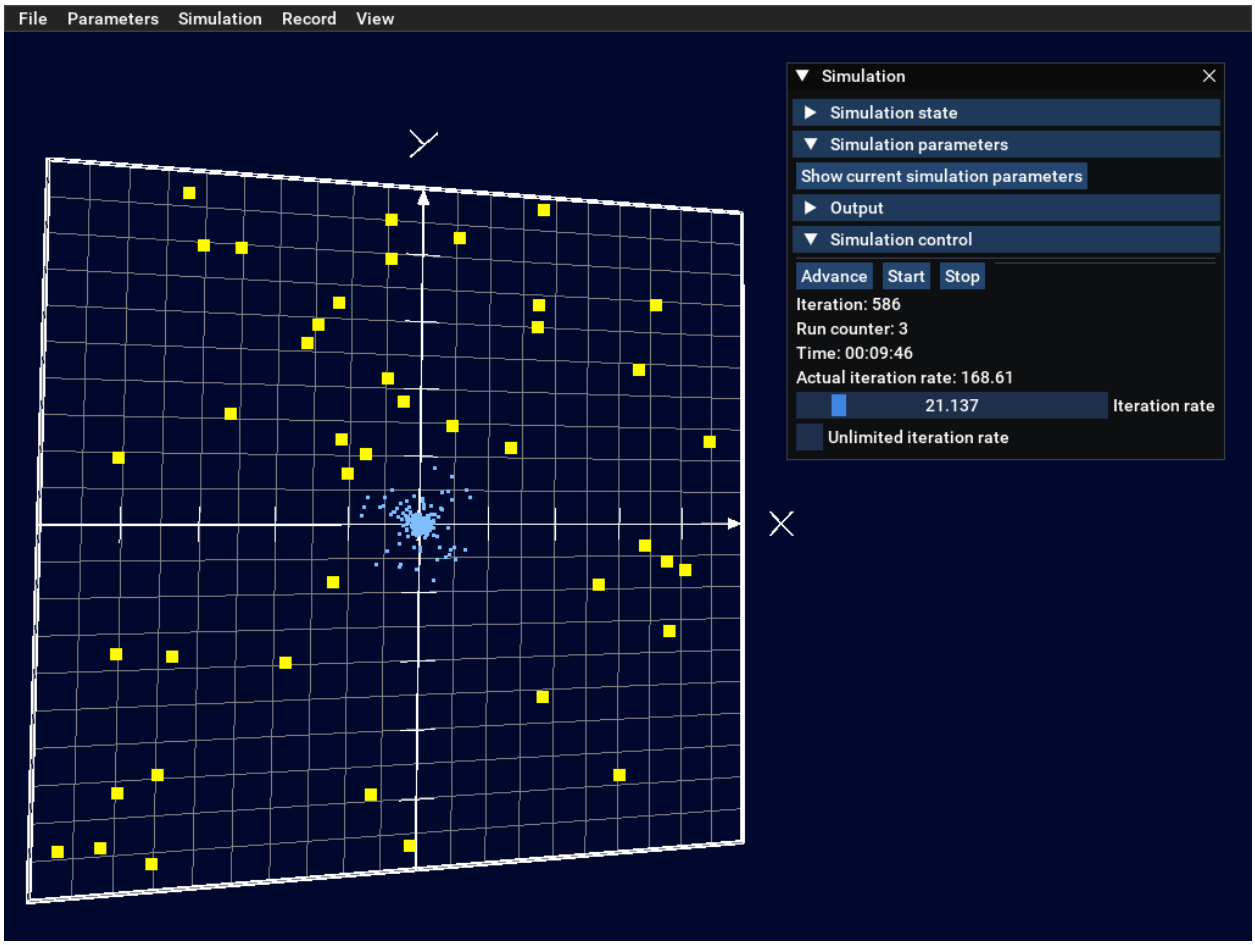

Figure 4.1: Screenshot corresponding to the mixed search scenario described in section 4.1 on page 26. It shows the scenery after 10 minutes of simulation time. Yellow dots represent the insect swarms, blue dots show the bats.

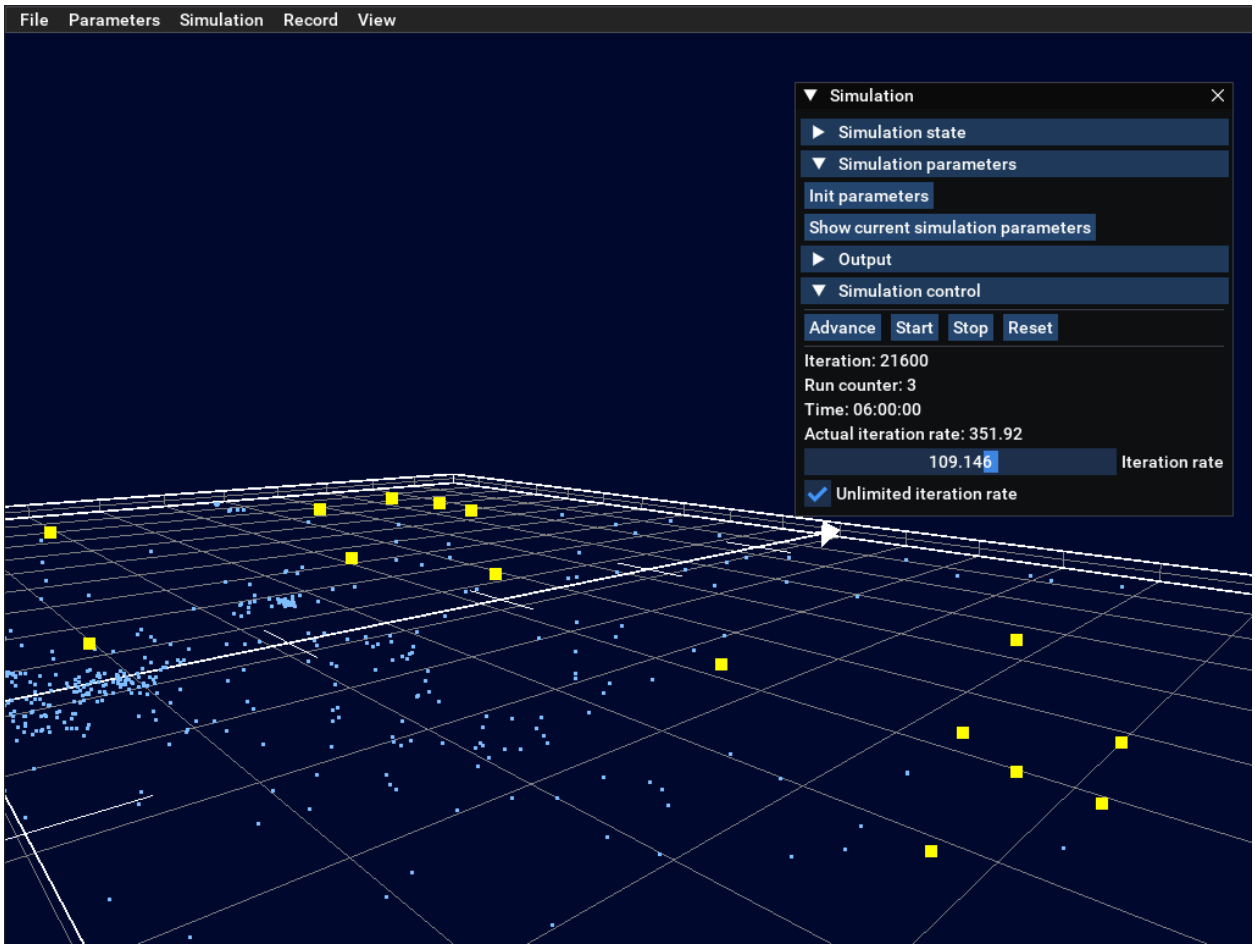

Figure 4.2: Screenshot corresponding to the mixed search scenario described in section 4.1 on page 26. It shows the scenery after 6 hours of simulation time. Yellow dots represent the insect swarms, blue dots show the bats.

