THE IMPACT OF ERROR-TOLERANCE USING CUSTOMIZED DIJKSTRA ON LARGE SCALE PARALLEL CROWD SIMULATIONS

LUUK STERKE

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITI MALAYA
KUALA LUMPUR

2022

THE IMPACT OF ERROR-TOLERANCE USING CUSTOMIZED DIJKSTRA ON LARGE SCALE PARALLEL CROWD SIMULATIONS

LUUK STERKE

DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITI MALAYA
KUALA LUMPUR

2022

UNIVERSITI MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Luuk Sterke

Registration/Matric No.: WOA190011/17202595/1

Name of Degree: Master of Computer Science (Applied Computing)

Title of Dissertation ("this Work"): The Impact of Error-Tolerance Using Customized Dijkstra on Large Scale Parallel Crowd Simulations

Field of Study: Parallel Architecture and Design

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the Universiti Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date: 24 January 2022

Subscribed and solemnly declared before,

Witness's Signature Date: 29.01.22

Name:
Designation:

THE IMPACT OF ERROR-TOLERANCE USING CUSTOMIZED DIJKSTRA ON LARGE SCALE PARALLEL CROWD SIMULATIONS

ABSTRACT

This Dissertation aims at exploring and quantifying the errors that occur in parallel crowd simulation when not implementing measures to prevent these errors. Consequently, the simulation should run faster, as less code needs to be executed. A fully functional crowd simulator has been developed and various features for measuring accuracy and performance have been implemented. Part of the research was identifying the right metrics for measuring these quantities. It turns out that a heat map in combination with waiting times, walking times and local flow measuring entities together create a bijection between this data and the simulation producing it. The impact of allowing errors on the simulator's accuracy can be measured quite well by comparing these statistics. The performance is then measured using different internal stopwatches keeping track of the time needed to simulate a fixed amount of agents. From many simulation environments and settings, it becomes clear that significant speedups can be achieved using the proposed techniques. These speedups of up to 15% are achieved at the expense of the simulator's accuracy, where flow and waiting times are off by single-digit percentages. Heat maps deviate more percentage-wise because they are more sensitive to small changes. In one case, error tolerance was not faster and less accurate since error tolerance was not applied on the scale of all agents but on reducing internal overhead time. Allowing errors to examine many simulations quickly and then perform precise simulations on just the promising ones are recommended. Whether the errors observed are large or small is not conclusive as that depends on the simulated event and because such conclusions are outside of this dissertation's scope.

Keywords: Error Tolerance, Crowd Simulation, Parallelization.

KESAN TOLERANSI SESAR DENGAN PENYESUAIAN DIJKSTRA PADA SIMULASI KERUMUNAN SELARI BERSKALA BESAR

ABSTRAK

Disertasi ini meneroka dan mengukur sesaran yang berlaku dalam simulasi kerumunan selari apabila langkah-langkah untuk mencegah sesaran tidak dilaksanakan. Sehubungan itu, simulasi akan berlaku lebih cepat kerana kurangnya kod untuk dilaksana. Disertasi ini menyumbang satu simulator kerumunan dengan metrik khas untuk mengukur ketepatan dan prestasi simulasi. Metrik tersebut mengambilkira keputusan peta haba, masa menunggu, masa berjalan, dan entiti pengukur aliran tempatan dan menjana bijeksi antara data dan simulasi yang menghasilkan data tersebut. Kesan membiarkan sesaran terhadap ketepatan simulator boleh diukur dengan baik dengan membandingkan statistik ini. Prestasi kemudian diukur menggunakan jam randik dalaman yang mengawasi masa yang diperlukan untuk mensimulasikan jumlah agen yang tetap. Berdasarkan pelbagai tetapan persekitaran dan simulasi, didapati kelajuan simulasi sehingga 15% dapat dicapai dengan teknik yang dicadangkan. Walaupun kelajuan ini mengorbankan ketepatan simulator, peratusan aliran dan masa menunggu tersesar masih di bawah angka satu digit. Peratusan sesaran peta haba adalah lebih tepat kerana lebih sensitif terhadap perubahan kecil. Dalam satu kes, toleransi sesaran diamati tidak lebih pantas atau tepat apabila digunakan untuk mengurangkan masa overhed dalaman dan bukan pada skala semua agen. Untuk itu, dicadangkan sesaran dibiarkan supaya simulasi boleh diperiksa dengan cepat, dan simulasi tepat hanya dilakukan untuk kes terperinci yang berpotensi. Tetapan simulasi dilihat mempengaruhi sesaran, tetapi samada sesaran tersebut besar atau kecil tidak dapat dipastikan kerana diluar skop disertasi ini.

Kata kunci: Toleransi Sesaran, Simulasi Kerumunan, Keselarian.

ACKNOWLEDGEMENTS

I would like to extend my sincere thanks to Dr Zati Hakim Azizul Hasan for her help with making this dissertation possible. I am also very grateful to my friends and family, who supported me unconditionally during the writing of this dissertation. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

CPU : Central Processing Unit

GPU : Graphics Processing Unit

CUDA : Compute Unified Device Architecture

HUD : Heads-Up Display

RAM : Random-Access Memory

NVRAM : Non-Volatile Random-Access Memory

GPS : Global Positioning System

UML : Unified Modeling Language

fps : frames per second

LIST OF APPENDICES

PREFACE

Crowd simulation is a vibrant and widely used field within computer science. While many aspects of crowd simulation are well-known and researched, the field is constantly changing and looking for ways to make simulations faster and more realistic. This dissertation aims to determine how a simulation outcome is changed, if errors are not prevented from speeding up the simulation. Breaking point crowd safety due to errors is not part of the research, the focus of the research is the trade-off between simulation accuracy and speed. It was written as a part of my Master of Computer Science degree at Universiti Malaya.

The inspiration for this project came from my bachelor's degree in computer science, where I had the privilege of working on a crowd simulator in the industry and saw how the complex simulations affected the potential number of agents that could be simulated in real-time. This number could be more significant if the simulator would run faster. The dissertation starts with an introduction explaining the scope and goal of this research further, followed by a review of the literature related to this topic. From there onward, the research I carried out is explained in detail. I hope you will enjoy reading this work.

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

Crowd simulation has been a topic of interest for many people such as event organizers and urban planners for many years now. It offers a cheap and safe way to test if the layout of any building or event sufficiently accommodates the intended flow of people through it and shows potential shortcomings in an evacuation situation.

There are many examples where crowd simulation played a role in real projects, but a recent one we can highlight here is the start of the Tour de France in 2015 in the city of Utrecht. Utrecht is a city with a centre that is hard to reach by car and therefore many people who wanted to come and watch the tour came using public transportation. Public transportation creates a large flow of people from the central train station towards the spectator areas. A crowd simulation was used to find the potential choke points to create a safe way for so many people to walk this route. A choke point is a small area in the (simulation) environment where the flow of people is too large for the area to let everyone through, thus creating a crowd where people have to wait and get stuck.

Simulating such an event is not something new; the challenge faces by developers of these simulations is therefore not to simulate a crowd in a given environment. The challenge is to simulate the crowd in a very realistic and above all fast way. For the opening day of the Tour the France in 2015, a hundred thousand spectators would not be an unrealistic number. A computer that has to run such a simulation needs to be very fast, but no matter how fast it is, simulating such large amounts of people will take time. Therefore, many developers and researchers are concerned with the objective of speeding up such simulators using a

wide range of techniques. The objective forms the first inspiration for this dissertation as well. The goal here is to evaluate a somewhat controversial technique to speed up large scale crowd simulations.

Later in this section, this controversial aspect is explained. First, the problem and proposed solution themselves will be examined. Figure 1.1 shows ten frames in an evacuation simulation. At frame 6, we start seeing the first problem; not all agents fit through the door. At frames beyond frame 6, as the agents try to follow the shortest path to the goal area, even fewer of them fit through the narrow hall than through the door. When planning this floor, adjustments can be made to accommodate the expected amount of people. Possible solutions are to limit the flow of agents spawning in the spawn area if possible, or sending the agents to the goal along a longer, wider path.

Now let's say we want to simulate something similar, but with more agents at the same time, say 10,000. There are plenty of events with such numbers of agents, so simulating the movements of so many people is still very relevant. Of course, the people should move somewhat naturally and generate data for individual people and the crowd as a whole. Furthermore, it saves time to do multiple such simulations simultaneously, perhaps with small environmental changes or various numbers of agents. Suddenly there are millions of calculations for each frame, even for a non-trivial simulator. The traffic would result in a simulation or set of simulations that run at less than one frame per second (fps).

The first question we must ask ourselves is, is this a problem? After all, the simulator can run continuously overnight, generating data and potentially all the frames. The next day, all answers to questions about the simulation are ready. There are two fundamental

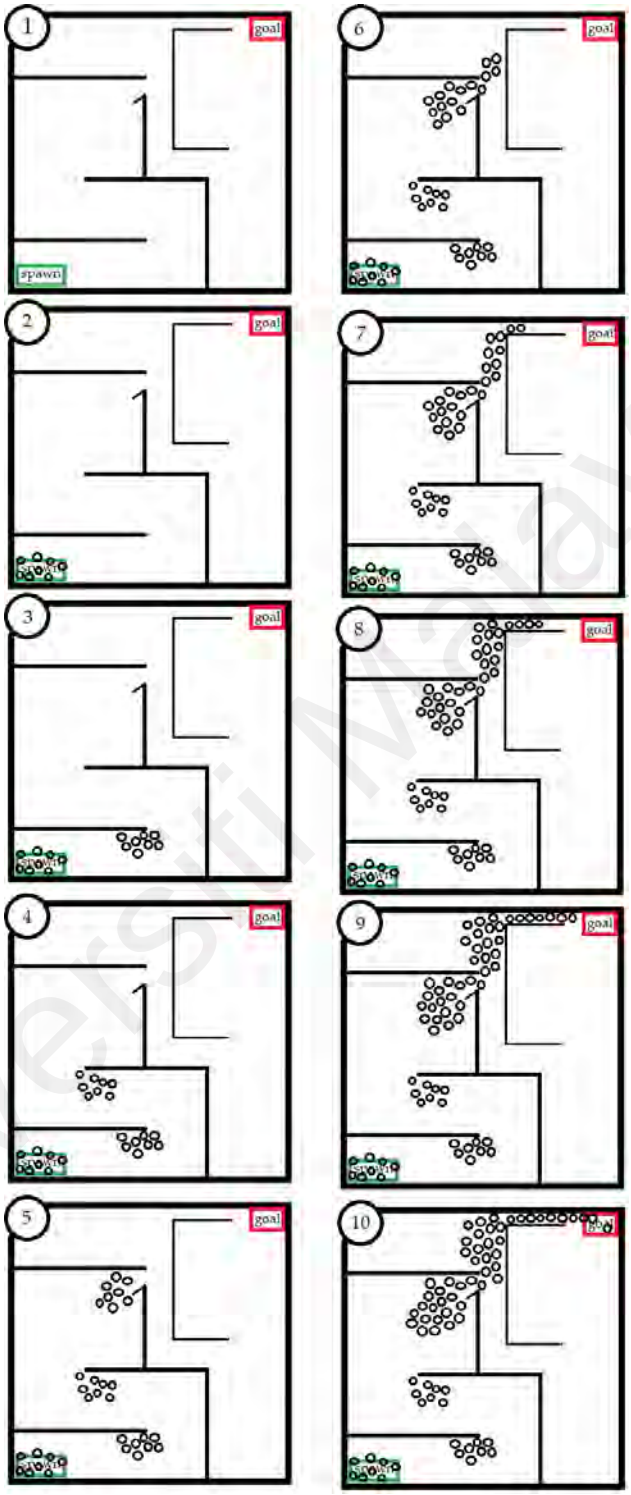

Figure 1.1: Here is an example of a simulation with a goal and a spawn. Every step, the spawn spawns eight agents with the same goal, e.g. a lift that delivers people to the floor every time.

reasons why this is far from good enough. First, when planning an event, many small changes and tweaks will be made over time. The organizers may need to simulate different settings hundreds of times for long enough to generate useful data. That means there is no time to let the simulator run for many hours and await the results; by the time all the simulations would be done, the event will have passed a year ago. A second reason is that the simulator should be able to simulate an area in the case of an emergency. With sufficient knowledge of the area and estimated amount of people inside, a simulation can show where the streams of people will go and help or instructions can be sent accordingly. In this case, the faster the simulation generates the data, the better. If this process to produce results takes an hour, it is too late for those results to be used.

The question that follows is, can we do better? Many researchers have asked this question regarding crowd simulation. From complicated mathematical models to machine learning techniques (Peymanfard & Mozayani, 2018) (Weiss et al., 2019) (Sillapaphiromsuk & Kanongchaiyos, 2019) (Q. Wang et al., 2019), many ways to both speed up and increase accuracy have been tried with varying degrees of success. One traditional way to speed up crowd simulations and many computer programs is to put more than one logical processing unit core to work. A computer's CPU varies from 16 logical cores on a high-end processor such as an *Intel i9* CPU to 4 logical cores on a lower-end *Intel i3* series CPU. That already would cut calculation time by 75 to 93.75 per cent if we could perfectly distribute the work over all logical cores. But it gets better; with a good GPU such as an *GeForce RTX 2070*, which has 2,304 CUDA cores, we could speed up the most resource-intensive part of the simulation more than 1,000-fold in an ideal situation. Such a boost is the speed up needed to go from an overnight pre-calculated simulation to a real-time simulation. Since this is not a new concept, other researchers have explored this option and found success in doing

so (Yilmaz, 2010) (Chen et al., 2013).

There is one major issue with parallelization of simulations of many kinds, however. When two processes read and write to the same part of the memory, they may create conflicting results. Figure 1.2 explains such a case. In short, suppose there are two agents with positions $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ each with radius r without overlapping. They try to walk to a goal $G = (x, y)$ which is in roughly the same direction for both agents. They might both try to walk to intermediate point $P_3 = (x_3, y_3)$. As two parallel processes control them, they can both check position P_3 for obstructions such as walls and other agents simultaneously. If so, neither process will see an obstruction and move towards P_3. Then both processes write the new position for these agents, P'_1 and P'_2 to the memory. Now the situation can occur that in the new situation $\sqrt{(x'_1 - x'_2)^2 + (y'_1 - y'_2)^2} < 2r$, meaning the two agents overlap. Although parallel simulation could help all the proposed solutions for making the simulations more realistic, produce more accurate data and run faster, this issue cannot be ignored. A possible solution is to lock data that is being used by a process. For example all agents within a $2(r + s)$, where s is the agent speed, range gets locked so other processes cannot read or write to those parts of the memory. Another solution could be to smart-schedule the processes so that parallel processes calculate agents who are far away from each other.

All these solutions to the parallelization problem cost extra processing power. For example, in the worst-case scenario, locking could result in many parallel processes just waiting for each other and not speeding the simulation up at all. Since agents are always on the move, the scheduling will also require many extra calculations at each step if a waterproof schedule is possible. Therefore, we cannot expect all these extra calculations to

have a negligible impact.

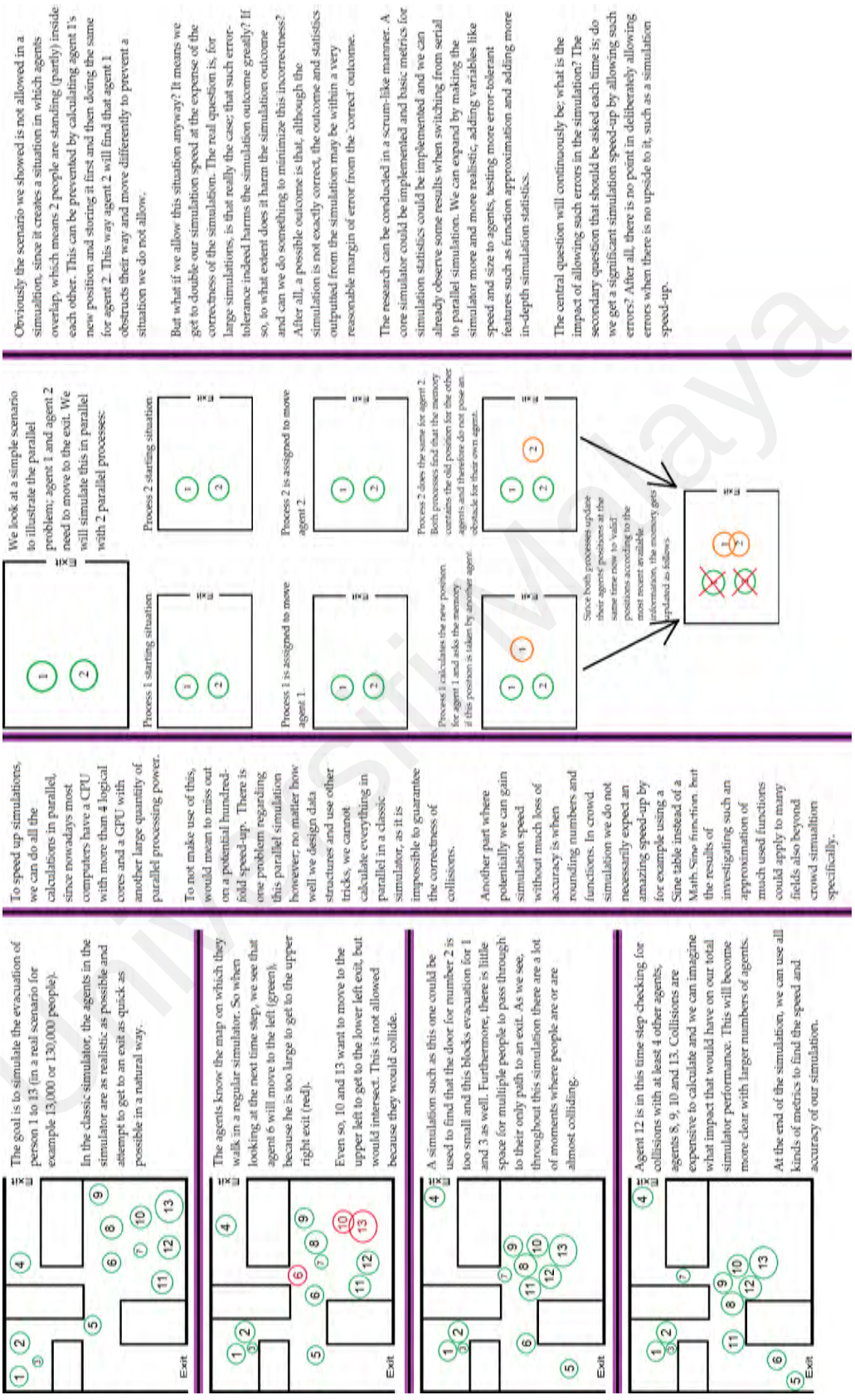

Figure 1.2: An explanation of the problems with uncontrolled parallel simulations

So what if we don't try to solve these issues? What if we simply let the simulator run massively parallel and read and write to any part of the memory it needs? All the errors described earlier could occur, but we do not know if that will also result in significantly inaccurate simulation results. After all, the simulator generally still does not allow collisions and simulates the agents using the same rules as in a regular simulator. There are three possible outcomes in terms of accuracy.

1. No matter the implementation, the errors result in a massively different outcome compared to the non-error-tolerant approach.
2. General trends in the outcome such as jams and flows stay intact, but the numbers and more specific statistics vary significantly.
3. The simulation outcomes stay largely the same; within a reasonable margin of error, we see the same statistics in the error-tolerant approach as we see in the original one.

Besides finding out how error tolerance impacts the simulation accuracy, it is crucial to investigate the potential speed-up. Both accuracy and speed are benchmarked on simulations with varying amounts of cores. Suppose the speed-up is marginal or the simulation results in outcome one for each test. In that case, the conclusion will be that error tolerance in parallel crowd simulation is not of added value. As parallelization is a way to speed up many programs beyond crowd simulation, the findings in this dissertation can also be of value to other fields.

1.2 Problem Statement

In today's world, computer programs cannot be fast enough. Whether the application is crowd simulation, video editing or any other field; speed is key. As processors are reaching a limit to their maximal clock speed (Mattsson, 2014), one way to speed up computers and therefore the applications run on them is to increase the amount of cores within a

CPU. As of January 2021, top processors such as *AMD Ryzen 9 5950X* and *Intel Core i9-10900K* have 32 and 20 parallel threads respectively. Such processing power implies a potential speed-up of a (crowd) simulation by a factor of 20. However, there is some overhead in distributing the tasks over all the threads. Also, the prevention and taking care of parallelization errors consume a significant portion of the CPU's processing power. An unorthodox way of further increasing a crowd simulation would be to use the processing power of error prevention for the simulation instead. The approach means these errors are not prevented and therefore the application is likely to produce inaccurate results at some point during the simulation. Depending on the application, this can be a problem. An example of an application where this can lead to a terrible result is Conway's Game of Life (Gardner, 1970), as one wrong cell could change the evolution of the colony as a whole.As a result, error tolerance in a parallel Game of Life generally produces significantly wrong results. Now we want to investigate whether such an approach, where we do not account for parallelization errors, would lead to significantly wrong results in a crowd simulation.

1.3 Research Questions

The following research questions form the start of this research project:

1. Which metrics and statistics can show the differences between an error-tolerant parallel simulation and a non-parallel simulation?
2. Does not having the overhead of preventing parallelization errors result in a significant speed-up?
3. For each metric, how do the simulation outcomes differ between the error-tolerant and the error-preventive methods?

1.4 Objectives of the Study

As a result of conducting this research, a set of three goals are achieved. These goals are as follows:

1. To identify metrics that define the course of a simulation.
2. To build a complete crowd simulator to implement serial, error-tolerant parallel and error-preventive parallel simulation code.
3. To evaluate the simulator performance.

1.5 Scope of the Study

Various tests will determine whether error-tolerance in parallel crowd simulation results in a faster simulation, while keeping the simulation relatively accurate. The simulator will be able to simulate agents of various sizes and speeds within the same simulation. Two different types of pathfinding are implemented; global pathfinding to simulate a crowd with complete knowledge of the environment and compass pathfinding to simulate agents who only know the direction of their goal. The simulator will run serial code or CPU-run parallel code. The test cases will be manually drawn, multiple-spawn, multiple-goal fictional environments.

More complex agents, such as agents changing their behaviours during the route or prefer a specific type of route over the optimal route, are outside the study's scope. During simulations, agents will also only use the pathfinding as described above. Although we find many different pathfinding algorithms, it is not in the scope of the study to test the research questions for each possible pathfinding algorithm. Lastly, real environments obtained by scanning maps or other means of recreating them are not tested in our simulator. Real maps serve as an inspiration for the simulation environments, but to create environments

that can really challenge our hypotheses, they need to include specific elements in specific places. Examples include doors or narrow halls in key positions and strategically places walls.

1.6 Significance of the Study

In some applications, including crowd simulation, parallel computing is necessary to get the simulation done within a reasonable amount of time. $O(\log n)$ or more time will get lost to prevent common parallelization errors, plan and distribute the work over all processing units correctly. This time could include time for organizing all agents in a tree-type structure. Figure 1.3 shows a simple implementation of such a tree, based on position, that can be produced in $O(n)$ time. The figure shows how a tree can be constructed to simulate a number of agents in parallel on two processing units (PU). The concept can be extended to any number of PUs. As the goal of using multiple PUs is to speed up the simulation, an analysis of time complexity of constructing such a tree is necessary. As it takes $O(n)$ time to construct the tree in Figure 1.3, this tree-structure could have impact on the simulator's performance. Any approach, including locking of data, will similarly result in some processing time $O(f(n))$, where $f(n)$, $\frac{d}{dn}f(n) > 0$ for all $n > 0$, being lost. Therefore, the study's significance lies in the potential speed-up of crowd simulations and perhaps many other applications by this factor $O(f(n))$ while still producing valuable results.

1.7 Dissertation Organization

This dissertation is written in five chapters. The first chapter is introductory and provides the field of study and context in which this thesis is written. The chapter includes background information and research objectives. The objectives are explicitly stated and are sufficient to answer the research questions. Chapter 2 carefully reviews relevant literature to this

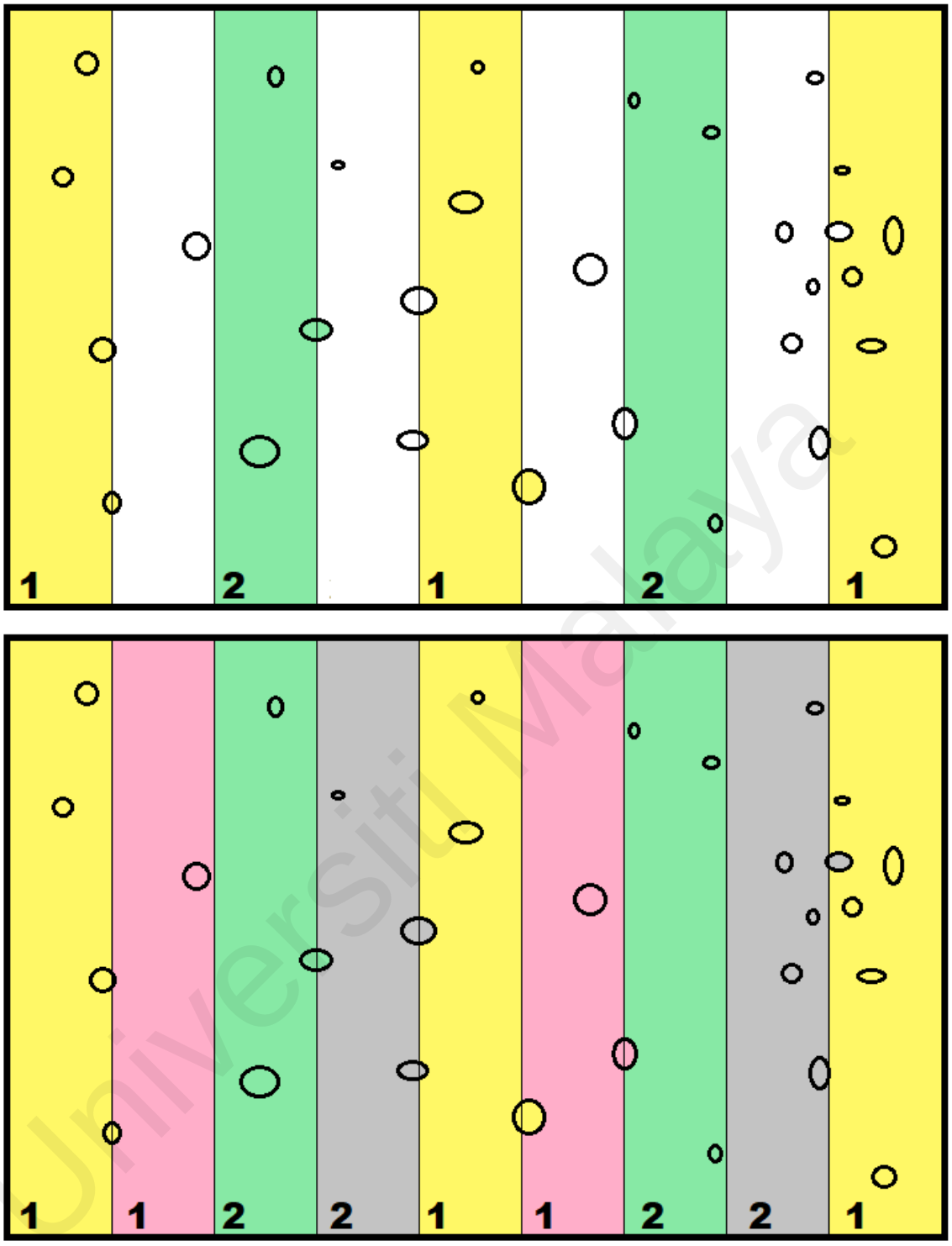

Figure 1.3: A simple plan for distributing work over two processing units (PU). First the yellow lanes are updated by PU 1 and the green lanes are updated by PU 2 simultaneously. Then the same is done for the rose lanes and grey lanes. This approach leaves at least one full lane between any two agents updated in parallel and therefore cannot interfere with each other or be put in the same position.

dissertation. As the research questions asked in this thesis have not been asked before in the context of crowd simulation, there is no literature targeting this topic in particular.

The research fields of crowd simulation and error evaluation, however, are rich in articles and literature. The literature review will therefore focus on all individual aspects of this research and combinations thereof. This way, the literature review should give a substantial knowledge base upon which this research is built. Chapter 3 shows the more technical side of the research and includes important design choices and the tests run on the simulator. All results are reported and discussed in chapter 4 and the dissertation concludes with a conclusion and potential future works in chapter 5.

CHAPTER 2: LITERATURE REVIEW

2.1 Overview

Extensive reading into previous studies and articles has been done to have a complete and inclusive comprehension of the research topic. The chapter begins by reviewing the necessity of crowd simulations. The chapter continues with studies on applications of crowd simulators and the significance of their outcomes. The chapter moves on to review techniques in simulating crowds appropriately. Many researchers have applied parallel computing in large scale crowd simulations in the past. Any branch of computer science dealing with vast amounts of data and computations attempts to parallelize these computations to divide all the work over many cores to speed the process up significantly. Therefore a few articles that have worked on parallelization in crowd simulations specifically will be discussed next. After establishing a sound basis of knowledge on crowd simulations, applications, and parallel computing within this field, the chapter ends with an analysis of the last two categories; error tolerance in previous studies' models and the impact of errors in crowd simulation

2.2 On Crowd Management

The first research evaluated here is on the risk involved with the evacuation of a large crowd (J. Wang et al., 2013). It is one of many studies on crowd management. It is included in the literature review to illustrate the importance of good crowd management, where the predictive power of accurate simulations can improve. This study specifically focused on evacuation planning in large crowds. It finds indicators of evacuation plan efficiency such as waiting time and number of waiting agents. They look at relationships between crowd densities and the rescue strategies that utilize simulation techniques for evaluation.

The researchers' main conclusions were that the efficiency of rescue and evacuation plans are related to the density. A more significant density means more passageways will be necessary. They also found a crowd density limit beyond which the strategies they used, involving waiting queues, do no longer work. An increase in passageways does not yield a significant increase in efficiency anymore. This limit they found to be around five people per square metre. Interestingly, throughout the paper, numbers are rounded to one decimal with this limit of 5 p/m^2 as an exception.

Another article that suggests that accurate crowd simulations can support the design of comfortable, efficient and safe public spaces is (Gorrini et al., 2019). The article studied the relation between crowd grouping, densities and the dynamics of that crowd. The researchers estimated the stress pedestrian experience in a crowd to find the shape of personal space in a series of experiments. The researchers proposed a method to increase comfort for pedestrians while walking in a crowd. This method they verified using crowd simulations, implying the crowd simulations were an important factor in their research. This is why this article is reviewed here as well, as it legitimizes the need for crowd simulations further.

2.3 Using Crowd Simulation

The first journal item read is (Kim et al., 2019), to get insight in usage of crowd simulation and what it can be used for. The researchers analyze walking speed changes as functions of crowd densities in existing simulators and proposed new equations for changing the walking speed of agents during a simulation. They also analyzed the effect of a fallen person in real-life examples. Though this is an exciting event to study, the event could be more exciting when generalized. If the fallen person cannot be stepped on by any agents, then we can generalize it to any event that results in an obstacle, as well as obstacles that

have influence on agents but do not prohibit agents from entering a specific area, such as a bad smell. We think that was a missed chance in this research.