Listing 4.1: The bats' behavior algorith correponding to the illustration example for the effect of the search strategies

```
template <typename T>
Point3D<T> Agent<T>::moveStrategy2(const State<T> &state)
{

  /* Communicate food sources, search and approach food */

  // If at food source, stay
  if (foodContact(state)){
    return (Point3D<T>(0.0, 0.0, 0.0));
  }

  // Approach food sources, if some are known
  if (foodAudible(state)){
    return approachClosestFoodSource(state);
  }

  // If no food sources are known, approach friends which hear food
  if (friendAudible(state)){
    Point3D<T> move;
    move = approachClosestFriendsHearingFood(state);
    if (!move.isZero()){
      return move;
    }
  }

  // Search, if no other strategy applies
  return search(state);

}
```

illustrates the ease of creating different configurations with the framework, while at the same time using the parameter editor ensures that the configuration is valid.

Figure 4.3 on page 31 shows three different views of the scenery for the mixed strategy example after 6 hours of simulation time. It only shows the inner 2x2 km of the scenery, such that individual agents can be seen. By using the frameworks feature to only display selected agent classes, the effect of the different strategies can be visualized. On the left side, the view of the scenery with only the agents not sharing information is shown. The middle view shows the scenery with all agents, while the right view shows the scenery with only the agents sharing information. One can clearly see, that the agents sharing information are stronger clustered, while the agents not sharing information are more evenly distributed over the area. The cluster at 6 o'clock at about 200 m below the center (distance between grid lines 1000 m) of both agent groups is the result of a gathering of agents at a food source, which at the simulation time of taking the screenshots was already depleted. This feature to selectivly display agent classes is very useful for visual interpretation of simulation results.

Table 4.2 on page 30 shows the results of the simulation. One can clearly see that the agents sharing information are more successful in finding food. Sharing information led to more than 50% increase of foraging success. Interestingly, the benefit is the largest if agents use the Brownian search strategy, here food uptake increased by 86%. Probably this is related to the lower total food uptake of the Brownian search strategy, as the area in which agents search for food is smaller and the scarcity of food is higher. This example nicely illustrates that the framework is well suited to study the scientific question posed by Dr. Kamran Safi wether bats do benefit from the information aquired about the location of food sources from echolocation calls of other bats. In this example, the exchange of information increased the bats' foraging success significantly and let to the formation of clusters, which hints that the sharing of information could play a major role in the formation of bat collectives.

	Energy uptake		
Strategy	Group A	Group B	Factor
Brownian search	939	505	1.86
Mixed search	2659	1779	1.49
Levy search	5200	3491	1.49

Table 4.2: Average predator energy uptake for different search strategies. Setup with two groups of each 1000 predators, group A shares information about food sources, group B does not. Predators start with 1 unit of energy. One can see that the Levy search strategy is the most effective and the Brownian search strategy the least effective. Sharing information about food sources increases the energy uptake for all search strategies. The benefit of sharing information is more pronounced for the Brownian search strategy. Configuration same as in listing 5, with the bat class configuration duplicated and differing only in using different strategies, p. 62, random seed set to 1.

Listing 4.2: The non-communicating bats' behavior algorithm correponding to the illustration example for the effect of the information exchange on foreaging success. Also showing the use of convenience functions to model complex behaviour

```
template <typename T>
Point3D<T> Agent<T>::moveStrategy1(const State<T> &state)
{

  /* No communication, only search and approach food */

  // If at food source, stay
  if (foodContact(state)){
    return (Point3D<T>(0.0, 0.0, 0.0));
  }

  // Approach food sources, if some are known
  if (foodAudible(state)){
    return approachClosestFoodSource(state);
  }

  // Search, if no other strategy applies
  return search(state);

}
```

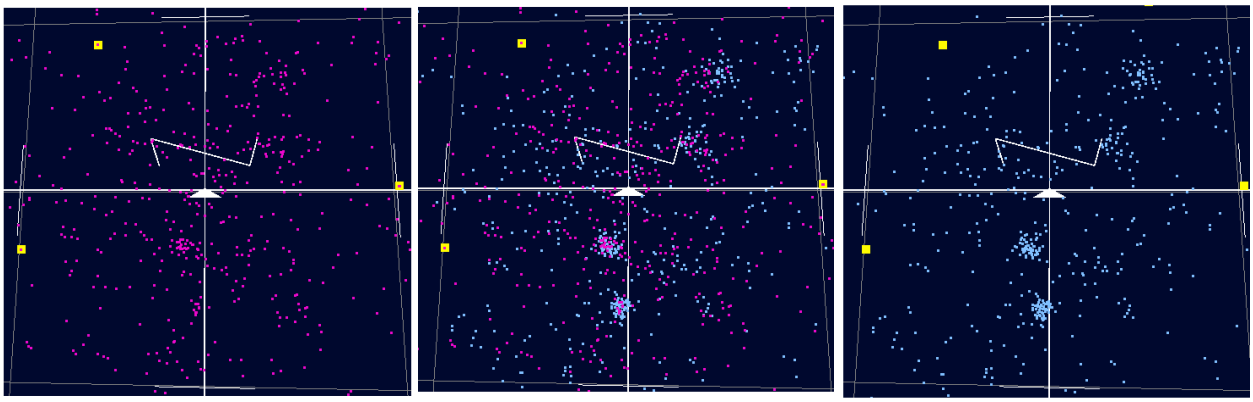

Figure 4.3: This figure shows the effect of information exchange on the formation of bat collectives. It corresponds to the scenario described in section 4.2. Showing three different views of the same scenario. Left only showing the group of bats which did not communicate (pink), right only showing the group of bats which did communicate information(blue), middle showing both. Yellow dots show insect swarms. One can see, that the group which shares information about food sources is more clustered. The white zig-zag line above the center is the 3D Z axis label.

Chapter 5

Computational aspects

This chapter describes computational aspects like numerical stability, memory alignment, parallelization, and performance optimization. All tests were performed on a workstation with an Intel Xeon W-2195 CPU and 64 GB of DDR4-2666 RAM. The program version used for the tests is in the commit number 52ffe74ceec604065ea6e5877d864012e22d1ef8 of the development repository (`https://gitlab.mpcdf.mpg.de/bsim/bsim.git`), branch 'release.working'. For reproducibility, the configuration used is attached in the appendix (see listing 1), and the tool was run with a fixed random seed of 1.

5.1 Precision

All simulation classes are templetized, by default the simulation runs in float precision (32 bit), but the user can change it to double precision (64 bit) by changing the template parameter to double. If the precision is double, animation must be disabled, or the agent coordinates must be converted to float and stored in a separate array for animation. Currently, animation does not support double precision. Float precision was chosen as the default because it is faster and uses less memory and it is fully sufficient for the simulations in this project.

5.2 Numerical stability

According to Higham, there are three main sources of errors in numerical computation: rounding, data uncertainty and truncation [30]. Data uncertainty can be excluded in this project, because the algorithm does not process any input data. Truncation errors are introduced by the discretization of continuous problems. They cannot occur in this project, because the algorithm is based on a discrete model. Thus rounding errors are the only possible major concern in this project. To evaluate the influence of rounding errors, it is necessary to first have a look at our model. There are only two variables of major importance for the evaluation of the simulation results, the position and the energy of the agents. The position usually changes in a non-contiguous way, because the agents move in discrete steps. In

every iteration a vector is added to the position of the agent to calculate the new position. This is an addition or substraction operation, for which the error is in the order of the machine epsilon. Addition and substraction are numerically stable operations [30]. The energy of the agents is also only changed by addition and substraction. Thus, rounding errors are also not a concern in this project. This has also been confirmed by the test for correctness of the energy conservation in section 5.3.

5.3 Correctness

Correctness means on the one hand, that the algorithm is correct, such that the agent behaviour is as expected, and on the other hand that computations are correct.

Correctness of the Agent Behaviour Algorithms

The term 'agent behaviour' in this context does not refer to the resulting actions of the agents, but to the algorithms that determines the actions of the agents.

As the agent behaviour algorithms are to be implemented by the user according to the user's needs, correctness of the agent behaviour algorithm is in the user's own responsibility. The framework only guarantees that the convenience functions are correct. The user has to ensure himself that his agent behaviour algorithm is correct. Very helpful for testing the correctness of the agent behaviour is the possibility to visualize the agent behaviour in the simulation. Often faulty behaviour can be detected by visual inspection of the simulation at a glance. The user therefore has to think of simple test cases to validate his agent behaviour algorithm.

Here are two examples of test cases which have been used to validate the correctness of the convenience functions.

Example test case 1 (exact configuration see listing 3 on page 58):

- Setup: several predator agents (bats), one prey agent (insect-swarm)
- Agent movespeed set larger world size (they can travel the whole world in one iteration)
- Food audibility set larger world size (agents can hear the food source from anywhere in the world)
- Algorithm: if food audible, move to prey (provided by convenience functions)
- Expected result: all predator agents move to the position of the prey in one iteration
- Expected result: the prey energy is reduced by the number of predator agents times the predators' 'energy_uptake_rate' parameter
- Expected result: if the food source energy drops below zero, the food respawns at a new position

Example test case 2 (exact configuration see listing 4 on page 60):

- Setup: One predator agent (owl), one prey agent (bat)
- Predator movespeed set larger world size (it can travel the whole world in one iteration)
- Predator and prey audibility set larger world size (they can hear each other from anywhere in the world)
- Predator algorithm: if prey audible, move to prey (provided by convenience functions)
- Prey algorithm: if attacked by predator, stay in place (provided by convenience functions)
- Prey algorithm: if predator audible, move away from predator (provided by convenience functions)
- Expected result: first iteration, predator moves to prey, prey moves away from predator
- Expected result: second iteration, prey is immobilised, predator moves to prey
- Expected result: following iterations, prey energy is reduced by the predator's 'energy_uptake_rate' parameter
- Expected result: if prey energy drops below zero, prey is respawned and the same behaviour is repeated

This test case is interesting because it shows the importance of the 'attacked' status. An agent gets automatically the status 'attacked' if it is within reach of its predator. Otherwise, the predator would always move to the last known position of the prey, which would result in a never ending chase. Other test cases are similar to the above examples. All implemented convenience functions have been tested in this way to validate their correctness.

Correctness of Computations

The correctness of the computations is ensured by the framework. On the one hand side there are major parts of the framework that are implemented in a way that they are correct by design. For example the distance computations use the distance method of the Point3D class. As it is correct for two points, it is also correct for any number of points. Testing is simple and can be done by comparing the results of the computation with the expected results.

As energy exchange is a very important part of the framework, it is also guaranteed that it does not contain any errors, if the user uses the provided convenience methods for energy exchange correctly. This has been tested by summing up the energy turnover and all energy removed and added to the system. The debug build provides a test that checks the correctness of the energy exchange. As a matter of fact energy exchange is subject to small rounding errors that cannot be avoided. Fortunately, these errors do not accumulate to a significant amount in the intended use of the framework. There is a small amount of error that is acceptable and does not affect the simulation. In the long run, the sum

of the error tends to be negative, which means that the system loses energy. This is due to the fact that agents are removed from the system if their energy is below a certain threshold close to zero. We call this the 'removing error', it is explained in more detail below. For the model this is not a problem at all and the loss of energy can be interpreted as dissipation of energy in the system.

Table 5.1 on page 36 shows the error of the energy exchange in a simulation with 2060 agents, run in floating point precision. The simulation is run with the configuration shown in listing 1 on page 53. The test can be repeated with the command (debug build):

```
./bsim -n 3 -t N -v 1 -s 1 | grep 'SystemMetrics'
```

where N is the number of iterations to run the simulation. This will run the simulation 3 times with a fixed seed and print the total energy turnover and the sum of all energy added and removed from the system.

The table shows the absolute error of the energy exchange in the system. This error is caused by rounding errors in substraction and addition of energy exchanged between agents and between agents and the environment. Another source of error is the respawning of agents. To ensure that there is not more energy in the system than intended, agents are removed from the system if their energy is below a certain threshold. This variable (DEATH_THRESHOLD) is by default set to 0.1, to avoid numerical issues which would occur if it gets close to zero. So every time an agent is removed from the system, up to 0.1 energy is lost.

As one can see in the table, the relative error per iteration decreases with the number of iterations and the 'removing error' outweighs the rounding errors. This is the reason why the error in the long run is always negative and can be interpreted as dissipated energy. While the absolute error is O(10^1) for a single iteration and O(10^0) for 100 iterations, it is O(10^{-1}) for 10, 1000 and 10000 iterations. [1] This absolute error becomes meaningful in relation to the total energy in the system, which in this test is O(10^8), resulting in a relative error of O(10^{-7}) for a single iteration and O(10^{-9}) in the long run. Energy turnover and netto in/out energy are O(10^3) per iteration.

As a summary, rounding errors are outweighed by the 'removing error', which is the main source of error. To keep the 'removing error' in a reasonable range, the removing threshold must be kept in mind when setting the energy levels and energy exchange rates of the agents. In the test configuration, a system with 1000 bats, 1000 insects, 50 trees and 10 owls, the removing error sums up to the energy contained in one insect every 100 iterations, which is definitely not a problem (compare listing 1 on page 53). As a rule of thumb the 'removing error' or 'dissipating energy'can be approximated by the number of agents expected to die per iteration times the removing threshold. If exact energy preservation is needed, the removing threshold could be reduced, but it had to be kept in mind that this could lead to numerical issues if it is set too close to zero, so it should not be done without testing correctness of rounding errors for the specific case. For the intended use of the framework, the default value of 0.1

[1] O(10^x) means the error is in the order of n times 10 to the power of x, where n < 10, called big O notation or Bachmann-Landau notation.

does not introduce any problems and ensures that there are no numerical issues and only neglectable rounding errors.

5.4 Algorithmic complexity

While the prototype had an algorithmic complexity of nearly O(N^3), the final implementation has a complexity of O(N^2) (where N is the total number of agents). This is a significant difference, because this means that runtime increases quadratically with the number of agents, instead of cubically.

The major difference between the two implementations is the way information is communicated between agents. In the prototype, for example, agents searching food were asking other agents in reach for the closest food source they knew of. This resulted, in the worst case, in nearly O(N^3) complexity, N agents asking N agents to do a minimization on M food sources. For example with N=100 agents and M=100 food sources, cubic complexity means that with $N * N * M = 1,000,000$ operations needed, which is $\frac{1}{8} * (N+M)^3$, while quadratic complexity results in $(N+M)^2 = 40,000$ operations only, which is the complexity of the final version. Even if this case was not the most common, it was still a significant bottleneck and the major design flaw of the prototype.

For the algorithm, the agents need to know whether other agents are in contact, reachable or audible. This means their distances has to be compared to three different thresholds. The result of these comparisons is stored in matrices, which are used to calculate the agents' behaviour. In the final version this information is calculated by the StateMachine for all agents instead of being calculated 'on demand' by the agents themselves. This meta information forms the conciousness of the agents and is used to determine the agents' behaviour. It is stored in shared memory and can be accessed by all agents. Calculating this information is only O(N^2) because each agent is checked against every other agent only once. Besides the major benefit of reducing the computational complexity by one order of magnitude, there are two additional benefits of the final implementation: By calculating the meta information which is needed for the agents' decisions in the StateMachine for all agents at once, cache efficiency is increased which compensates for the information which is calculated but not used, and the meta information can be logged and used for statistical evaluation of the simulation.