(Datta & Behzadan, 2019) research hazards that can happen to people in accidents, such as an injury. They proceeded to present EVAQ, a framework for modelling large crowds, considering characteristics, level of knowledge about the environment and hazard intervention systems efficiency. Using crowd simulation, they tested their framework for many different settings on the previously mentioned influences. Then they verified their findings by confirming the correct working of the crowd simulator by running tests on the simulator with a building fire that really happened in the past, while having the real world data of a building fire available too.

In (Wong et al., 2017), the researchers present an algorithm they developed to find optimal routes for evacuation of local regions in a bigger environment. In short, their algorithm starts with a shortest path algorithm as the first potential optimal evacuation route and then tries to improve this route using simulations. Their measure of the algorithm's effectiveness includes studying the maximum number of people evacuated using a crowd simulation. The iterative algorithm runs a series of crowd simulations to optimize the route step by step until an optimum is reached. In their article, they present a mathematical analysis to validate the steps. They run the algorithm several times to experimentally verify its results in a given situation. The article is an excellent example of the usage of crowd simulation in research fields in practical situations. The only shortcoming here is that their algorithm assumes that the distribution of people demographics is known prior to running. The argument here is that the algorithm could provide more refined outcomes at the expense of its speed. However, generating multiple distributions of people and producing a set of

evacuation routes corresponding to a set of characteristics of the agents in the simulation may help plan a route in a building. The demographics may not be known yet while giving a better understanding of the risks involved with this planning, given the types of people potentially present in that building.

2.4 Crowd Simulation Techniques

The method for crowd simulation by (Peymanfard & Mozayani, 2018) introduces the concept of holonification. The holonification is a graph theory approach to demonstrate crowd simulation. Their tests show that holonification can significantly increase the effectiveness of a crowd simulation. The researchers apply their method to a pre-defined simulation environment based on real-world data. The aim initially is to reduce the communication necessary between agents. After they established a set of rules for their model that increases the accuracy of their simulation results, they applied the model to a second environment to verify the generality of the proposed model. They used single-holonification theory and machine learning techniques to achieve the result.

A recently published article developed a position-based crowd simulation method that can run a hundred thousand agents in real-time, making it suitable for applications in video games (Weiss et al., 2019). The researchers first studied Position-Based Dynamics to develop a crowd simulator that reduces agents to particles. Then, these particles apply constraints such as speed and position, planning or routes and physics such as collision constraints. The aim is to simulate complex crowds of different densities with robust collision avoidance simulations with huge crowds. The show a simulation with 100,000 agents in a frame rate of 23 frames per second (43.66 ms/frame).

The authors refer to global planning algorithms combined with local collision avoidance algorithms to improve effectiveness and efficiency (Sillapaphiromsuk & Kanongchaiyos, 2019). They argue that agents can get stuck when groups of people walk in opposite directions and block each other or agents walking to or forming congestion themselves. These are clear concerns that have a realistic chance of occurring when not accounted for in the algorithms or models used for the simulation. The goal of this article is to develop a method of crowd simulation that can reduce such problems. The researchers tapped into the literature on biomechanics and optimal control theory to define a measure of energy in path planning. They then included this measure of energy to make agents find minimal-energy paths. The results presented in their article show that it helps agents avoid congestions and overtake more efficiently in a bi-directional stream of people. The researchers verified the real-world predictability by capturing video footage on real-world bi-directional passageways with pedestrians, creating a virtual copy of the initial situation, and then simulating it. Although the proposed method produces realistic crowd simulations, the researchers do not detail the efficiency of their method in running speed. Not having running speed as an objective is not necessarily a shortcoming, but it may affect the method's usefulness in large crowd simulations. This gap may be resolved by fully utilizing the capabilities of all CPU and GPU cores.

(Karamouzas et al., 2017) develop a crowd simulator that is guaranteed to be collision-free. Using concepts from classical mechanics and describing crowds with an anticipatory potential, the researchers proposed something they call the Implicit Crowds Model. They managed to simulate high-quality behaviours in crowd simulations that were indeed collision-free. If any, error margins need to be closely watched when using this model to preserve its collision-free properties. Their model is interesting for works involving real-time crowd simulation.

2.5 Parallelization in Crowd Simulation

Researchers develop an environment to perform crowd simulations on an agent basis in a parallel manner (Malinowski & Czarnul, 2019). They proposed NVRAM to simulate large crowds (>10,000 agents) in large areas (> $1km^2$). They selected NVRAM rather than RAM as it saves overhead in their simulation and guarantees simulation speed. On the topic of parallelization in crowd simulation, the researchers said, quote "In our opinion, there is a need for a general framework for 2D agent-based simulations, that would not only be able to run efficiently in a parallel environment but would also be easy to use in terms of implementing own agent models and verifying these in a convenient way".

Malinowski & Czarnul (2019) presented an architecture that can be used for parallel crowd simulations and described the implementation thereof. Further, they focus mainly on the utility of using NVRAM in addition to RAM when simulating large crowds. Although this change in memory distribution can speed up simulations, the addition of NVRAM would speed up the simulation with and without error tolerance evenly. However, it is less attractive to need hardware changes to verify a significant increase in running speed. Thus, the NVRAM method is less feasible when research considers error allocation to speed up parallel simulations without changing the outcome significantly but within that goal.

Earlier work on parallel crowd simulation evaluates the contemporary methods of crowd simulation and identifies that most of them perform using a GPU for visualization and a CPU for the simulation calculations. The CPU is required, even while the GPU has much more processing power with its many cores. (Yilmaz, 2010) questions this approach and explore methods to fully utilize the power of the GPU by using it as the central processor for simulating large crowds. The author claims the methods evaluated hundreds of thousands

of agents for simulation in real-time. The results presented a simulation of over one million characters and an increase in efficiency by about one hundred times by replacing CPU work with GPU work. Yilmaz (2010) also concludes the bottleneck for this method: memory read and write speed. Tracking memory read and write actions in an environment that are not too memory-intensive seems to work for crowd simulation bearing similar risks to Yilmaz's (2010). For example, it is possible to measure the time needed to perform one simulation step and get the potential increase in efficiency, even with a reduced number of agents.

One work researches the behaviour of a crowd when confronted with one in combat or special operations, where an unexpected confrontation in a crowded area can occur (Chen et al., 2013). The paper suggests that previously, the simulation of such a situation with tens of thousands of agents in a realistic way was not possible. The finding suggests a contradiction to Yilmaz (2010), whose simulator has over one million agents. However, it is observed that Yilmaz's (2010) simulation environment is simplified. In contrast, Chen et al. (2013) show more complex environment settings at the expense of the number of agents that can be simulated simultaneously.

For example, Chen et al. (2013) attempt to send most instructions traditionally handled by the CPU for their simulation to the GPU, so many more calculations can be made within every millisecond. Parallelization using multiple cores at once with GPU extension is an exciting combination for large crowd simulations. The use of general-purpose processing on the GPU is a central concept that serves as a guide on parallel simulations. As such is used as a reference when implementing the simulator.

2.6 Error Tolerance Models

A seed paper for the method in developing an error-tolerant simulator is (Liu et al., 2018). The paper presents parallel computing design on a very large scale where the design tolerates specific errors, as it has mechanisms built-in to repair these errors. The design includes sensing and discovering errors, which it then resolves on the go. For example, specific processes running on the CPU with expensive checks to prevent errors can now run parallel to a GPU to resolve errors. The paper proposed a framework that aims to have similar fault tolerance features as biological neural networks in a scalable way for massive parallel computing.

Liu et al. (2018) found that their system, which they presented with both hardware and software components, is as reliable as conventional systems but had a much lower power consumption. The system consumes less because of the lack of complicated tests and logic required to prevent errors in conventional systems. The paper reports that their system is more efficient than a conventional system if the error rate is below 0.7 already. The article affirms that error tolerance in parallel computing is a way forward, and allowing specific errors to occur to increase the speed is plausible. However, they propose increasing the quality of hardware design to support their hypothesis further. If the errors happen in a controlled way, the method may still be effective. The question will then be, can one speeds up the crowd simulation while keeping power consumption low when implementing only at a software level.

Another paper that references error tolerance to speed up a program without affecting the running outcome is (Dou & Li, 2018). This paper focuses on viewshed analysis, which is a computationally expensive process in data processing. A method is presented to reduce

redundancy by removing parts that would only prevent minor errors during run-time, but drastically reduce the running speed. The program can make full use of all cores on the CPU and GPU, resulting in a speedup of about 15 times the original XDraw algorithm the researchers are referring. Compared to the original XDraw algorithm, their algorithm has a precision rate of about 99.4%. However, the algorithm proposed has some built-in error correction. It is worth noting that using error correction techniques is a feasible option. After all, agents should not get stuck as the result of an error and block the whole simulation.

2.7 Impact of Errors in Crowd Simulation

The last element to review is work relating to error impact. Although the researchers Dou & Li (2018) already put a number on the precision of their algorithm, they did have a few error corrections in place. They did not explicitly analyze the impact of specific errors. (Zhang et al., 2019) specifically analyze the impact of position errors in crowd simulations. However, there are assumptions that the errors analyzed are generated by the imprecision of devices, such as positions collected from GPS-enabled devices. In real-time, they want to predict congestions and increase safety based on real-time information. The researchers specifically want to see the effect of position errors, as mentioned in the predicting abilities of the crowd simulation models. They learn that position errors significantly influence the outcomes produced by models. The predictability of such crowd simulations has somewhat limited reliability, even when filtering techniques are applied.

The research performed by Zhang & Miranskyy (2019) is insightful and spur the interest in utilizing filtering techniques and better error position consideration in crowd simulation models. The message is clear, tolerating errors of magnitude in scale can be a factor, particularly in crowd simulation, where the GPS errors can gain several meters in dis-

placement. Significantly, they show that error tolerance might result in adverse outcomes when the errors allowed are too large due to external influence. There is a higher chance of keeping the error displacement under control, primarily when the crowd simulation software itself produces the errors. Errors should not be visible immediately, but if they produce significantly 'wrong' results, they should show deviations that increase gradually over time.

Crowd simulations are often used to detect densely occupied areas where people could suffocate or find themselves under high pressure (Sun & Badler, 2018). They use a physics-based system to simulate compression of crowds in areas with barriers, preventing people from getting in or out at the boundaries other than the gateways provided. The domain-specific approximate crowd simulator is designed to simulate the effects on a crowd as it evolves in a given situation and area. One uses a conventional crowd simulator to evaluate the same scenario and compare its results to the ones presented in this article. However, Sun & Badler (2018) has an entirely different perspective on the issue presented. They propose that simulations and results can be achieved in multiple ways rather than a universally optimal solution, implying crowd simulations can be performed with a certain degree of error tolerance.

2.8 Concluding Remark

Crowd simulation is a popular way of testing a crowd's complex behaviour in various situations and is a popular field for researchers to investigate. Several essential elements to consider in building a crowd simulator include crowd management and simulation techniques. They form a solid base of knowledge upon which this dissertation stands. Parallelization, models of error tolerance and its impact are proven to advance crowd simulation work and are worth investigating. Interestingly, none of the literature reviewed

in this chapter considers speeding up a crowd simulator by allowing errors to occur. A potential reason for this is that error tolerance is somewhat an unorthodox way of speeding up a crowd simulator, especially in a parallel context. The intended error tolerance methods render the simulator non-deterministic, which means none of the results can be reproduced precisely. This gap that was left in the existing literature is addressed and filled by this work.

2.9 Chapter Summary

Current literature related to crowd simulation covers most aspects of the field, such as the usage and relevance of crowd simulation, ways to simulate a crowd realistically and in the fastest way, and the impact and significance of errors in such simulations. At the intersection of these topics, the idea proposed in Chapter 1 is left untouched. By creating a complete crowd simulator from scratch, the gap from previous works can be tested, and the conclusions can be helpful for any speed-driven field where error tolerance is an option. The next chapter will describe how this is all implemented and how the results are found.

CHAPTER 3: METHODOLOGY

This chapter consists of a detailed explanation of the simulator's design choices and technical side and the tests and data generation. The first part covers the program architecture, followed by in-depth descriptions of core features within the software. The second part of the chapter will take the reader through the test settings and the combination of data-generating objects and data collecting and reporting objects.

Regarding the research objectives, sections 3.2 through 3.6 solely support research objective two, which is to build a complete crowd simulator that allows for parallel, error-tolerant and error-proof code. Sections 3.7 through 3.11 focus on both objectives one and two. They describe the simulator further and in doing so, many elements that play a significant role in the course of a simulation get exposed. Many test simulations have been performed to test and fine-tune every aspect of the simulator. These tests further verify the important simulation metrics and find the remaining ones. Following the metrics identification and completion of the simulator, the chapter describes the statistics and data gathered in 3.12 through 3.14.

3.1 Definitions

To prevent ambiguity on the meaning of terms used in this thesis, this section defines them exactly.

Term	Definition
Error-Tolerant	Not having measures to prevent all possible errors. The errors that are deliberately not prevented, will have to be corrected at a later stage.
Error-Preventive	Having measures to prevent errors that can have a significant impact on a simulation, such as position and parallellization errors. This does not mean the simulator is completely error-proof, since the simulator could still have rounding errors.

3.2 Program Architecture

The first goal was to build a simulator from scratch rather than using a crowd simulator already on the market. There is no crowd simulator on the market that lets its user implement or use elements known to make the simulation non-deterministic and prone to errors. Therefore, the simulator in this dissertation is built to have complete control over the code and implementation. In doing so, the simulator features choices to have the simulation run as fast as possible. The guideline, however, excludes the set-up time. Whenever possible, calculations are done beforehand so that the simulator can run large simulations with a frame rate as high as possible.

3.2.1 Language and Libraries

The simulator is written in C#, an object-oriented programming language. Developed by Microsoft, C# offers very extensive documentation and there are many libraries available for this language. C-family languages are also known to be faster than other conventional languages such as Java or Python. The object-oriented approach suits the application very well, as every agent and every environmental element can be coded to be a separate object. C# is not the fastest from the C-languages but is the most recent and offers the most

functionality. C# also provides a fantastic toolbox to use for simulator development. A set of such C# tools is found in the OpenTK toolkit used here. OpenTK is the perfect fit for this research as, in the developers' own words, *"OpenTK is not a game engine, nor does it try to be one. It offers a useful set of libraries, and leaves it up to the developer to make the decisions on how to use them. It is the ideal starting point if you want to write a game or scientific application from the ground up."* OpenTK offers a cheap and, therefore, fast way of using a *game loop* and pixel drawing. Some alternatives could be XNA or Unity3D, which do present themselves as game engines. These engines provide more tools and more graphically appealing simulations. This added functionality is provided at the expense of its speed, however. For most applications this is no problem, since they are not slow engines, but if sample size and speed are the most important factors, then the engine of choice should not trade any speed for aesthetics.

3.2.2 Program Structure

Figure 3.1 shows the structure and classes that make up the simulator. As seen in the diagram, many static classes manage and support the application. The classes SimulatingState, Camera and DrawHelper from the UML diagram support drawing the screen as described in 3.3. The SimulationManager, Scene and InputHelper classes help loading environments and detecting input as described in Sections 3.4 and 3.5. Section 3.6 discusses the line objects, which correspond to classes Line, CountLine and SolidPolygon in the UML diagram. The behaviour described in Sections 3.7 and 3.8 are internally handled by the Graph, Scene, Agent, Spawn and Goal classes. As seen in the diagram, many static classes manage and support the application. When the program starts, as with any program, the Main function is called, and the program is instantiated. The Main function is also where the game loop resides. After instantiating the program itself, the static classes are all set up for their respective jobs. First and foremost, the program calls for initialization

of the `SimulationManager`. This class is static as there is only one class managing the simulation. From anywhere in the application, a method should call for the simulation manager to change values on an application level. For example, if the application needs to be paused or the user wants to export statistics, the responsible classes can directly communicate with the managing class.