Making use of shared memory, the final implementation reduced the complexity to O(N^2) by not implementing any communication. The complexity results from the quadratic complexity of calculating the meta information of the state, there is no communication between agents. Instead of really com-

No. Iterations	1	10	100	1.000	10.000
Run 1	-15.2	-19.5	358.8	449.4	-1856
Run 2	-15.2	-20.9	380.0	529.2	-1529.5
Run 3	-14.9	-19.0	458.4	191.4	-3654.5

Table 5.1: Absolute energy exchange rounding error in a simulation of 2060 agents after the given number of iterations. The total energy of the system is O(10^8), and energy turnover is O(10^3) per iteration. All units are simulation units. The configuration is shown in listing 1 on page 53.

municating information between agents, agents can access other agents' information directly, stored in the meta information of the State object. Which information is shared between agents is determined by the behaviour algorithm of the agents. For example agent A can look up in the meta information of the State object, whether agent B is audible, reachable or in contact and if Agent C (for example a food source) is reachable to agent B - without any communication between the agents. The State object also stores agent-to-class and class-to-class information, for example agent A can also look up if agent B is in reach of any agent of class C at all. This information is enough to implement the algorithm and to reduce the complexity by one order of magnitude by using shared memory instead of communication.

Using the linux perf tool, the final implementation was profiled and the most time consuming functions were identified. The result is shown in table 5.2 on page 38. It shows the profiling results of the final implementation being run with the test configuration for 1000 time steps (see listing 1 on page 53). One can clearly see that most of the heavy computation is done by the StateMachine. The top ten functions calculate agent-to-agent or agent-to-class information, these calculations are O(N^2) and in sum make up for more than 90% of the total runtime, thus the complexity of the final implementation must also be O(N^2).

To experimentally verify the O(N^2) complexity, the final implementation was run with different numbers of agents. Therefore, the tool was compiled to a sequential version and run on a single core. This ensures that the runtime is not influenced by parallelization overhead. All tests were run for 1000 time steps. The problem size of the test config (see listing 1 on page 53) was increased in 10 steps from 206 agents to 2060 agents. For the smallest problem, the number of agents of each class of the test config was divided by 10. For each step, the number of agents was increased by the number of agents of the smallest problem. Each problem size was run three times and averaged. The measurements were conducted using perf stat. As the first four problem sizes were sampled at a CPU frequency of 3.6 GHz and the last six at 2.3 Ghz, the runtime of the first four problem sizes was scaled by the ratio of the clock speeds (3.6/2.3=1.57), because for comparability, the runtime should be measured at the same clock speed. Figure 5.1 on page 41 shows the runtime of the final implementation for different problem sizes. The measured runtime is shown as circles, the expected runtime is shown as a line. One can see that the circles are almost exactly on the line, indicating that the measured runtime scales almost exactly quadratically with the number of agents.

% of runtime		Function
-O0	-O3	
14.52	NA	Point3D<float>::distance
13.60	16.96	StateMachine<float>::calculateContactAgentToAgent
12.16	13.45	StateMachine<float>::calculateAudibilityAgentToAgent
10.72	13.77	StateMachine<float>::calculateReachabilityAgentToAgent
9.26	3.51	StateMachine<float>::calculateDistancesAgentToAgent
7.16	10.31	State<float>::calculateAverage
5.07	9.01	StateMachine<float>::calculateAudibilityAgentToClass
5.03	7.99	StateMachine<float>::calculateReachabilityAgentToClass
5.02	6.28	StateMachine<float>::calculateContactAgentToClass
4.38	NA	_sqrt_finite
2.16	7.27	unknown kernel space function (0xffffffff93001074)
1.78	2.91	AgentCass<float>::findMin
1.49	NA	_f32xsqrtf64
0.99	1.81	AgentCass<float>::findMin (doubled due to overloading)
0.72	3.22	OpenMP function (gomp_iter_dynamic_next)
94.06	96.49	Total (all other functions are less than 1 each)

Table 5.2: Relative execution time of the major time consuming functions in the tool. The first column shows the overhead of the function when compiled with auto-vectorization disabled (-O0), the second column shows the overhead when compiled with maximal compiler optimizations enabled (-O3), including autovectorization. The simulation was run for 1000 iterations with the test configuration (see listing 1 on page 53), using 8 threads, with seed 1, sampled at 2.3 GHz. Execution time was measured using perf Perf. Functions with NA in the second column do not exist in the optimized binary.

5.5 Memory layout

As described in section 3.1.2, the system state is represented by a set of variables for each agent. These are stored in the State class objects. Of the State class, several objects are created, representing the state of the system at different points in time. The data layout in the State object is a Structure-of-Arrays (SoA), which is a data structure that stores the data of each variable in separate arrays. This is done to improve cache locality, as the data of each variable is accessed together. In other words, each variable is stored in a separate contiguous array, and the data of each agent is stored in the same index in each array. The data is not grouped by agent, but by variable. As the StateMachine accesses the data of each agent in every iteration to create the meta-information (see section 3.1.3), this layout is beneficial for performance. The meta-information is also stored in the State object, in a similar SoA layout. An exception is the position variable, which is stored in an array of structures (AoS) structure (x, y, z, x, y, z, ...). This is the suitable layout for the animation with OpenGL and has been tested to have no significant impact on performance. These sub-structures are Point3D objects, which are used to store the position of each agent and provide utility functions for vector operations.

5.6 Parallelization

The program is fully parallelized for shared memory systems using OpenMP. OpenMP is a standard for parallel programming on shared memory systems, which is supported by most compilers. It has been chosen for portability, maintainability, scalability and ease of use as it comes with a simple and flexible interface and is well documented [29]. Up to now, the parallelization does not account for non-uniform memory access (NUMA) based architectures. The simulation part can be compiled to a sequential headless version; the parallel version depends on OpenMP. Parallelization with OpenMP can be switched on or off in the CMake configuration.

As described in section 5.4 and shown by table 5.2 on page 38, the major work is done by the StateMachine class. All the O(N^2) calculate functions listed in the table, which account for more than 90% of the total runtime, are parallelized. They use simple parallel for loops, which are easy to implement and have a low overhead.

O(N) functions are not parallelized, as their share of the total runtime is negligible and parallelization would change the order of the calculations, which would prevent the production of deterministic results (because randomness is used). This is because random number generators always generate the same sequence of numbers when started with the same seed. If the sequence at which agents draw random numbers (for example for a search direction) is changed, the results will be different. Thus, not parallelizing these functions ensures fully deterministic results. For the animation with GUI, two eternal main loops run in parallel, one for the simulation and one for the GUI. This construct is implemented as a parallel sections construct in OpenMP. The thread of the simulation section then spawns the threads for the parallelized calculate functions. Thus multi-level parallelism must be enabled with the OMP_-

MAX_ACTIVE_LEVELS environment variable set to 2. The tool comes with a bash script to set these environment variables.

With this parallelization model, the simulation can utilize all available cores on a shared memory system. Figure 5.2 on page 42 shows the runtime of the simulation part for different numbers of threads. One can see that the runtime decreases with the number of threads, but the speedup is not linear. For eight threads, the speedup is about 3.4, which could be improved, but satisfies our needs.

From the experience of using the tool, parallel efficiency depends a lot on the problem size. For small problems with about 1000 agents, it is not efficient to use more than 4 threads. As can be seen in figure 5.3 on page 43, the number of instructions per cycle (IPC) decreases with the number of threads from 1.9 (1 thread) to 1.5 IPC (8 threads). IPC are dependent on the problem size and the number of threads used; the more threads are used on small problems, the lower the IPC. Thus it has to be kept in mind, that it must not always be the best solution to use all available threads. Usually, there is a saturation point, where the overhead of parallelization is higher than the gain from parallelization. If performance is critical, it is recommended to find the optimal number of threads for the specific problem size. Still, the parallelization is effective and significantly reduces the runtime of the simulation part.