Classes can also use user input from anywhere by calling for the `InputHelper` class. This class stores keyboard information and is updated outside the simulation time step. The static class allows detectable user input at all times and anywhere, regardless of the application's state.

The last two static classes that are not directly linked to simulating the agents are the `Camera` and `DrawHelper` classes. These static classes are the reason no object can draw itself directly onto the screen. Any object drawing anything on the screen does so by calling for the camera. This way there is a central place where the camera offset is stored, updated and used. If the camera moves to a different part of the environment, it remembers the new location and all objects drawing themselves using global coordinates are drawn accordingly. The HUD stays in the same place and does not get affected by the camera moving around. In turn, the camera uses the DrawHelper by calling for sets of pixels for shapes such as circles. These pixel sets are stored as they do not need to be calculated often and classes call for the DrawHelper to build a new shape when one occurs.

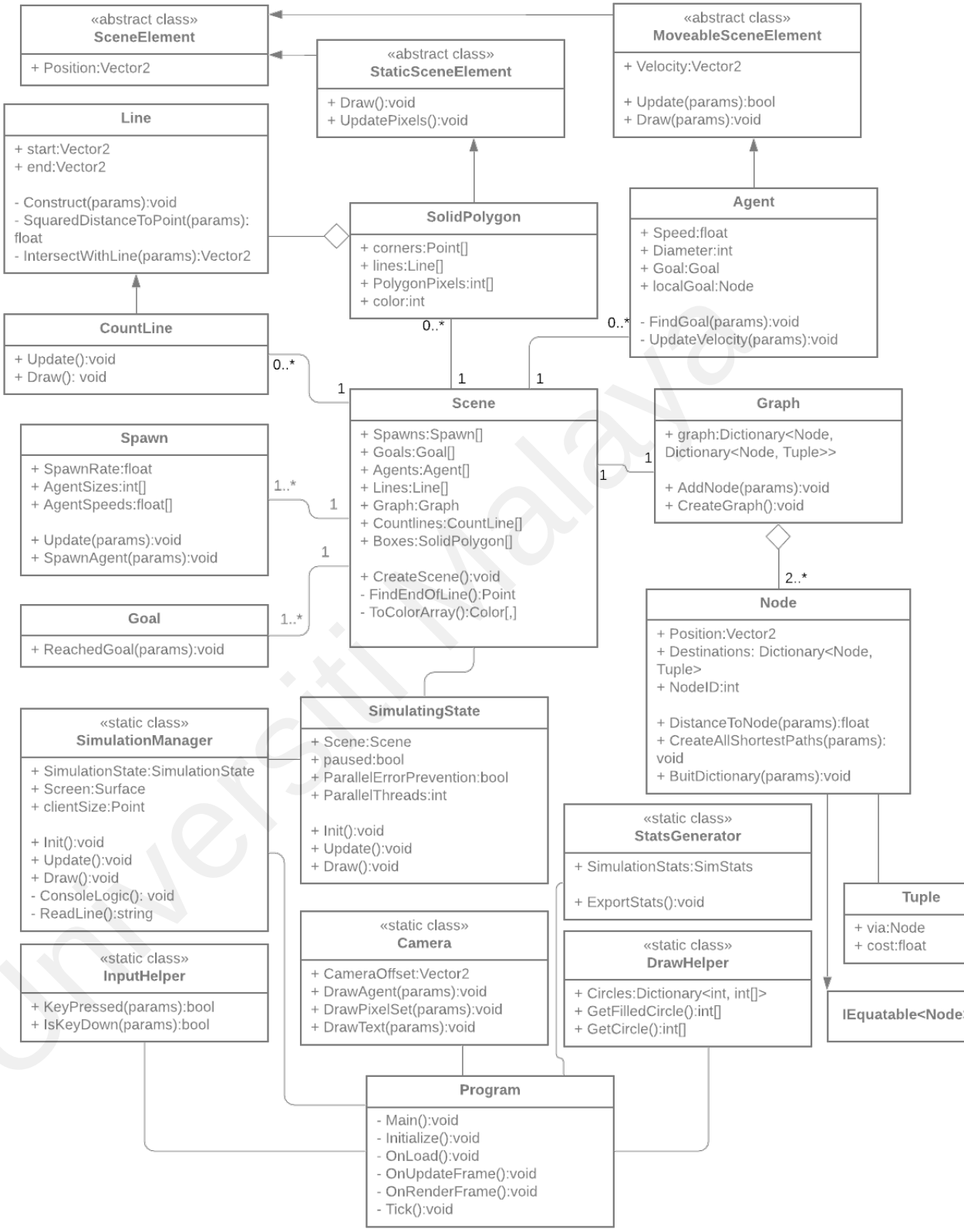

Figure 3.1: This UML diagram shows the most important classes and links between classes that together make the simulator.

Together, these static classes form the foundation upon which the simulator is built. The simulator itself starts with the `SimulationState` class. The class is an instance of the simulator's state, which is by default ready to run. The class also has practical variables, such as the Scene and the settings for parallelization. The individual updates for objects and user input are called and handled in this class since this class has all the simulation information. This class is the lowest level at which almost the entire calculation and drawing of individual frames can be seen. The lower level classes all only have pieces of the update and drawing logic for a full-frame.

The most crucial class below the simulation state is the `Scene`. The scene encompasses all the information about the environment, including walls, waypoints, agents, spawns, and goals positions. The scene also builds and contains the `Graph` that tells agents where to go next when walking along the shortest path. The spawns and the agents maintain the list of agents within the scene. The spawns add the agents to the list, and agents who arrive at their destination remove themselves from the list. These `Agents` are objects that use the graph and scene to navigate and to avoid collisions. They receive a fixed speed and size when created but do not necessarily have the same speed and size as other agents. All agents share the same pathfinding algorithm; more on that later in this chapter. Just before an agent removes itself from the list, it passes its statistics on to the goal it moved to so the goal can compile the statistics from all agents into more simulation metrics.

The last major part of the architecture is the statistics generation. Besides the libraries `System.Collections.Generic`, `System.Linq` and `System.Threading.Tasks`, which are pretty standard for any program using lists or parallel computation, the simulator taps into the `System.Diagnostics` library to make use of Windows' built-in diagnostics tools

such as the stopwatch. By letting functions from this library handle the timing of time steps, the simulator is guaranteed to get accurate calculation time measurements. The place where this all comes together is the static `StatsGenerator` class. The class exports the origin and destination of the stats generated throughout the application into a text file.

3.3 Drawing the Screen

OpenTK is used for the drawing of every frame. The OpenTK library provides its user with OpenGL functionality in C#, so its interface can be used for rendering vector graphics. In the case of the proposed simulator, it can draw single pixels on the screen cheaply. The screen is consequently stored as a one-dimensional array of integer values where pixel (x, y) is coloured as follows:

```
Screen[x + Width * y] = 0xff00ff;
```

The Width is the screen width and 0xff00ff is the hexadecimal representation of the colour fuchsia, in regular RGB written as (255, 0, 255). Any 24 bits colour can be assigned to any screen pixel using such statements. This method is one of the cheapest ways to draw a frame.

The main objects to draw are

1. circles, defined as $C(P, r)$, where P is the position of its centre and r is the radius,
2. lines $l : \binom{x}{y} = \binom{S_x}{S_y} + \lambda\binom{D_x}{D_y}$, where S is the line start, D is the line direction and $0 \leq \lambda \leq 1$ is a variable,
3. polygons, defined as a collection (`List`) of lines.

Circles are generated in run-time so that any sizing agent can be drawn. Of course not all sizes are needed and therefore they are generated on a need-build basis. A dictionary in the

`DrawHelper` class stores coordinate sets with the diameter as the key for any coordinate set. Algorithm 1 shows how a such a pixel set is made and Algorithm 2 shows how the correct circle is returned when asked for it by another class.

Algorithm 1: Get the coordinate set for a circle

```
Result: The circle of choice in rA
diameter d;
border b;
returnArray rA;
switch type do
    case "thin circle" do
        b = 1
    end
    case "thick circle do
        b = 4
    end
    otherwise do
        b = 0.5*d
    end
end
for j = 0; j < d; j++ do
    for i = 0; i < 2 * π; i += 0.001 do
        rA.Add(Cos(i) * (d-j) * 0.5 + Width * Sin(i) * (d-j) * 0.5)
    end
end
```

Algorithm 2: Receive the coordinate set in screen coordinates

```
Result: All screen coordinates for this circle rA
diameter d;
type t;
position p;
returnArray rA;
if key d ∉ dictionary for t then
    Get circle and add set to library;
end
copy the circle from dictionary to rA;
foreach Coordinate c in rA do
    c += p;
end
```

It is important to note that the spawns will call for these functions before spawning

agents. Spawns are updated in serial rather than parallel and, therefore, will not try to add the same key-value pair to the dictionary twice. As a key can only occur once in a dictionary, parallel processes adding the same key-value pair to the dictionary by accident can crash the application. The approach to draw circles comes from the notion that drawing large amounts of agents can be pretty expensive. With this approach, the Cosine and Sine functions and the loop for finding the local circle coordinates are eliminated from the game loop, saving precious processing time. The current approach uses a Dictionary, which in C# has an $O(1)$ time-complexity and a loop through the already available coordinates. In practice, this loop is not of order $O(n^2)$ as one might expect.

In all test runs, the size of agents was limited to a specific diameter, d_{max}, because a large scale simulation does not work if no significant amount of agents fit in the given environment. Since the circle described by the coordinates fits inside the square of size d_{max}^2, the time-complexity of the loop is bound by the constant d_{max}^2 and therefore is of complexity $O(1)$ itself. For completeness however, when using uncontrolled random diameters or simulations where the diameter is not explicitly limited, the time complexity would indeed be $O(d^2)$. For the proposed simulator, the conclusion is that the total time complexity for drawing agents is constant and very fast in absolute terms. As the amount of agents increases, so does the amount of circles drawn. With the current implementation of circle drawing, this poses no bottleneck to the simulator.

Lines provide the luxury of not requiring extensive speed-ups on the drawing end, because the simulation environments used have a limited number of lines. It is therefore more important that lines are drawn correctly rather than quickly. Lines are drawn in the same way circles are drawn, except that the coordinate set is a simple set of coordinates following

the line formula. Cropping of lines and circles happens within the camera that has to draw the actual pixels. Note that in the test environments, lines are used to represent straight walls, not rounded walls. The lines as implemented are not suitable to use in large quantities of small sizes to approximate continuous non-straight curves. These would require different representations and are not implemented as the added value concerning the research questions is very small.

As agents can be drawn in different sizes to accommodate for larger crowds, the average agent has a diameter measured internally in pixels and represents a person with a diameter of 1 meter. For example, when drawing a few agents in a small area, agents could have a diameter of 20 pixels internally, which would mean 1 pixel on the screen corresponds to 5 *cm* in the environment that is being simulated.

3.4 Reading input environments

There are multiple ways to define an environment. The least elegant is, as with so many applications, to hard code the environment and settings. For obvious reasons, this is not the preferred method of defining an environment. An alternative is to save all the coordinates and information into a text file and read everything from there. Although coordinates make it easy to define a very precise environment, they are hard to comprehend. Tweaking the environment, on the other hand, is quite tricky using only coordinates. For this reason, the simulator uses a picture instead of a list of coordinates. This picture can be any extension, such as PNG or JPG. The picture is converted from an image file into a two-dimensional array of `Color` objects using the `System.Drawing` library. The colours in this array can easily be compared and values of its red, green and blue components can be read and changed. By default, the background is white. The lines are read by looking for the pixels that are not white and following their trail as seen in Algorithm 3.

Algorithm 3: Get from the array of colors to a line

```
Result: Line object
colorArray C;
Line L;
find first pixel (x, y) for which C[x,y] ≠ White;
start = (x, y);
lineColor lC = C[x,y];
C[x,y] = White;
end = start;
while end coordinate changed do
    for offsetX = -1; offsetX < 2; offsetX++ do
        for offsetY = -1; offsetY < 2; offsetY++ do
            if C[end_x+offsetX, end_y+offSetY] = lC then
                end = (end_x+offsetX, end_y+offSetY);
                C[end_x+offsetX, end_y+offSetY] == White;
            end
        end
    end
end
assign a new line Line(start,end) to L;
```

There are two side notes to Algorithm 3. First, it assumes a line is one pixel wide. If multiple pixels of the line colour surrounds the most recent end, the last one seen will be chosen as the next pixel in the line, while the others are turned to white. The information may get lost, and possibly the line gets split up this way. The other side note is that a single line uses a single colour. Lines that cross each other can be identified using a larger offset, where the algorithm checks multiple layers of pixels around the current end pixel. The lines must have different colours to do so successfully. Figure 3.2 shows how different inputs can lead to correct and incorrect outputs, when the guidelines are not followed. Some guideline-violating inputs still get converted correctly, but this is not guaranteed. Since the research objective is not to convert all possible input lines into correct output lines, converting just the guideline-abiding inputs correctly is sufficient. Spawns and spawn settings are slightly trickier. It is not feasible to draw those out, especially considering the number of settings a spawn can have. One solution in consideration was to draw them

and use the console to assign the values. For this particular research project, this was not necessary, since simulations will only have a very limited amount of spawns and goals and the position of these areas will not be tweaked as much at their settings and therefore these are manually entered into a text file and read at the simulation set-up stage.

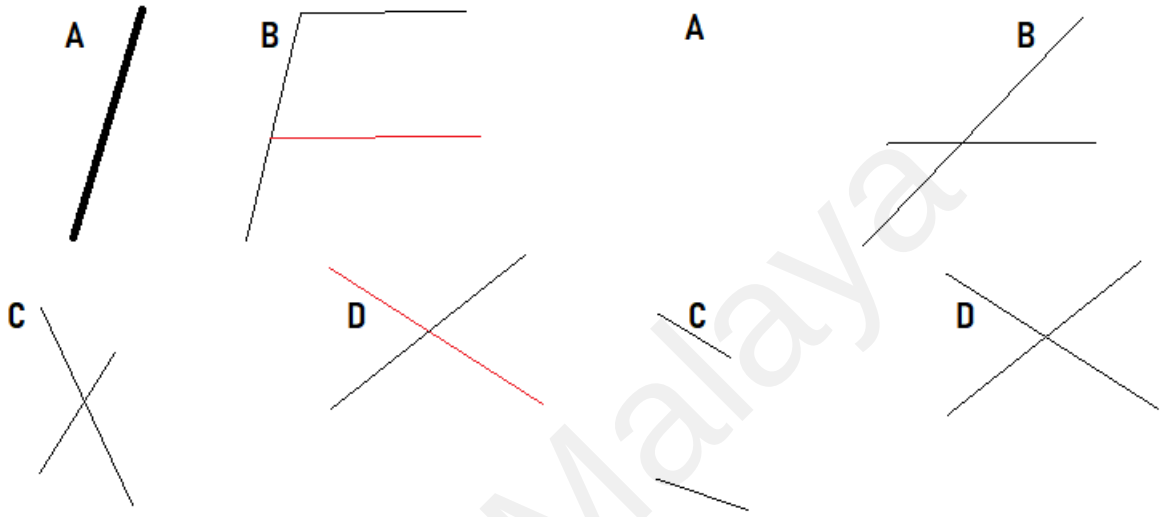

Figure 3.2: A: The line is too thick and gets erased completely. B: The red coloured line gets converted well, the piecewise linear black line become one line from start to end. C: The same-coloured lines are followed along the wrong path, causing two different lines. D: The same lines using different colours get correctly converted into the simulator.

3.5 The Console

OpenTK provides input detection for the mouse and keyboard. This input is detected using the previously mentioned `InputHelper` class. Every frame the simulator checks if the user made an input through a hotkey. An example of such a hotkey is an arrow key that changes the camera offset and consequently moves the screen to the left or right. When the hotkeys are not sufficient for the input a user wants to implement or give, the console can be used. The console is connected to the application but has its own window. Every tick, the console can be read from and written to. This makes the console perfect for tasks that would otherwise be tedious, such as changing the camera speed. A regular `Console.ReadLine()` action, however, will not work, since this is a busy-wait function.

Using that function would mean waiting for an input to be entered every frame before rendering the next. Since the console does not have an interrupt built-in for when the user types in a command, the `ReadLine()` function is used with a time-out upon receiving no input. This time-out is a single millisecond. A millisecond is long for the application to detect line entering, but not so much as to slow down the simulation.

3.6 Lines

In order to handle collisions, find distances and build the graph, the `Line` class needs to be built up. A line is more than just some starting point and some ending point. First of all, the line is defined as

$$l : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} S_x \\ S_y \end{pmatrix} + \lambda \begin{pmatrix} D_x \\ D_y \end{pmatrix},$$

where (x, y) are screen coordinates, S is the starting point, D is the line direction as seen from the starting point and λ is a variable in $\{x \in \mathbb{R} | 0 \leq x \leq 1\}$. Now assume agent A is defined by

$$A = (P, r),$$

where P is its position and r is its radius. For A to collide with a wall represented by l, the distance between P and l must be smaller than r. Assume the line has infinite length. In this case, the shortest path from P to l follows a straight line perpendicular to l. This means it follows the line

$$k : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \end{pmatrix} + \mu \begin{pmatrix} -D_y \\ D_x \end{pmatrix},$$

where $\mu \in \mathbb{R}$ is a variable. The point of intersection between l and k is found by solving

$$\begin{pmatrix} S_x \\ S_y \end{pmatrix} + \lambda \begin{pmatrix} D_x \\ D_y \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \end{pmatrix} + \mu \begin{pmatrix} -D_y \\ D_x \end{pmatrix}. \tag{3.1}$$

This system of two equations results in

$$\mu = \frac{S_y + \frac{D_y}{D_x}(P_x - S_x) - P_y}{D_x + \frac{D_y^2}{D_x}}$$

and

$$\lambda = \frac{1}{D_x}(P_x - \mu D_y - S_x).$$

Now, as μ has a value, the distance between the line and the point becomes clear. The distance between point P and l is the distance between the intersection of l and k and P itself. From equation 3.1 and the coordinates of P it is clear that the difference between these two points is the length of vector $\mu\binom{-D_y}{D_x} = |\mu|\|D\|$. Before calculating the distance between P and l, one thing first needs to be set straight. The above mathematics was done under the assumption that l has infinite length, which is equivalent to saying $-\infty < \lambda < \infty$. But since $0 \leq \lambda \leq 1$ and λ is known now, this condition can simply be checked. If $\lambda \notin [0, 1]$, then it means the shortest distance is not $|\mu|\|D\|$, but simply the distance from P to the closest line end. The final answer is

$$\text{Distance}(P, l) = \begin{cases} |\mu|\|D\| & \text{for} \lambda \in [0, 1] \\ \text{Min}(\|P - S\|, \|P - S - D\|) & \text{else} \end{cases}. \tag{3.2}$$

The distance between an agent and a wall is calculated many times during the simulation. Many tricks can be used to reduce the number of times this distance needs to be calculated, but it will at least be $O(n)$ times, since agents must figure out this distance every frame or every few frames. It is therefore imperative to make this process take little time. Luckily, many terms in $|\mu|\|D\|$ are in fact constants. Since S and D are immutable and known at

the start, part of μ can be calculated once and recycled every time like so:

$$|\mu|\|D\| = |\frac{S_y + \frac{D_y}{D_x}(P_x - S_x) - P_y}{D_x + \frac{D_y^2}{D_x}}|\|D\| = |c_1 + c_2(\frac{D_y}{D_x}P_x - P_y)|\|D\|,$$

where

$$c_1 = \frac{S_y - \frac{D_y}{D_x}S_x}{D_x + \frac{D_y^2}{D_x}},$$

$$c_2 = \frac{1}{D_x + \frac{D_y^2}{D_x}}$$

and

$$\|D\| = \sqrt{D_x^2 + D_y^2}$$

So c_1, c_2, $\|D\|$ and $\frac{D_y}{D_x}$ are calculated once and stored in the `Line` object. Then one problem remains to be solved. In these calculations, several fractions can have a denominator equal to zero. This is the case for $D_x = 0$ and $D_x + \frac{D_y^2}{D_x} = 0$. First of all, the latter would mean that $\frac{D_y^2}{D_x} = -D_x \implies D_y^2 = -D_x^2$, the solution of which is not in $\mathbb{R}$ for $D_x \neq 0$. The only case that would result in a non-existent value for μ would therefore be $D_x = 0$. The result can be seen immediately when the `Line` is instantiated. In such a case, a different formula for μ is used. Solving the system of equations 3.1 in a different order yields the solution

$$\lambda = \frac{1}{D_y}(P_y + \mu D_x - S_y),$$

$$\mu = \frac{P_x - \frac{D_x}{D_y}(P_y - S_y) - S_x}{\frac{D_x^2}{D_y} + D_y}.$$

When using these formulae for λ and μ, the constants c_1 and c_2 are adjusted accordingly. The case $D_y = 0$ is no concern, as this would mean both D_x and D_y are equal to zero, implying there is no line, but only a single point. The program does not allow a line that is

not a line.

For a line intersecting another line, the same techniques are used. Suppose two lines may or may not intersect,

$$l : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \end{pmatrix} + \lambda \begin{pmatrix} D_{P_x} \\ D_{P_y} \end{pmatrix}$$

and

$$k : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} Q_x \\ Q_y \end{pmatrix} + \mu \begin{pmatrix} D_{Q_x} \\ D_{Q_y} \end{pmatrix}.$$

A potential intersection will be the solution to

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} + \lambda \begin{pmatrix} D_{P_x} \\ D_{P_y} \end{pmatrix} = \begin{pmatrix} Q_x \\ Q_y \end{pmatrix} + \mu \begin{pmatrix} D_{Q_x} \\ D_{Q_y} \end{pmatrix},$$

which is

$$\lambda = \frac{1}{D_{P_x}}(Q_x + \mu D_{Q_x} - P_x), \mu = \frac{Q_y - P_y - \frac{D_{P_y}}{D_{P_x}}(Q_x - P_x)}{\frac{D_{P_y}}{D_{P_x}} D_{Q_x} - D_{Q_y}}.$$

As before, it is checked first that $\frac{D_{P_y}}{D_{P_x}} D_{Q_x} - D_{Q_y} \neq 0$. In contrast to the calculation on point-line distances, the intersection is immediately determined non-existent when the denominator would be zero. If it would be zero, that implies $\frac{D_{P_y}}{D_{P_x}} = \frac{D_{Q_y}}{D_{Q_x}}$, meaning the lines are parallel as they have the same direction. Normally, this means the lines either got zero points of intersection, or the lines coincide. The latter is not possible, since the lines are read from an image file and every pixel can only have one colour, meaning every point in the environment can only belong to one single line. It follows that there is no point of intersection when the denominator would be zero. When the denominator is not zero, the lines intersect if $0 \leq \mu, \lambda \leq 1$ and the point of intersection can easily be calculated by plugging either μ or λ into their respective formulae.

3.7 The Shortest Path Graph

3.7.1 Dijkstra

There are several ways to find the shortest path from A to B. A customized version of a very well-known algorithm known as Dijkstra's algorithm is used for this simulator. Developed by Dutch computer scientist Edsger W. Dijkstra, this algorithm was initially designed to find the shortest path from one node (the *root*) to another. It does so by starting with an array containing all the nodes with status "unvisited" and distance ∞ except for the root, which has distance zero. Then an array for the finished nodes is instantiated. A node is finished when a known path to this node is the shortest path towards that node. The root is included in this array from the start. Now, all the nodes that have a direct connection to the root are evaluated. The one with the shortest distance is added to the finished nodes, as it is impossible to get there faster under the assumption that distances are strictly positive. This node is then saved with its distance from the root. This process is repeated until there are no reachable nodes left in the "unvisited" list. Depending on the implementation, the nodes that are not finalized can store their temporary distance from the root if they are connected to any finished node. Our implementation of the algorithm follows in a section dedicated to the Custom Dijkstra algorithm specifically, but first there must be a graph to begin with.

3.7.2 The Graph

A graph is a collection of nodes or waypoints and connections between these nodes. Not every node is connected to every other node, and not every node is necessarily reachable by every other node. For nodes that are connected, there is a cost to get from one node to the other. In general, this connection can be one-way and the cost can be any number or a cost function. In this simulator, every connection is two-way and has a constant cost, called the distance, which is always greater than zero. All nodes with no walls intersecting the

straight line between them are connected and the cost in both directions is the Euclidean distance. For implementation purposes, nodes that are not connected, get distance ∞. In run-time the distances are not used; they are only used in building the graph.

The nodes are generated based on the lines created from the image input. The `Line` objects store information about starting point S, direction D and ending point E and functions to quickly find intersections with other lines. Algorithm 4 shows how from this collection of lines the collection of way points is made. The nodes on each end are found from the following formulae:

$$\begin{aligned} n_{1,2} &= S - d\frac{D \pm D^{\perp}}{\sqrt{2}\|D\|} \\ n_{3,4} &= E + d\frac{D \pm D^{\perp}}{\sqrt{2}\|D\|}, \end{aligned} \tag{3.3}$$

where d indicates the distance from the line end to the waypoint and $D^{\perp}$ is the vector perpendicular to D. For line intersections there must be four way points. For any point of intersection P between lines l and k, the four nodes are

$$\begin{aligned} n_{1,2} &= P \pm d\frac{D_l + D_k}{\|D_l + D_k\|} \\ n_{3,4} &= P \pm d\frac{D_l^{\perp} + D_k^{\perp}}{\|D_l + D_k\|}. \end{aligned} \tag{3.4}$$

These are used in the second part of algorithm 4.

Indeed, the time-complexity of this operation is $O(n^2)$ as a result of the double for-loop at the end. Considering the number of lines and where this happens, this is not of any concern in the set-up stage. Then, with a similar double for-loop, the program builds a distance table, where for each pair of nodes (n_1, n_2) a temporary line is drawn and if the line intersects with any permanent line, the distance is ∞ and otherwise it is $\|n_1 - n_2\|$. Since the `Node` class inherits from `IEquatable<Node>`, nodes can be used as keys for a dictionary. The distances can then be stored in a double dictionary and can easily be found

Algorithm 4: Building an unsorted list of all relevant way points

```
Result: A list N containing all nodes for the graph
arrayOfLines L;
allNodes N;
for i = L.Length - 1; i > -1; i– do
    Add 4 nodes to N, two on each end of the line, as in equation 3.3;
end
for i = L.Length - 1; i > -1; i– do
    Line l = L[i];
    Remove l from L;
    for j = 0; j < i; i++ do
        if l intersects L[j] then
            Add nodes to N, according to equation 3.4;
        end
    end
end
```

in $O(1)$ time using the two nodes as keys.

3.7.3 Customized Dijkstra

Usually, an agent calculates its path when it spawns using Dijkstra or Dijkstra variant A*, but that is not going to cut it for large-scale simulations. These algorithms, which scale with (powers of) the number of nodes and connections, can take too much processing power when performed in real-time. Since this is unnecessary, Dijkstra is used once before the simulation and its results are stored. Most agents follow similar paths, so it would be a big waste of computational power to calculate these paths more than once. In Algorithm 5 the algorithm is shown. The result is a double dictionary called Destinations, which takes any two nodes as keys and returns a tuple (N, c). N represents the next node to walk in getting to the destination using the shortest path and c represents the cost of getting there. This way, the entire path is not stored for each destination and agents must ask what their next waypoint will be every time they reach their current waypoint.

As such, the storage needed is only of size $O(n^2)$, with n the number of nodes, to store the entire graph. Since the algorithm uses a triple for-loop, looping over the nodes every time, the time complexity of this algorithm is $O(n^3)$. The time complexity can be improved by using a priority queue in the Dijkstra algorithm and changing the outermost for-loop to only go over the directly reachable nodes from the spawn. The current algorithm has proven to be fast enough for the proposed simulator, but for many nodes, these are potential fixes for slow set-up times. The relatively high time complexity of this part of the simulation set-up pays off significantly during the simulation. Since the agents only need to remember the next waypoint, the storage space needed is $O(1)$ for each agent. The agents ask for their next waypoint from the double dictionary, so finding the next waypoint and their globally shortest path only takes $O(1)$ time. All combined, the algorithm allows for a simulation that requires practically no computation time for global pathfinding.

3.8 Pathfinding

The simulator implements two types of pathfinding. The first is global pathfinding and is widely used in many applications within and outside the field of crowd simulation. As the name suggests, it helps agents find the shortest path from A to B anywhere in the environment. The second one is compass pathfinding, which is much less used and uses only local information to navigate towards the goal. The reason to implement two vastly different types of pathfinding is that there are many ways to simulate a crowd and they may all respond differently to error-tolerance. Therefore, if conclusions hold for both types of pathfinding and across different simulation settings, they are more likely to be true for crowd simulation in general. The setting fixes the pathfinding definition before starting the simulation and will not be altered during the simulation so that all agents will use the same type of pathfinding.

Algorithm 5: The customized Dijkstra algorithm

```
Result: The shortest path graph
Create double dictionary Destinations for final graph;
foreach node in allNodes do
    Create a dictionary allPaths uses a next node as a key to a list of nodes
      reachable from that next node and the distance;
    Create an array of key-value-pairs with the distances to each node;
    Create an array of bools for all finished nodes;
    foreach node from allNodes do
        Mark each node as distance ∞ and as finished False in the respective lists;
    end
    Add this node as the root to the lists;
    for i=0; i< allNodes.Count-1; i++ do
        u = -1;
        min = ∞;
        foreach node in allNodes do
            if node is not finished then
                Store smallest distance in min;
                Store its index in u;
            end
        end
        if min is ∞ then
            break;
        end
        Mark node u as finished;
        if u is not the index of the root then
            if the node is connected to the root then
                Add node to the allPaths dictionary as a new entry;
                Add node to its own entry;
            else
                Find the first node connected to the root that leads to this node and
                  add it to that dictionary entry;
            end
        end
        foreach node in allNodes do
            if node is not finished then
                update distance;
            end
        end
    end
    foreach list of nodes in the allPaths dictionary do
        Add each end node in this list to the Destinations double dictionary with the
          next-reachable node as the value;
    end
end
```

3.8.1 Global Shortest Path Finding

Global shortest pathfinding is a pathfinding technique that allows agents to find their way from A to B with the shortest travel distance possible. This type of pathfinding is used for environments where agents are assumed to have full knowledge of the environment or an abundance of clear signs indicating the shortest path to any destination. This type of pathfinding is suitable for environments such as airports and office buildings. At airports, many signs indicate where to go. In-office buildings, the same people visit the same areas many times, meaning they are well familiar with the map of the area.

When an agent spawns, it receives a destination in the goal area. These two nodes are not in the graph, and therefore there is no instant path to the goal. This problem is solved by having the spawn and goal check which nodes of the graph are directly reachable before the simulation starts. From the graph, it becomes clear which routes are the shortest from spawn to goal. Note that the spawn and goal are areas, not points. Therefore, there are many connections between the spawn and goal that are the shortest from their respective areas within the spawn. The agent will tell the spawn its position, and receives the first waypoint. When the agent reaches the waypoint, it can directly ask which waypoint is next to reach its goal. This next waypoint is obtained from the customized Dijkstra algorithm. The algorithm provides the information that is needed to know what node to move to next, given the current node and the goal. When the final waypoint is reached, the agent will walk straight to its assigned destination. Agents cannot walk through other agents. They will walk around other agents to reach their next waypoint when possible. The technical side of this pathfinding is explained in the previous section; the agents simply use this functionality of the graph to find a velocity.