5.7 Vectorization

Vectorization is a technique to speed up the program by processing multiple data elements in parallel. It means that the compiler generates SIMD (Single Instruction, Multiple Data) instructions for the CPU. Multiple data elements are processed in parallel by a single instruction. The data is loaded into vector registers and the instruction is executed on all elements in parallel. This can be achieved by manual vectorization using intrinsics or by automatic vectorization by the compiler. For the developing of the prototype, all major calculations were manually vectorized for testing purposes. But as it has shown, automatic vectorization by the compiler is more efficient. Also automatic vectorization is more portable and maintainable, because many types of SIMD instruction sets are supported by the compiler. The compiler will automatically choose the best instruction set for the target architecture. Thus the program performs best if compiled on the same architecture it will be executed on. The Release build (CMAKE_BUILD_TYPE=Release) of the program sets all necessary flags for the compiler to enable automatic vectorization.

To measure the effect of vectorization, the program was compiled with and without vectorization and the runtime was compared. The compiler used for the tests was G++ 13.2.1 with the optimization flags -O3 -march=native (enables auto-vectorization for the target architecture) or with -O3 -fno-tree-vectorization to suppress compiler optimizations for comparison. Running the test scenario for 1000 iterations (see listing 1, p. 53), the runtime was 52 seconds with vectorization and 126 seconds without vectorization (see table 5.3, p. 44). This is a speedup of 2.4 and shows the importance of vectorization in the sequential version. Vectorization speeds up the parallel version by a factor of 3.8 at 4 threads and 2.9 at 8 threads (see table 5.3, p. 44).

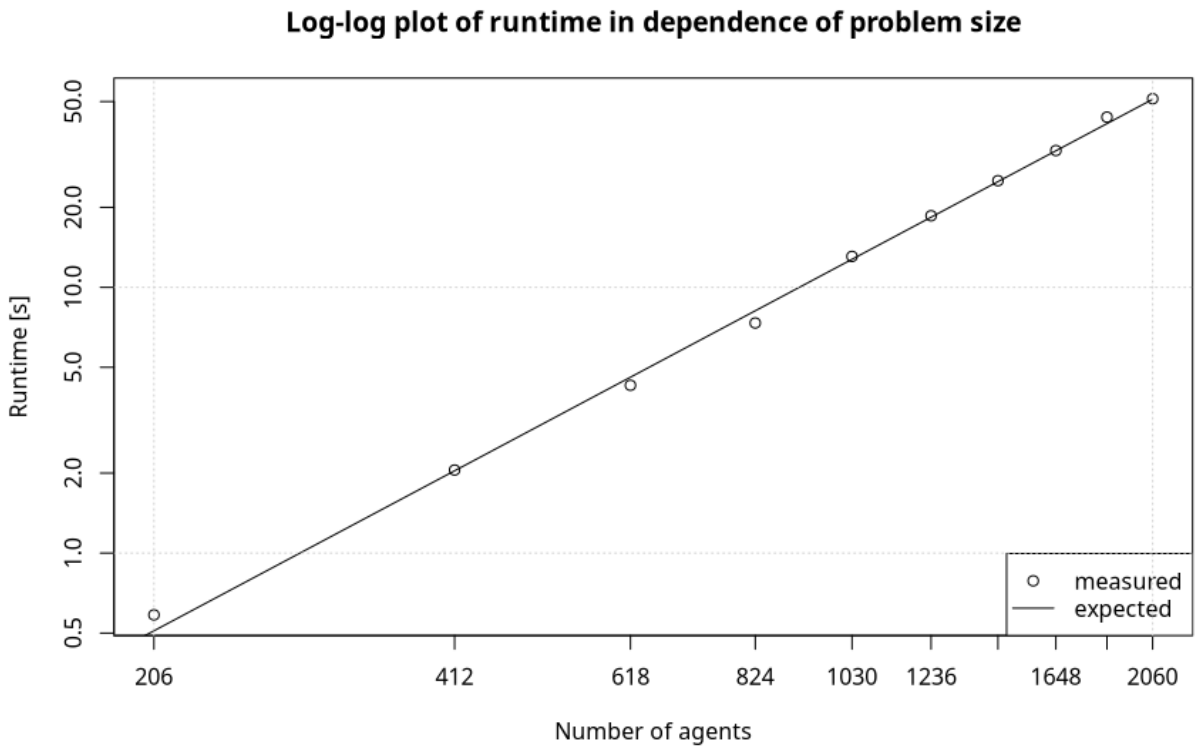

Figure 5.1: Runtime complexity in dependence of the number of agents, using the test config (see listing 1, page 53) scaled linearly to the total number of agents. Runtime for 1000 iterations, using sequential build and seed 1, average of 3 runs per problem size. The line shows the expected runtime based on quadratic interpolation of the largest problem size. The runtime is measured in seconds.

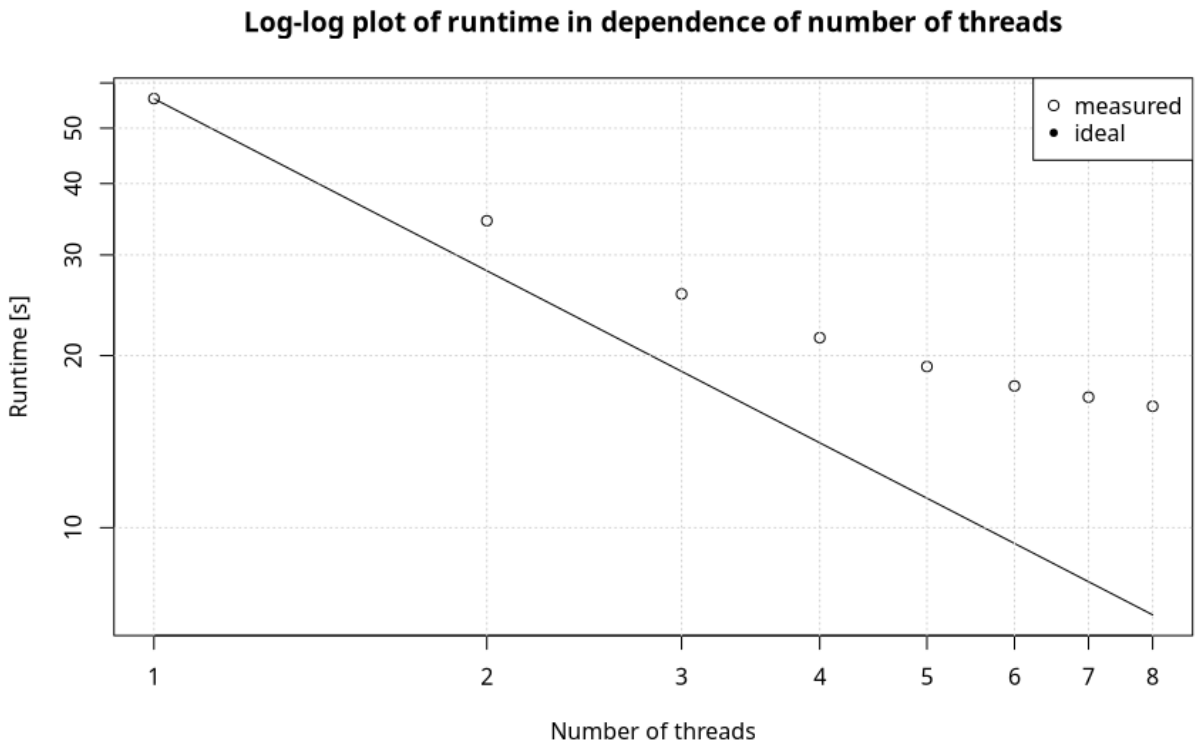

Figure 5.2: Runtime complexity in dependence of the number of parallel threads, using the test config (see listing 1, page 53) with 2060 agents. Runtime for 1000 iterations, average of 3 runs per number of threads. The points shows the expected runtime based on liner interpolation of the runtime for one thread. Circles show the measured runtime. The runtime is measured in seconds.