3.8.2 Compass Path Finding

Compass pathfinding means the agents know which direction their goal is in and try to walk towards it. In doing so, they do not necessarily follow the shortest path but locally try to get as close to the goal as possible. This pathfinding is suitable for emergencies where people try to get as far from the health hazard as fast as possible. Another situation where this pathfinding could be more realistic than global shortest pathfinding is when people walk in the streets of an unknown environment and try to walk towards a landmark, using the landmark as their point of reference. When agents run into a wall, the priority is to go into a direction that brings them closer to the goal. This direction is followed until the path towards the goal is free again. The approach is not always the shortest way towards the goal, but it is their best guess in a very local sense. Figure 3.3 shows the trajectory an agent can follow. The blue lines indicate walls, while the dots represent the agent, walking from the green to the red position. After starting to walk in the direction of its goal, the agent makes the following decisions:

1. The path is obstructed. Walking upwards shrinks the distance between the agent and the goal for the next frame while walking downwards would increase that distance. The agent walks upwards.
2. The path straight towards the goal is clear. The agent walks towards the goal.
3. The path towards the goal is obstructed again. Although from a global point of view, the goal is reached more quickly by going around the wall on the northern side, the agent will walk downwards, since that direction will bring it closer to the goal in the next couple of frames. The agent walks downwards.
4. The path straight towards the goal is clear. The agent walks towards the goal.

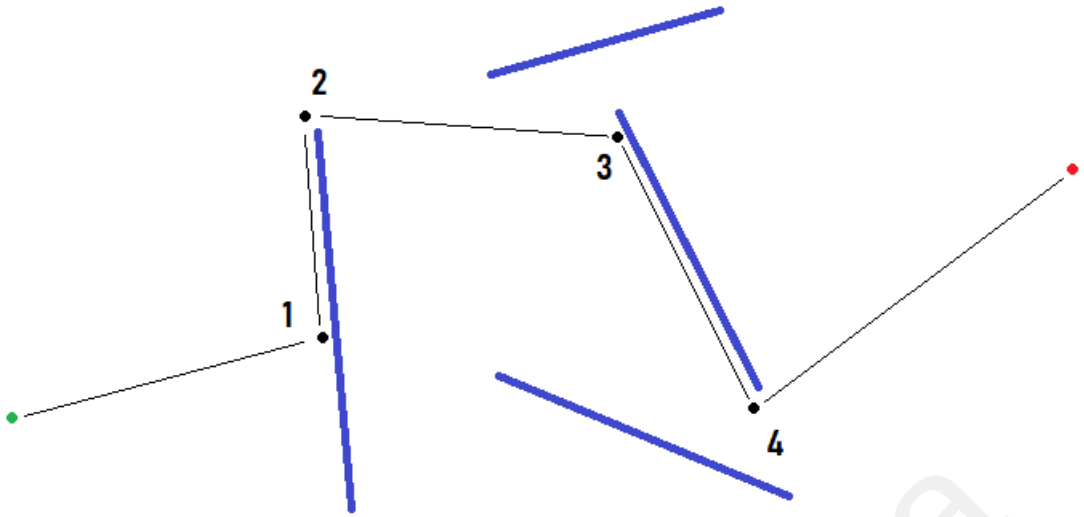

Figure 3.3: The path an agent follows when only knowing the direction it needs to walk into without environmental knowledge. The thick blue lines represent walls in the environment, while the thin black lines with enumerated nodes represent the path taken by an agent instructed to walk from the green starting point to the red ending point.

This form of pathfinding does not guarantee the shortest travel distance when walking from start to finish. It is, however, not unrealistic for people to find a route using similar decision making in certain situations. Besides that, this form of pathfinding changes the simulation dynamics compared to global shortest pathfinding and, therefore, could respond differently to error tolerance. One problem that is not solved for this form of pathfinding is the box problem. The box problem is when an agent walks into a wall and finds more walls in both directions of the wall. The blockage causes the agent to walk up and down alongside the first wall. Unable to find a way out, the agent gets 'boxed in'.

The box problem cannot be solved without the use of global information about the environment. Since that would change the whole premise of this type of pathfinding, these environments are altered. When an environment is used with such boxes in it, the boxes will be closed off before loading them in, using a line from one open end of the box to the other. Boxes are not detected and closed off automatically because this may negatively

affect floor plans where many lines are connected to form rooms..

When walking into a wall, the agent has two options; to walk along with the direction D of the wall, known in the `Line` object, or to walk along $-D$. The vector from the agent towards to goal is defined as $G = P_G - P_A$, where P_G indicates the goal's position and P_A indicates the agent's position. Then

$$\theta = \arccos(\frac{G \bullet D}{||G||||D||})$$

is the angle between these two vectors. Here $\bullet$ indicated the dot product. If $\theta < \frac{1}{2}\pi$, then the direction of D gets the agent closer to the goal in the next frame. If $\theta > \frac{1}{2}\pi$, direction D would get the agent further away, so then it needs to walk in direction $-D$. The agent then follows Algorithm 6 every step to find the velocity for the next step. The agent draws a line towards the goal and finds the closest intersection with another wall to figure out which wall the agent will bump. By saving this wall on normal frames, the agent does not need to do calculations for each wall in the environment. The agent only needs to look at the wall it is following, and the wall in front of it, for corners. In short, this pathfinding, like global shortest pathfinding, has time complexity $O(1)$.

3.9 Agents

This section presents the considerations and choices on implementing agents. The section includes subsections on how agents can be compared efficiently and on how agents avoid collisions. The agents' implementations also determine whether an error-preventive or an error-tolerant approach is used. For error-preventive simulations, the path finding algorithm does not change, but the agents use the methods described here in order to prevent parallel simulation of agents that are close to each other.

Algorithm 6: Finding a path towards the goal using only a compass

Result: The direction that takes the agent closer to the goal in the short run
Speed s;
if *Path towards goal is free* **then**
  return $s\frac{P_G - P_A}{\|P_G - P_A\|}$;
end
if *Current Velocity towards goal* $\wedge$ *path towards goal blocked* **then**
  if $\theta < \frac{1}{2}\pi$ **then**
    return $s\frac{D}{\|D\|}$;
  else
    return $-s\frac{D}{\|D\|}$;
  end
end
if *Can walk into current velocity direction* **then**
  return current velocity;
else
  return minus current velocity;
end

3.9.1 Nearby Agents

Agents have to compare their positions to other agents for several reasons. However, it is unreasonable for all agents to do calculations on all agents every single frame. This problem has been tackled in two different ways. The first is remembering nearby agents and the second is to use a two-dimensional array that sorts the agents every frame.

Remembering agents means that on frame i an agent checks for all agents how far they are, and stores which agents are close by for the subsequent x frames. Such activity raises the questions what x should be and how close by agents need to be in order to be remembered. There is an interesting dynamic here. If the circle around the agent marking the area in which other agents are nearby is tiny, then a new circle needs to be drawn quite often to deal with new agents entering the vicinity. In the most minimal case, the circle has radius $r_{\min} = 2R + 2S$, where R is the radius of the largest agent and S is the speed of the fastest agent. This is the smallest possible circle because it needs to be recalculated every frame to guarantee correct collisions. If the circle gets smaller and two agents are $r_{\min}$ away

from each other and walking towards each other, in the next frame, they will move so that both agents occupy some pixels. Recalculating the nearby agents every frame means every frame the time-complexity is $O(n^2)$. On the other hand, the maximum radius, $r_{\max}$, would be Max(`ScreenWidth`/2, `ScreenHeight`/2), since that includes all agents. The calculation means the agent checks for all agents every frame, since all agents are marked as "nearby". Therefore, it has the same time complexity as the smallest possible circle. Evidently, the optimum is somewhere in between. A conclusion that can be drawn already is that x is depending on the size of the circle. More precisely,

$$x(r) = \frac{r - 2R}{2S},$$

where r is the circle's radius. The question that remains is: What should r be? The conclusion after many tests is that the optimal r, r_{opt}, is variable itself. The time complexity of this method is not a function of r_{opt} directly, but of n, the number of agents. Every frame where all agents are checked, takes up $O(n^2)$ time and every remaining frame takes $O(na)$ time, where a is the number of nearby agents. Therefore, the total time remains $O(n^2)$ in this method since that is the largest time complexity, but is in practice vastly reduced, since it is derived from the time complexity of $(\frac{1}{x}O(n^2) + \frac{x-1}{x}O(na))$ per frame. Since n is given for any simulation and x is directly dependent on r, the only variable left in the time complexity that could help find r_{opt} is a. It makes sense to control this variable for more dense populations, as the circle will be smaller to deal with fewer agents per frame. A small algorithm within the `Update` function of an agent, Algorithm 7, adjusts the radius based on the number of agents counted as "nearby". With rapid changes in the crowd, it can take a few frames to adjust the circle correctly, but it will always get adjusted in the correct direction. Every agent gets a circle with a specific size, so agents in crowded areas

can have smaller circles than agents in spacious areas.

Algorithm 7: Adjusting the amount of agents marked as "nearby"

```
Result: Adjusted radius component R
target amount of agents t;
radius component R;
Calculate the nearby agents list based on distance and R;
if nearbyAgents.Count - t < -1 then
    R += 1;
end
else if nearbyAgents.Count - t > 1 then
    R -= 1;
end
```

The final remaining question is what the target amount of nearby agents t should be. Sadly, there is no universally correct answer to this question. Various scenarios have been tested for the performance of a range of values for t. The scenarios include no choke points, small choke points and a single large choke point. Various amounts of agents have also been tested. The results are shown in figure 3.4. The amount of agents in the graph legend means the total of agents active on the map. The variable t is the variable seen on the horizontal axis. Every value for t has had at least 50 frames and the graph shows the average time for each frame. This way the simulator has a few frames to adjust the radius r and still has plenty of frames to test this value for t. The lowest point on the average line is found at (49, 183.0). This means for all tested environments, 49 agents would produce the fasted combined simulation. However, not all test runs share the same number of agents for optimal simulation speed. The light blue line accumulates more agents in its choke point, meaning the total number of agents increases slightly over time. That does not explain the steep increase in computational time, but it does mean that it may not have gone off the chart near the end. The accumulation of agents in a choke point is a smooth process, so the sudden jumps in calculation time are not explained.

Note that the horizontal axis starts at 3. Simple interpolation from the graphs already shows that it would be non-sensible to evaluate lower values of t. Besides that, there is a practical side. Very close to zero, the algorithm does not work properly, since the radius can become negative. An explanation is that when the system adjusts r so much due to small amounts of nearby agents, the r becomes negative. The issue can be fixed, but that will not add any value to the simulator. For completeness, one environment with roughly 3,250 agents has been tested for its r_{opt}, r_{min} and r_{max}. Besides showing that $t < 3$ is not feasible, it shows the speed-up gained from using the nearby agents method. The results are as expected. For r_{min} the simulation averages 670 ms of calculation time per frame. For r_{max}, set at 1,000,000, calculating an average frame takes roughly 2739 ms. The r_{opt}, found when aiming for $t = 37$, results in an average calculation time of 122.8 ms. Clearly, this approach can speed up the simulations by a significant factor. Since the optimal r is different for each environment, it would be incorrect to assume that this method generally speeds up the simulation by a factor $\frac{670}{122.8} \approx 5.46$, but this test combined with the results shown in 3.4 does provide convincing evidence that a general speed up using this method is of a magnitude similar to the one in this single test case. The big difference between the performances for r_{min} and r_{max} has to do with the implementation of collision avoidance, explained in a later section. It is important to note that the relative speed-up using the nearby agents' method will increase as the number of agents increases beyond the current tests. The two extreme values for r result in a simulation that puts more weight on the complexity of n^2, while the optimal r would result in the most efficient combination of n^2 and an.

3.9.2 Tree Structure

Another approach in reducing the time complexity of agents comparing their position to that of other agents is using a tree structure. By dividing the map into tiles, the agents

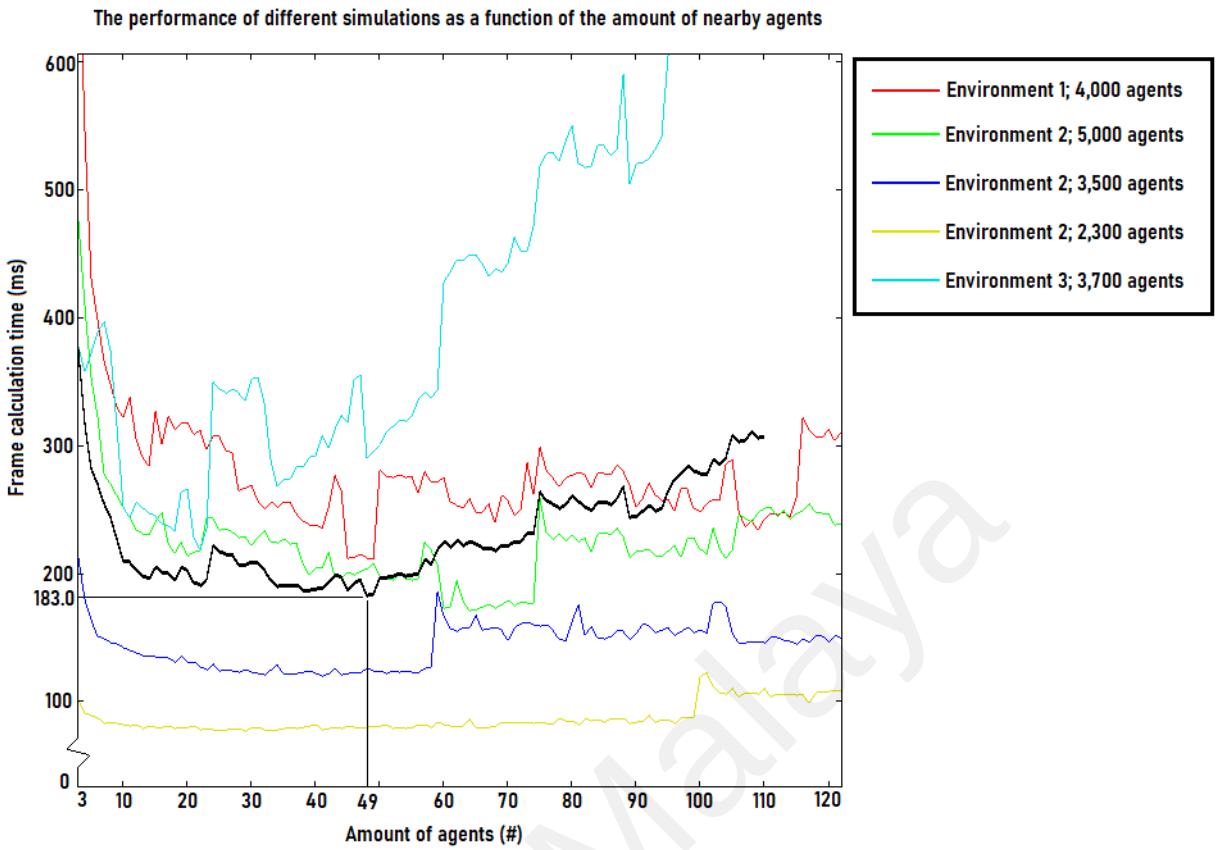

Figure 3.4: The performance of the nearby agents method under various circumstances. The thicker black line indicates the average of all colored lines.

can store their position into a two-dimensional array of tiles. Storing into tiles takes $O(n)$ time and when this is done, the agents can see which agents are close by in the array. As the tiles have finite area, there is a constant finite number limiting how many agents are in the tiles around an agent. The calculation time complexity for every frame is then $O(n)$. This also becomes clear from algorithm 8. The scenario does not necessarily mean "case closed" for the nearby agents' method, which has higher-order total time complexity. Indeed, when the number of agents is exceptionally high, say 1,000,000 agents, there is no doubt that $O(n)$ is faster than $O(n^2)$ no matter the composition of these time complexities. However, these agents are not feasible to simulate on a regular computer and on lower amounts of agents. There are scenarios where the overhead of creating the two-dimensional array and all the lists takes the total time higher than the

nearby agents method. The overhead quickly becomes clear when the tiles are reduced to single pixels. The example is too small for an accurate simulation, but shows how too much overhead can lead to a decrease in speed so large that the time complexity of the simulator is no longer the bottleneck. The case of tiles of size 1x1 would make a simulation of just 300 agents run at around 3 fps. Clearly, like before, there is an optimal tile size in terms of speed somewhere between the minimum tile size and maximum tile size.