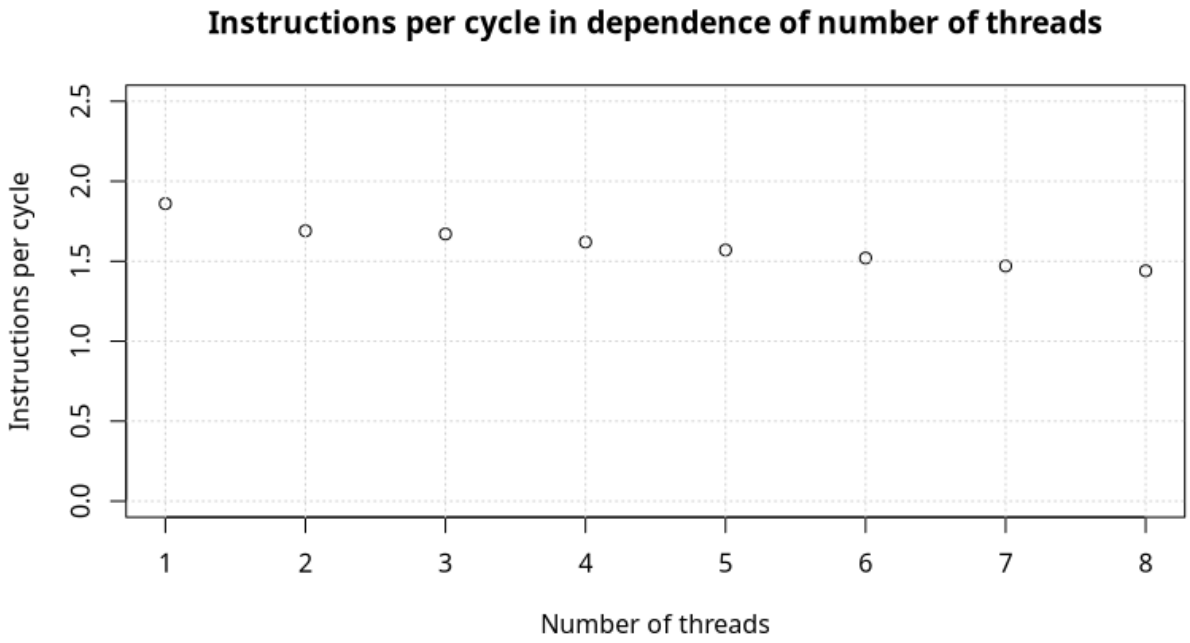

Figure 5.3: Instructions per cycle in dependence of the number of parallel threads, using the test config (see listing 1, page 53) with 2060 agents. Test run for 1000 iterations, average of 3 runs per number of threads.

Table 5.2 on page 38 compares the share of the most important functions in the total runtime with and without vectorization. One can see that for example the function Point3D<float>::distance, which accounts for 14.5% of the total runtime in the non-vectorized version does not appear in the vectorized version, also the functions _sqrt_finite and _f32xsqrtf64 are not present. This is probably because the distance function is inlined in the vectorized version. Inlinining is a technique where the compiler replaces the function call with the actual code of the function. The intrinsic sqrt(=squareroot) functions dissapear, because the compiler replaces them by other instructions, probably SIMD. The calculate audibility/reachability/contact functions do not significantly change their share in the total runtime, but the total runtime is still reduced by a factor of 2.4 to 3.8, depending on the configuration, as shown above. The factor of +-3 hints that the compiler used 128-bit SIMD instructions, which can process 4 single precision floating point numbers in parallel. One can see that the vectorization is very efficient and has a big impact on the runtime.

5.8 Performance

At a problem size of 2000 agents, the fully optimized version processes about 100 timesteps per second (test config, 8 threads). This means it takes about 3 minutes to simulate 6 hours of real time in time steps with $d_t = 1s$ for this number of agents on a standard laptop. According to Dr. Kamran Safi, this is

totally acceptable for the intended use case. Table 5.3 on page 44 shows the speedup of the optimized version compared to the unoptimized version. One can see that the optimized version (run with 8 threads) is about 12 times faster than the plain version.

5.9 Code statistics

Table 5.4 on page 45 shows the statistics of the codebase. The codebase in total comprises more than 130,000 lines of code in more than 200 files. About 100,000 lines of code are written in C++ and 30,000 lines of code are written in C. A marginal fraction of the codebase is written in CMAKE, R, YAML, TOML, XML, Markdown, TypeScript, F#, GLSL and bash. Ten percent of the codebase is written by the author of this thesis, the rest is third-party code. The simulation part comprises about 11,500 lines of self-written code and 17,000 lines of third-party code. The animation part comprises about 4,000 lines of self-written code and nearly 100,000 lines of third-party code. The 3rd party code in the simulation mainly serves the command line and configuration parsing. Third-party code in the animation part mainly serves the rendering and the GUI. A dozen CMakelists.txt files are used to build the codebase. While this is a lot of code, the codebase is well-structured and easy to navigate. As described in chapter 3, the codebase is divided into several modules. Each module is responsible for a specific task. The modules are loosely coupled and communicate via well-defined interfaces. This makes the codebase easy to maintain and extend.

Setup	Runtime (s)	Speedup
Sequential	126.3	1.0
Sequential, vectorized	52.0	2.4
Parallel (4 threads)	43.3	2.9
Parallel (8 threads)	30.7	4.1
Parallel (4 threads), vectorized	11.5	11.0
Parallel (8 threads), vectorized	10.5	12.0

Table 5.3: Runtime and speedup for 1000 iterations of the test config with random seed 1 (see listing 1 on page 53), using different compiler flags and number of threads.

Files	Count (.h)	Lines (.h)	Count (.cpp)	Lines (.cpp)
Simulation	20	1641	17	9788
Animation	17	1993	7	1925
3rd party (simulation)	-	-	4	16829
3rd party (animation)	-	-	3 88	98953
Total	-	-	218	137954

Table 5.4: Statistics of the source code showing the number of files and lines of code for the simulation and animation parts of the project. The 3rd party code is also included, but not split into header and implementation files. Statistics were generated using the cloc tool (v 1.90), counting only pure code lines (excluding blanks and comments).

Summary

The starting point of this project was to create a software which is able to simulate a swarm of bats, to find out which role social information about food sources plays in the formation of bat collectives. Starting with the intention to create an agent based simulation for this specific use case, after all a versatile tool was created, capable of handling most complex scenarios. The tool is able to simulate the behavior of agents in a 3D environment, with the possibility of handling an arbitrary number of agent classes, each with its own behavior and characteristics. To visualize the simulation, an animation framework was developed, which allows the user to see the agents moving in 3D space. The animation further provides a graphical user interface to control the simulation, such as editing parameters, starting and stopping the simulation, changing the speed of the simulation, and changing the camera view. It is fully parallelized for shared memory systems and intended to be used on normal desktop computers. The code is open source and available on GitHub (`https://github.com/de-arl/bsim`) making it accessible to a wide range of users.

During the project it turned out that performance is not the major concern. While initially a program designed for distributed memory systems was planned, it showed that the capabilities of a normal computer are sufficient for the task. Therefore, extainability, maintainability and usability are more important. The final tool is a well balanced compromise between performance and usability, and can be used for a wide range of applications.

Outlook

A suitable tool has been created to serve Dr. Hannah Williams and Dr. Kamran Safi in their research of bat collectives. The first reaction of the researchers was very positive. Dr. Safi plans to hire a student to work on a publication based on the findings with the tool.

First tests have shown that the tool works well and is able to simulate the behavior of bat collectives in a realistic way. According to Dr. Williams, the resulting movement patterns are very similar to the ones observed in the field. The first tests have also shown, that social information about food sources and the search parameters have a significant impact on the behavior of the bats, the formation of collectives, and the efficiency of the foraging process.

Thanks to the design optimization, the tool became highly flexible and can be used for a wide range of applications. It allows for interactive exploration of various parameters and scenarios. Systematic experiments can be conducted to investigate the influence of different factors on the behavior of the agents. The parameter space can be explored in a structured way, and the application programming interface (API) allows for integration with machine learning or artificial intelligence tools to find optimal parameter settings.