Algorithm 8: Updating all agents

```
Result: Updates agents
Create 2D array of lists of agents;
foreach Agent a do
    Add a to correct list in the array;
end
foreach Agent a do
    Make list of agents in tile of a and the 8 surrounding tiles;
    Calculate collisions with agents from this list;
end
```

The array divided the environment into tiles of $3d$ width and height, where d is the maximum diameter of an agent. There is some margin of error, as the furthest agents can be apart and collide in a parallel simulation is $2(r + s)$, where r is the largest agent's radius and s the fastest agent's speed. In practice, to account for rounding errors, $2r + 3s$ would be appropriate to ensure correctness even in exceptional cases.

The best optimal size for simulation speed is not constant, just like r_{opt} for the nearby agents method. In contrast, the optimal tile width is much more predictable. The time to set up the array of tiles and fill it with empty lists is constant for a given tile width; call it $C(w)$, where w is the tile width. Of course, as w gets smaller, C gets bigger, since there are more tiles and lists to make. This part does not scale with the amount of agents n, except where agents add themselves to the appropriate list and check for collisions scales with n. It does so

regardless of C; each agents must add itself to some tile, no matter the total number of tiles. What decreases as the tiles get smaller and C gets larger is the time to check for collisions. After all, the number of tiles containing agents that need to be checked is always 9 and as the tiles are smaller, there are fewer agents in them. This effect gets stronger as n increases, because the more agents are in the environment, the more agents will be in each tile and the more strict the tile size needs to be in order to have fewer comparisons to make. Let $V(w)$ be the variable time needed for an agent to calculate its collisions. Obviously, as w increases, this variable time increases too, since more agents need to be considered for collisions. The case is true for any amount of agents, but it still scales with n for the simulation as a whole, since each agent needs to spend $V(w)$ calculation time on its collision detection.

Combining these findings, the total time to update all agents is

$$T(w, n) = C(w) + nV(w). \tag{3.5}$$

Now it becomes clear that if w is fixed, say $w = w_0$, then the total time will be $T(w_0, n) = C(w_0) + nV(w_0)$. Now, if a $w_1 < w_0$ would decrease the simulation time, that would mean

$$C(w_1) + nV(w_1) < C(w_0) + nV(w_0),$$

which implies

$$n > \frac{C(w_1) - C(w_0)}{V(w_0) - V(w_1)}.$$

Since C increases as w decreases, $C(w_1)$ must be bigger than $C(w_0)$ and as V increases as w increases, $V(w_0)$ must be bigger than $V(w_1)$. The conclusion is that both the denominator and numerator are positive and the inequality yields consequently that there

is an n for which $T(w_1, n) < T(w_0, n)$. If w_1 would be the optimal tile width, it would also mean that for any $w_2 < w_1, T(w_2, n) > T(w_1, n)$. This means

$$C(w_1) + nV(w_1) < C(w_2) + nV(w_2)$$

and combined with the inequality for w_0, it can be established that

$$\frac{C(w_2) - C(w_1)}{V(w_1) - V(w_2)} > n > \frac{C(w_1) - C(w_0)}{V(w_0) - V(w_1)},$$

where again both bounds are positive. This means that if such an n exists for any bounds around w_0, that w_0 is the optimal w as long as the simulation has n agents where n is within the given bounds. It is trivial that there is an optimal w for each n, since for any w there is some computation time and such a list of computation times must have one or more smallest numbers. From experiments, it becomes clear that the computation time for a simulation of n agents as a function of w is not a flat graph at any point, and therefore there is a single optimal w. For this reason, there is a w_{opt} for each n, but not every w necessarily has an n such that is is optimal. Every w likely has an n such that it is the optimal tile width, but this cannot be concluded from the proposed time analysis. The exact form of the time given by function 3.5 must be known for such a conclusion, and this function is unknown. Only its characteristics are known. What can be said, however, is that if w_1 is optimal for n_1 agents and w_2 is optimal for n_2 agents ($w_1 \neq w_2$), then it must be the case that

$$\frac{C(w_3) - C(w_2)}{V(w_2) - V(w_3)} > n_2 > \frac{C(w_2) - C(w_1)}{V(w_1) - V(w_2)} > n_1 > \frac{C(w_1) - C(w_0)}{V(w_0) - V(w_1)}$$

for any $w_3 < w_2$, hence $n_2 > n_1$. This is important, as it shows that w_{opt} decreases as n

increases. Experimental data, as displayed in figure 3.5, support these findings. The lower amounts of agents are simulated up to 42 pixels width, while the higher amounts of agents had up to 12 pixels width, which translates to 14 meters and 4 meters respectively in the real world. The trends beyond 12 pixels are upwards, and updates in that range take significantly more time for big crowds. Therefore, no minima were expected there for the larger crowds. For this experiment, parallel threading was used with four parallel threads to speed up the process. Since the speed was the variable of interest here and is only dependent on w and n, (parallelization) errors did not matter during these tests. However, those errors did occur, especially with very low w, where agents did not calculate collisions correctly or at all anymore. As error tolerance during the parallel simulations later on is relevant for w smaller than the previously mentioned $(2r+3s)$, it is for those tests pivotal that taking w that low actually increases the speed at all. In a later section, these tests will be explained in more detail.

Now that the nearby agents method and the tree structure are explained, they need to be compared. A graph is not suited for this, as there are multiple variables to control for. Since the time measurements are real-time and depend on many variables out of our control, such as Windows background processes and misfortune with specific agent formations during the simulation, the time measurement for each frame can show bumps and the averaged time measurement can have local minima. This makes dynamically adjusting the r in the nearby agents method and the w in the tree structure very difficult. This would still have been implemented to create a graph showing the methods in a comparison that is *ceteris paribus*, if it were not for the fact that it was not necessary. Simulations with 3,000 agents in a medium-sized environment that has no choke points showed that the tree structure was on average 37.6% faster than the nearby agents method. Furthermore, this difference would be 12.3% on average for a large-sized environment with one choke point. For large

crowds (>10,000 agents), the tree structure was typically two to three times faster than the nearby agents method. The tree speed does not mean that the nearby agents' method has no value. For test runs with around 1,800 agents on a tiny map with a single choke point, the nearby agents method was in fact 20-50% faster than the tree structure. However, this environment was created so that it would have the least favorable conditions for the tree structure and was quite unrealistic as a real-world example. Results obtained from these tests mainly illustrate that it can be worthwhile to test both methods for smaller crowds.

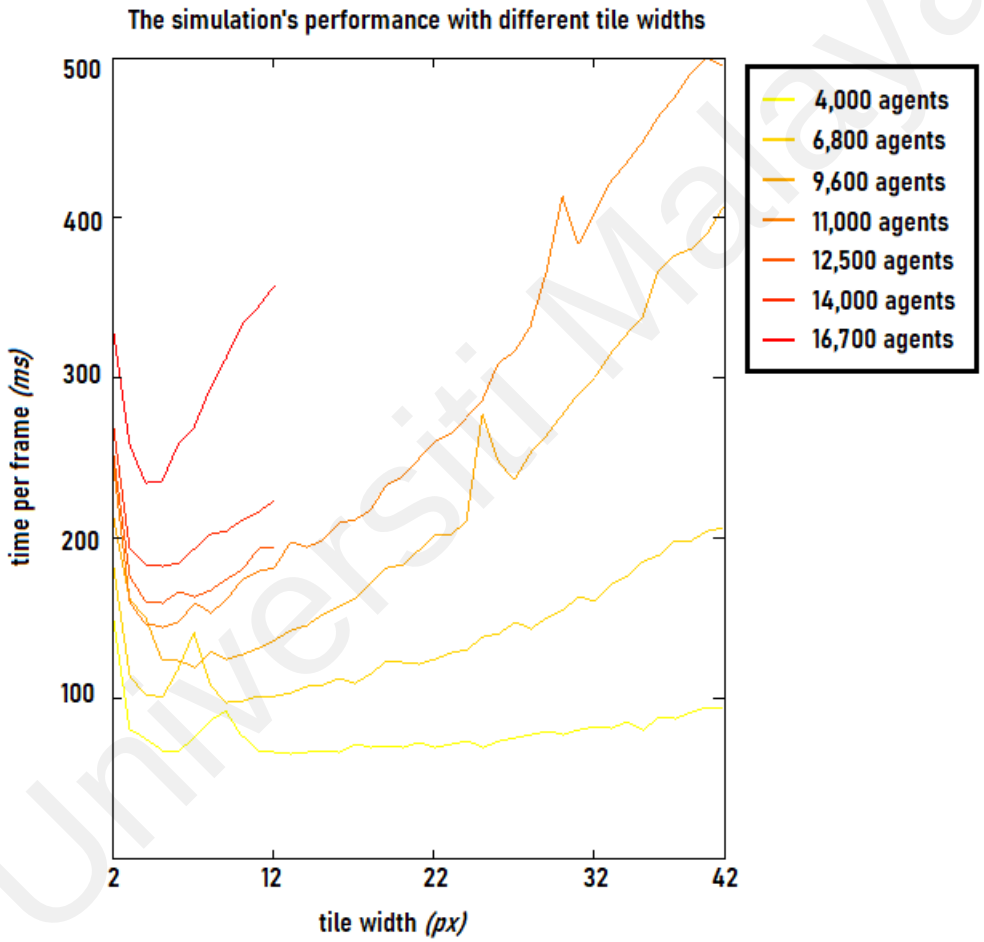

Figure 3.5: For each amount of agents, the tile width has different influence on the calculation times. The minima show a clear tendency to be smaller as the amount of agents gets bigger.

One could also pose the idea to combine both methods in order to get even faster simulations. The idea would be to have larger tile width or look at agents within a 25 tiles radius so for each frame, more agents can be saved and used for multiple frames. Since an extra

frame would allow agents that are an extra $2s$ distance away from the current maximum reach, the tiles would need to be at least $2s$ extra wide. For every $2s$ added to the tile width, one extra frame of building a new tree structure can be skipped. The frame has been tested and it turns out that adding a multiple of $2s$ to the tile width and then skipping that amount of frames building new arrays results in the same simulation speed within a deviation of about 2 ms, which is not significant. The explanation for this is not too difficult. As the tiles get bigger, the work to set up the tree structure gets quadratically less, as the number of tiles gets less in two dimensions. At the same time, the number of agents found within the collision range increases in two dimensions and therefore increases quadratically. The effect of these two consequences of increasing the tile size is that they cancel each other out. There is one way this trick could be used, to realize a small speed-up. When increasing the tile width, the work needed to build the two-dimensional array decreases with x^2 where x is the length of width of a tile (they are squares). Still, the agents do not necessarily need to check the corners since those are slightly further away than the outer-most tiles directly up, down, left or right from the middle tile. To be more precise, an average agent on a corner tile is $\sqrt{2}$ times further from the centre than an average agent on an outer-most tile directly upwards. A circle could be drawn that checks whether corner tiles are within collision range within the subsequent few frames.

When seeing the grid as a continuous field with infinitesimal tile sizes, this trick would mean the agents do not need to remember exactly the amount of surface area saved by using bigger tiles. If the surface of the tile surrounding an agent is $4r^2$, the agents only need to remember agents in an area of πr^2, reducing around 21%. This situation is ideal and only occurs when the tile size relative to the collision range approaches zero, as seen in figure 3.6. Note that 25 tiles are not enough to skip any tile at all, so at least 49 tiles

would be needed to even cut off the four corner tiles. Since the minimal tile size must be larger than 4, that means the collision radius for 49 tiles would be very large, while the speed-up is fairly minimal. An even larger amount of tiles needs to be considered to reach the 21% reduction, which is not feasible..

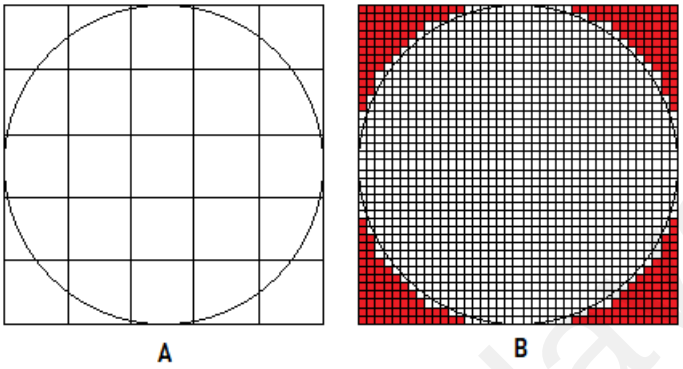

Figure 3.6: Dividing the environment within collision range of the middle tile into differently sized tiles. A: Division into 25 tiles; no tiles are outside collision range. B: Division into about 1,600 tiles; on the corners there are a number of tiles outside collision range.

3.9.3 Collision Avoidance

During the simulation, a lot of agents bump into each other. Just telling an agent not to move forward when another agent is in the way is not sufficient to solve this issue. At the same time, this issue had to be solved since agents who walk through each other are not realistic and require no collision checks. These checks are a big reason for crowd simulations to be as slow as they are for large amounts of agents. Therefore, implementing collision avoidance is a vital part of the simulator used to test the research questions. Two simple steps make the simulation of single agents realistic. The first is to stop an agent from walking into a position that another agent already occupies. If an agent is obstructing the way, the second is to find a way around that agent. The second step does not just have to do with agents reaching their goal faster. Figure 3.7, with a special focus on the zoomed-in areas, shows how queues form when agents do not look for ways around obstacles. Queues may be a

nice feature when simulating places where people always queue up for everything, but in general, this is not realistic behaviour. The top simulation looks far more natural and realistic because the agent behaviour is more complex, and the crowd interacts more dynamically.

Figuring out whether agents will run into other agents is not so difficult. The following is a simple comparison of their new position with positions of nearby agents. When updating agent a, if

$$\|(p_a + v_a) - p_b\| > r_a + r_b, \forall b \in \texttt{nearbyAgents}, \tag{3.6}$$

where v_a is agents a's velocity, p_i is agent i's position and r_i is agent i's radius, the agent can move freely to the next position.

Finding a new direction is slightly more challenging. In order to get closer to its goal, an agent walks into the direction of its velocity v. The agent can still get closer to its goal as measured on at least one or two axes by deviating from this direction by no more than $\frac{1}{2}\pi$. When an agent b has been identified as being in the way, algorithm 9 adjusts the velocity so that the best possible alternative is chosen.