The tool is user-friendly and easy to use. It is designed to be flexible and can be used in a variety of scenarios. By not limiting the number of agent classes, complex systems can be modeled. For example, it could be used to simulate the behavior of fish schools, bird flocks, or human crowds. Besides predator-prey interactions, other types of interactions can be modeled as well. For example the spread of diseases, the diffusion of information, or the formation of social networks. The framework, which works like a clockwork, and the agent behaviour algorithms are separated; this ensures that it cannot be broken by changing the agent behaviour and the agent behaviour can easily be changed without changing the framework.

It includes memory of the system state, which is available to the agents; to allow for the implementation of even more complex behaviours. Agents could for example base their decisions on derivatives, of any order; of state variables over time. Even most complex scenarios are thinkable, where agents mate, reproduce, and die, and where the offspring inherits the memory of the parents - to model evolution.

Many features could be added to the tool to make it even more versatile. This includes for example real-time data processing and visualization. If performance needs to be improved, the parallel efficiency could be further optimized. Also the tool could be parallelized for distributed memory systems, if very large models should be simulated.

After all, the software developed in this project has to be seen as a template for general agent-based modelling and simulation. With little effort, it can be adapted to many different use cases. The framework breathes life into the simulation and the agents, and the user only has to tell the agents what to do. Still, this requires to dive into the code and understand the underlying principles. Even though this might be a bit challenging at first, it should still cost much less time than developing a new tool from scratch. It should be noted that the tool is still in development and that the author is open to suggestions for improvements. If you are interested in using the tool for your own research too; please feel free to contact the author.

Danksagung

Ich möchte mich ganz herzlich bei meiner Familie bedanken, ohne die mein Studium nicht möglich gewesen wäre. Mein weiterer Dank gilt besonders auch Frau Professor Kuttler, die mir den Quereinstieg in die Welt der Mathematik, Informatik und des Hochleistungsrechnens mit modernen Supercomputern ermöglicht hat. Ich bedanke mich auch ganz herzlich bei Herrn Professor Laure, der mir die Möglichkeit gegeben hat, meine Masterarbeit an seinem Institut zu schreiben und mir mit Dr. Rampp einen ausgezeichneten Betreuer zur Seite gestellt hat. Dr. Rampp hat mir den nötigen Freiraum gegeben, um eigene Ideen zu entwickeln und umzusetzen, und er hat mich immer wieder in die richtige Richtung gelenkt und stets mit Rat und Tat unterstützt. Außerdem hat er den Kontakt zu Dr. Williams und Dr. Safi hergestellt, bei denen ich mich auch ganz herzlich für die gute Zusammenarbeit bedanken möchte und hoffe, daß sie mit Hilfe der gemeinsam entwickelten Software wertvolle Erkenntnisse gewinnen können.

München, 1. Juni 2024, Andreas R. Laible

Bibliography

[1] Technische Universität München. *Studiengangsdokumentation Masterstudiengang Computational Science and Engineering*. 2021.

[2] Gauvain SAUCY. “Bat swarming: reviewed definition, overestimated functions and new research directions”. In: *preprint* (2019).

[3] *Wikipedia: The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Agent-based_model`. Accessed: 2024-05-15.

[4] Volker Grimm and Steven F Railsback. “Agent-based models in ecology: patterns and alternative theories of adaptive behaviour”. In: *Agent-based computational modelling: Applications in demography, social, economic and environmental sciences* (2006), pp. 139–152.

[5] Stefano Mariani and Andrea Omicini. *Special issue “multi-agent systems”*. 2020.

[6] Alex Smajgl and Olivier Barreteau. “Empiricism and agent-based modelling”. In: *Empirical Agent-Based Modelling-Challenges and Solutions: Volume 1, The Characterisation and Parameterisation of Empirical Agent-Based Models*. Springer, 2013, pp. 1–26.

[7] Seth Tisue and Uri Wilensky. “Netlogo: A simple environment for modeling complexity”. In: *International conference on complex systems*. Vol. 21. Citeseer. 2004, pp. 16–21.

[8] Sean Luke et al. “Mason: A multiagent simulation environment”. In: *Simulation* 81.7 (2005), pp. 517–527.

[9] Michael J North, Nicholson T Collier, and Jerry R Vos. “Experiences creating three implementations of the repast agent modeling toolkit”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 16.1 (2006), pp. 1–25.

[10] Alexis Drogoul et al. “Gama: A spatially explicit, multi-level, agent-based modeling and simulation platform”. In: *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer. 2013, pp. 271–274.

[11] Michael J North and Charles M Macal. “Agent-Based Modeling and Computer Languages”. In: *Complex Social and Behavioral Systems: Game Theory and Agent-Based Models* (2020), pp. 865–889.

[12] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[13] *Wikipedia: The Free Encyclopedia*. https://en.wikipedia.org/wiki/Unix_philosophy. Accessed: 2024-05-15.

[14] *Wikipedia: The Free Encyclopedia*. https://en.wikipedia.org/wiki/KISS_principle. Accessed: 2024-05-15.

[15] Mark Gillard. *toml++: Header-only TOML config file parser and serializer for C++17.* https://marzer.github.io/tomlplusplus/.

[16] Jarryd Berg. *cxxopts: Lightweight C++ command line option parser*. https://github.com/jarro2783/cxxopts.

[17] Michael Wichura. *C++ library which computes the inverse of the Normal Cumulative Density Function, by Michael Wichura.* https://people.math.sc.edu/Burkardt/cpp_src/asa241/asa241.html.

[18] *GLFW: Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop.* https://www.glfw.org/.

[19] *OpenGL Mathematics (GLM)*. https://github.com/g-truc/glm.

[20] Omar Cornut. *Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies*. https://github.com/ocornut/imgui.

[21] Stephane Cuillerdier. *ImGuiFileDialog: Full featured File Dialog for ImGui*. https://github.com/aiekick/ImGuiFileDialog.

[22] *WebP: Modern image format that provides superior lossless and lossy compression for images on the web.* https://developers.google.com/speed/webp.

[23] GitHub. *GitHub: Copilot*. https://copilot.github.com/. Accessed: 2024-05-15.

[24] *CMake: Cross-platform family of tools designed to build, test and package software.* https://cmake.org/.

[25] *Performance Application Programming Interface (PAPI): Portable interface to hardware performance counters on modern processors.* https://icl.utk.edu/papi/.

[26] Gandhimohan M Viswanathan et al. “Lévy flights in random searches”. In: *Physica A: Statistical Mechanics and its Applications* 282.1-2 (2000), pp. 1–12.

[27] *OpenGL: The Industry’s Foundation for High Performance Graphics*. https://www.opengl.org/.

[28] *glad: Multi-Language Vulkan/GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs.* https://github.com/Dav1dde/glad.

[29] *OpenMP: API for parallel programming in C/C++*. https://www.openmp.org/.

[30] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

Appendix

Test configuration

Listing 1: Example configuration file for five classes used for testing energy balance

```
# Simulation parameters

[simulation]
  simulated_time = 86400
  dt = 1
  seed = 0
  num_runs = 1
  memory_size = 10

[world]
  worldsize = [1600, 1000, 100]

[agent_class_params]
  classes = ["male_bats", "female_bats", "owls", "insects", "plants"]

[male_bats]
  id = 0
  friends = [  "male_bats" ]
  enemies = [  "owls" ]
  food_sources = [  "insects" ]
  strategy = 3
  status = 1
  num_agents = 500
  audibility = 300
  mu_energy = 100000
  sigma_energy = 1000
```

```
  energy_uptake_rate = 10
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.3
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.8
  movespeed = 12
  energy_consumption_per_time = 0.1
  energy_consumption_per_distance = 0.1
  size = 0.1
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 0
  color = [ 0.500000, 0.750000, 1.000000 ]
  pointsize = 3

[female_bats]
  id = 1
  friends = [  "female_bats" ]
  enemies = [  "owls" ]
  food_sources = [  "insects" ]
  strategy = 3
  status = 1
  num_agents = 500
  audibility = 300
  mu_energy = 100000
  sigma_energy = 1000
  energy_uptake_rate = 10
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.3
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.8
  movespeed = 10
  energy_consumption_per_time = 0.1
  energy_consumption_per_distance = 0.1
```

```
  size = 0.1
  start_position = [ 0.000000, 1.000000, 0.000000 ]
  random_start_positions = 0
  color = [ 1.000000, 0.750000, 0.800000 ]
  pointsize = 3

[owls]
  id = 2
  friends = [  "owls" ]
  enemies = [  "none" ]
  food_sources = [  "male_bats",  "female_bats" ]
  strategy = 1
  status = 1
  num_agents = 10
  audibility = 50
  mu_energy = 1e+06
  sigma_energy = 1000
  energy_uptake_rate = 1000
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.5
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 15
  energy_consumption_per_time = 1
  energy_consumption_per_distance = 1
  size = 1
  start_position = [ 0.000000, 0.000000, 2.000000 ]
  random_start_positions = 1
  color = [ 1.000000, 0.000000, 0.000000 ]
  pointsize = 5

[insects]
  id = 3
  friends = [  "insects" ]
  enemies = [  "male_bats",  "female_bats" ]
  food_sources = [  "plants" ]
```

```
  strategy = 6
  status = 1
  num_agents = 1000
  audibility = 100
  mu_energy = 10
  sigma_energy = 1
  energy_uptake_rate = 0.1
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.3
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.5
  movespeed = 2
  energy_consumption_per_time = 0.001
  energy_consumption_per_distance = 0.001
  size = 10
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 1.000000, 0.987342, 0.000000 ]
  pointsize = 2

[plants]
  id = 4
  friends = [  "none" ]
  enemies = [  "insects" ]
  food_sources = [  "none" ]
  strategy = 0
  status = 1
  num_agents = 50
  audibility = 500
  mu_energy = 1000
  sigma_energy = 200
  energy_uptake_rate = 0
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 1
```

```
  movespeed_approach_friends = 1
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 2
  energy_consumption_per_time = -1
  energy_consumption_per_distance = 0
  size = 50
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 0.000000, 0.500000, 0.000000 ]
  pointsize = 10
```

Listing 2: Configuration for the comparison of random walk types (mixed case)

```
[simulation]
  simulated_time = 21600
  dt = 1
  seed = 1
  num_runs = 1
  memory_size = 10

[world]
  worldsize = [10000, 10000, 1]

[agent_class_params]
  classes = ["male_bats"]

[male_bats]
  id = 0
  friends = [  "male_bats" ]
  enemies = [  "owls" ]
  food_sources = [  "insects" ]
  strategy = 2
  status = 1
  num_agents = 1
  audibility = 100
  mu_energy = 1000
  sigma_energy = 0
  energy_uptake_rate = 1
  mu_levy = 0
```

```
  c_levy = 0.5
  brownian_search_duration = 120
  movespeed_search = 1
  movespeed_approach_friends = 1
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 12
  energy_consumption_per_time = 0
  energy_consumption_per_distance = 0
  size = 0.1
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 0
  color = [ 0.500000, 0.750000, 1.000000 ]
  pointsize = 10
```

Listing 3: Example configuration file for a simple agent behaviour algorithm correctness test (description see first example section 5.3

```
[simulation]
  simulated_time = 2
  dt = 1
  seed = 1
  num_runs = 1
  memory_size = 10

[world]
  worldsize = [50, 50, 20]

[agent_class_params]
  classes = ["male_bats", "insect_swarms"]

[male_bats]
  id = 0
  friends = [  "male_bats" ]
  enemies = [  "owls" ]
  food_sources = [  "insect_swarms" ]
  strategy = 2
  status = 1
  num_agents = 100
  audibility = 100
```

```
  mu_energy = 1000
  sigma_energy = 0
  energy_uptake_rate = 1
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 120
  movespeed_search = 1
  movespeed_approach_friends = 1
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 10000
  energy_consumption_per_time = 10
  energy_consumption_per_distance = 1
  size = 0.1
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 0.500000, 0.750000, 1.000000 ]
  pointsize = 10

[insect_swarms]
  id = 1
  friends = [  "none" ]
  enemies = [  "male_bats" ]
  food_sources = [  "none" ]
  strategy = 0
  status = 1
  num_agents = 1
  audibility = 10000
  mu_energy = 1000
  sigma_energy = 0
  energy_uptake_rate = 0
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 1
  movespeed_approach_friends = 1
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 0
```

```
  energy_consumption_per_time = 1
  energy_consumption_per_distance = 0
  size = 10
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 0.000000, 0.500000, 0.000000 ]
  pointsize = 10
```

Listing 4: Example configuration file for a simple agent behaviour algorithm correctness test (description see second example section 4.3)

```
[simulation]
  simulated_time = 2
  dt = 1
  seed = 1
  num_runs = 1
  memory_size = 10

[world]
  worldsize = [50, 50, 20]

[agent_class_params]
  classes = ["male_bats", "owls"]

[male_bats]
  id = 0
  friends = [  "male_bats" ]
  enemies = [  "owls" ]
  food_sources = [  "none" ]
  strategy = 3
  status = 1
  num_agents = 1
  audibility = 10000
  mu_energy = 100000
  sigma_energy = 1000
  energy_uptake_rate = 10
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.3
```

```
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.8
  movespeed = 12
  energy_consumption_per_time = 0.1
  energy_consumption_per_distance = 0.1
  size = 0.1
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 0.500000, 0.750000, 1.000000 ]
  pointsize = 3

[owls]
  id = 1
  friends = [  "none" ]
  enemies = [  "none" ]
  food_sources = [  "male_bats" ]
  strategy = 1
  status = 1
  num_agents = 1
  audibility = 10000
  mu_energy = 1e+06
  sigma_energy = 1000
  energy_uptake_rate = 1000
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 10
  movespeed_search = 0.5
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 1
  movespeed = 10000
  energy_consumption_per_time = 1
  energy_consumption_per_distance = 1
  size = 1
  start_position = [ 0.000000, 0.000000, 2.000000 ]
  random_start_positions = 1
  color = [ 1.000000, 0.000000, 0.000000 ]
  pointsize = 5
```

Listing 5: Configuration for the comparison of search strategies (mixed strategy)

```
[simulation]
  simulated_time = 21600
  dt = 1
  seed = 1
  num_runs = 1
  memory_size = 10

[world]
  worldsize = [10000, 10000, 100]

[agent_class_params]
  classes = ["male_bats", "insect_swarms"]

[male_bats]
  id = 0
  friends = [  "male_bats" ]
  enemies = [  "none" ]
  food_sources = [  "insect_swarms" ]
  strategy = 2
  status = 1
  num_agents = 1000
  audibility = 300
  mu_energy = 1
  sigma_energy = 0
  energy_uptake_rate = 1
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 120
  movespeed_search = 0.3
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.8
  movespeed = 12
  energy_consumption_per_time = 0
  energy_consumption_per_distance = 0
  size = 0.1
  start_position = [ 0.000000, 0.000000, 0.000000 ]
```

```
  random_start_positions = 0
  color = [ 0.500000, 0.750000, 1.000000 ]
  pointsize = 3

[insect_swarms]
  id = 1
  friends = [  "none" ]
  enemies = [  "none" ]
  food_sources = [  "none" ]
  strategy = 8
  status = 1
  num_agents = 42
  audibility = 200
  mu_energy = 50000
  sigma_energy = 0
  energy_uptake_rate = 0
  mu_levy = 0
  c_levy = 0.5
  brownian_search_duration = 0
  movespeed_search = 0.3
  movespeed_approach_friends = 0.5
  movespeed_avoid_enemies = 1
  movespeed_approach_food = 0.5
  movespeed = 5
  energy_consumption_per_time = 0
  energy_consumption_per_distance = 0
  size = 10
  start_position = [ 0.000000, 0.000000, 0.000000 ]
  random_start_positions = 1
  color = [ 1.000000, 0.987342, 0.000000 ]
  pointsize = 10
```

Source code

The source code is available at `dummy`.