As seen in the algorithm, when no valid velocity can be found, the agent will have to wait and its velocity consequently becomes zero. The potential danger of this method is that agents who walk into opposite directions will move in one direction as a pair of agents, constantly adjusting their velocity and eventually reaching an impasse. To prevent this event, if agents a and b are colliding and

$$v_a \bullet v_b < 0,$$

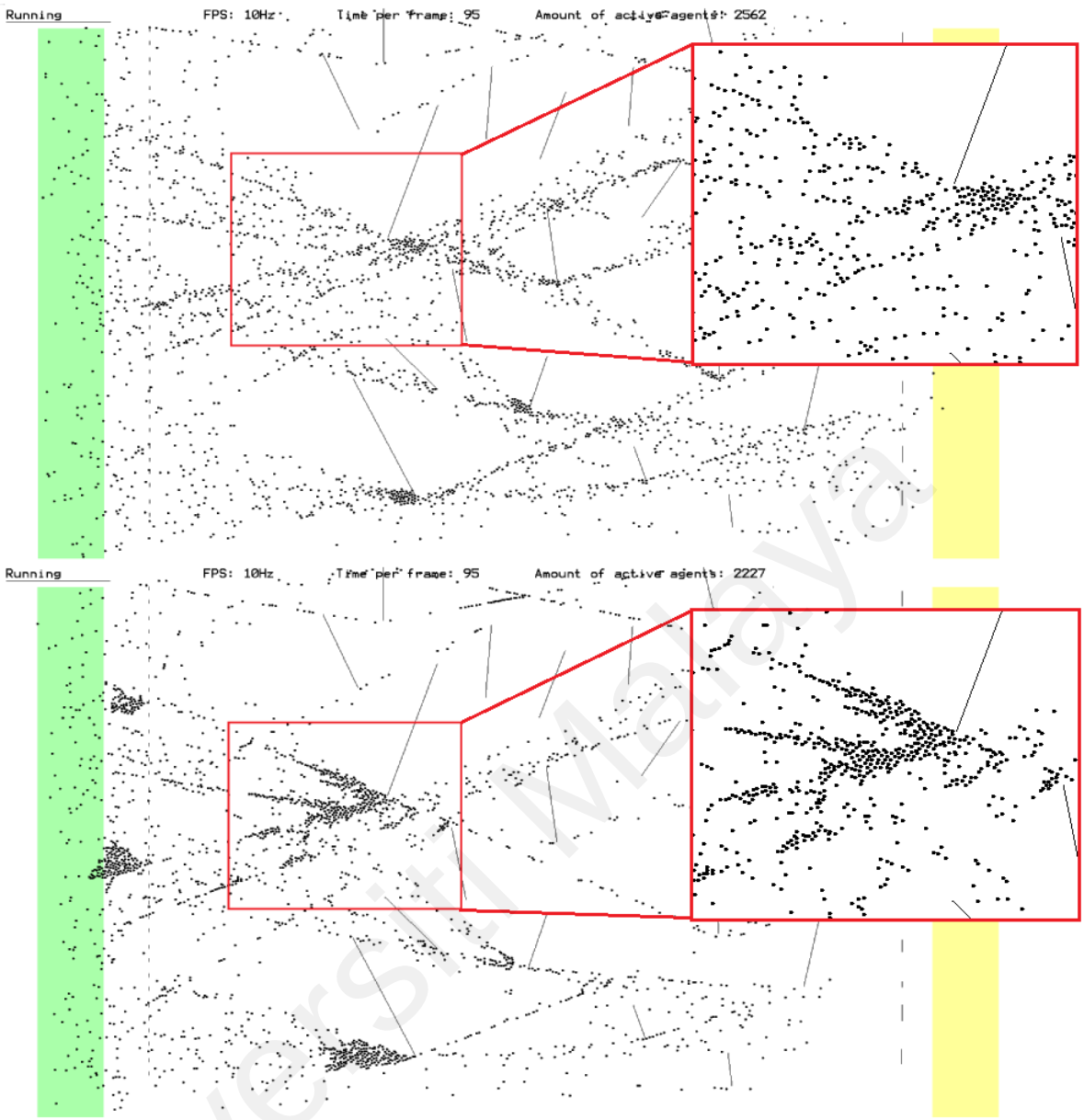

Figure 3.7: One simulation where agents walk around agents that are in their way (top) and one where they just stop if an agent is in their way (bottom).

then the agent that first figures this out freezes the other agent for a few frames to walk around that agent. A comparison to a real-life situation is where two people want to pass each other, they move out of the way for each other to the same side, blocking each other again. In the real world, this almost always leads to one person waiting for the other to walk around them. The same thing happens in the simulator when this rule is applied.

Although with all rules applied, agents should not be able to occupy any pixels more

Algorithm 9: Adjusting the agent's preferred velocity to a velocity leading a legal move

```
Result: Adjusted velocity
Get preferred velocity V;
if condition from equation 3.6 then
    return velocity V;
end
for i = 1/20 π; i ≤ 1/2 π; i+=1/20 π do
    newV = V;
    Adjust newV by i in clockwise direction;
    if condition from equation 3.6 with velocity newV then
        V = newV;
        return velocity V;
    end
    newV = V;
    Adjust newV by i in counter-clockwise direction;
    if condition from equation 3.6 with velocity newV then
        V = newV;
        return velocity V;
    end
end
return V = (0,0);
```

than once, this situation could still occur due to rounding errors or parallelization errors. Although the goal is to find the impact of these errors, the goal is not to get this impact as large as possible. Therefore, when these errors occur, they are solved on the spot. When the velocity of an agent becomes zero, it will check if it is intersecting with another agent. If agent a finds itself intersecting with agent b, agent a moves to position

$$P_a = (r_a + r_b)\frac{a-b}{\|a-b\|}.$$

Such a case could trigger a chain reaction in a choke point, but since these errors will always be very small, this chain reaction hardly disturbs the simulation. Besides that, fixing the error must be done since not fixing it disturbs the simulation greatly. The error fix also does not need to happen often since these errors are rare and are unlikely to occur again immediately after being fixed due to the nature of their causes (unfortunate rounding

of very close values or parallelization).

3.10 Spawns and Goals

Spawns and goals are areas on the map where agents spawn and despawn, respectively. The choice for an area rather than a point makes it easier to spawn agents consistently, even if the crowd gets so large that agents get stuck on top of the spawn area. The spawn has a buffer with agents to spawn that gets filled with every frame's given spawn rate. By having a spawn rate in agents per frame rather than agents per second, massive spawns are prevented when the frame rate gets low, and the simulation becomes more deterministic. As a result, it is easier to compare different simulation settings. On frame x, no matter the simulation settings such as pathfinding or number of threads, the number of agents on the map will be the same. Still, a floating-point number can be used for the spawn rate. The spawn buffer gets filled with the spawn rate at every frame. Then the spawns, as many full agents as there are in the buffer. Any leftover decimal number stays in the buffer and stays in the buffer for the next frame.

Spawns have a list of diameters, speeds and goal areas. Upon creating an agent, the agent is assigned a random diameter d, speed s and goal G from those lists. Then a random position is chosen within the spawn area. If the position is free, the agent is spawned. If not, a new position is chosen. This process is repeated up to 10 times. If by then no valid position has been found, the agent is cancelled. Without a maximum amount of attempts, an infinite while loop would be possible. With all this randomness, it is hard to claim that simulations are deterministic and comparisons are fair. For that reason, the randomizer used in the simulator is seeded. The first simulation is still completely random, using a random seed. Nevertheless, manually assigning the same seed to comparing simulations will show the same actions as the first. Once an agent has been successfully spawned, it

will receive the first node on its path towards its goal. Every agent also gets its unique ID. Using a static integer to store the ID counter and updating spawns in series instead of parallel, every ID is guaranteed to be used at most once.

3.11 Parallelization

Parallelization has been implemented in different ways, all of which use the namespace `System.Threading` from the Task Parallel Library. The most basic implementation is a simple parallel for-loop that divides all agents in as many groups as there are threads ordered by ID, the default ordering, and updates all groups simultaneously with no regard to anything else. This method requires the least overhead but has no error prevention. The second method sorts the agents based on their x coordinate, taking $O(n)$ time. n is the number of agents and has to be done each frame since each frame agents' x coordinates change. After this sorting, the agents are simulated in parallel in two rounds. A thread gets to simulate all agents with an x coordinate between x_0 and $x_0 + c$, where x_0 is a starting coordinate assigned to the thread when starting the updates and c is a constant for all threads. In round one, threads are assigned x_0s so that there is always a space of $2c$ between their assigned starting coordinates. Then in round two, the remaining agents are updated similarly in parallel. This process is also shown in figure 1.3 in the introduction. For large enough c, this implementation is parallelization error-proof. The last method is similar to the second, but is used when there is a tree structure in place already. That tree structure has already sorted the agents by x and y coordinates, so the update method can use these to plan the two rounds of parallel updates. As the y coordinate is not necessary for this planning, all lists of agents in the same x bounds are updated by the same thread, regardless of their y value.

One could argue that method two is the least efficient, because it has to sort the agents first,

where the other two methods don't. However, this is not entirely accurate because building the tree structure takes more time than sorting the agents by x coordinate. Method two is only used when the tree structure is not present, and therefore simulations using method two are faster in terms of total overhead. All three methods serve a specific purpose when testing the different scenarios and one cannot be substituted for another if the simulation has to be fair. For example, when there is no tree structure present, using method three would require building such a structure first, which makes the simulation artificially slow and results in biased test outcomes.

3.12 Characteristics of Simulations

Through the literature review and many test simulations, characteristics that define a simulation course have been identified and implemented. They assist in finding the conclusions to the research questions in the end.

3.12.1 Heat Map

Generally speaking, there is almost a bijection between a simulation and its heat map. Choke points and paths are all clear characteristics for a simulation, since they give fast information about the paths taken by the crowd and the color intensity also shows which paths are used the most. As the heat map is normalized, it shows how crowded areas are compared to other areas regardless of how long the simulation runs. There is also an aesthetical aspect to the heat map. It is an elegant and appealing way for a user to see a lot of information quickly.

3.12.2 Walking and Waiting Time

Agents have a counter for how long they have been on their way and how much time they have spent waiting. These are separate statistics. It is not true that the shortest path to get from A to B is calculated from the total walking time minus the waiting time. The

agents walking around others and are doing other manoeuvres to get closer to their goal are considered walking and not waiting. The waiting time gives clear information about agents getting stuck in queues and choke points. In contrast, the total time travelled gives information about waiting, walking straight towards (local) goals and avoiding collisions. Specifically, the latter could change average walking times under circumstances where collision detection and pathfinding are variable. When spawning a fixed number of total agents, the spawn rate can be vastly reduced to create a simulation where the shortest travelling times can be found. Comparing that number with the normal spawn rate and taking waiting times into account, the average number of frames spent avoiding collisions can be found. These statistics are important, as they can show both averages as well as absolute numbers. A heat map shows how crowded areas are relative to each other, meaning that an increased spawn rate on the same map may result in a very similar heat map, as long as no new choke points occur. The total time spent by agents and the waiting times can therefore assist in further characterizing specific simulations.

3.12.3 Counting Lines

Counting lines are a good way of finding the stream of people through specific areas. Especially streams of slightly different people in agents per second can be tricky to differentiate using a heat map. They may not show up on total waiting and travel times when these differences cancel out on the simulation as a whole. They are still important because they show from which side of the environment the most stress on a door or hallway comes. It gives the user the ability to search for a solution to a choke point further upstream. Together, heat maps, travel and waiting times, and counting lines complete a bijection between simulations and simulation statistics. The combination of these statistics can be used as a fingerprint of the simulation. In computer science terms, as a hash of the simulation. This unique information means when testing for different simulation settings,

the accuracy of the simulations can be examined thoroughly.

3.12.4 Simulator Performance

Specific simulations' performance can be measured relatively easily. The average calculation time for a frame can be measured using the Windows `StopWatch` class. This calculation is the primary measurement for simulation performance. A margin of error needs to be taken into account, since not every aspect of the computer can be accounted for. When doing nothing, a computer uses hardly any processing power, so this margin of error will be far smaller than any significant performance differences in the simulator. The simulator's performance can also be measured in memory usage. So far, in not a single test run memory usage has gotten to levels where this becomes an interesting factor, so the memory usage displayed by Visual Studio's diagnostic tools is not reported in this dissertation. Perhaps in massive maps (e.g. a city) and a million agents this must be taken into account, but that situation is not within the scope of this dissertation. It can be argued that computers capable of massive calculations will have access to a lot more memory and, with some intelligent memory management, can also handle such large amounts of data without much difficulty.

3.13 Stats Generator

The statistics generator is where statistics are ultimately collected and on-demand exported into the appropriate files.

3.13.1 Heat Map

Agents tell the generator their positions in each frame. When that happens, that particular point on the map gets three points in the Heat Map array. The coordinates directly around them get 2-point, and the layer of coordinates around that gets 1 point. When the heat map is exported, every element in the heat map gets normalized and multiplied by 255. This

data is then converted into `Color` objects using the map

$$h \longrightarrow \texttt{new Color}(255, 255 - h, 255 - h)$$

and is saved as an image using the `System.Drawing` library. The heat map always contains one or more perfectly red using this implementation and frequently used paths that will show clearly yet less red as there are spaces between the agents. Areas where agents walk more freely still show up marked but are less red than frequently used routes. Figure 3.8 shows the result when using this method. The paths are still clear and areas where agents do not follow a path but cross the area at will. The reddest area clearly shows where the choke point is on this map.

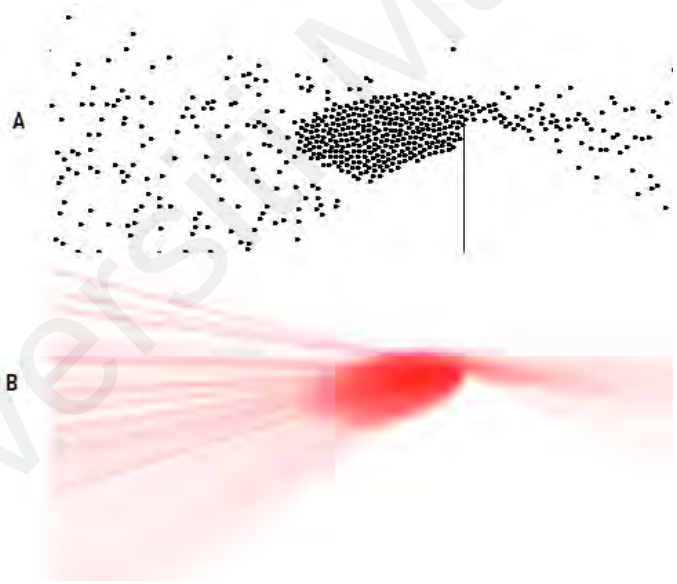

Figure 3.8: The generation of a heat map. The situation in A as input for the stats generator results in the heat map in B.

When comparing simulation accuracies, it could be the case that the heat map does not give the information that was looked for, because shades of red that are close to each other visually look the same. For this reason, the differences could either be squared to amplify the errors, or a difference heat map could be made. A difference heat map would be a heat map that is subtracted from another and then normalized. If there are any differences

at all, they will be well visible as a result of the normalization. The differences are not normalized for the raw data since they are not drawn but reported as numbers among the simulation data.

3.13.2 Walking Times and Counting Lines

As mentioned previously, the heat maps are assisted by counting lines and waiting and walking times of agents. Simple statistics, such as calculating the mean and the standard deviation, are applied to these times to give an accurate and complete view of these two metrics. The counting lines keep a sorted list of agent IDs for agents that have passed the counting lines. Since this list is sorted, a quick binary search can determine whether an agent has been seen and where to add a new ID. The size of this list shows the number of agents that passed the line. Doing this for every agent and using binary search amounts to a total of $O(n \log n)$ time, where n is the number of agents.

3.14 Simulation Environments and Settings

There are two clear types of agent movements during a simulation; one gets stuck early on and finds a clear path for a large part at the beginning. Agents who get stuck on choke points early on often have much higher waiting times, and agents who do not, usually have close to zero waiting time. The scenario brings us to the first criterion for environments. The environments have varying degrees and amounts of choke points to test the simulation accuracy for given simulation settings. The environment setup includes environments with several walls and a funnel environment, no choke points, limited choke points, and one massive choke point can be created, respectively. Some environments are also tested with spawn and a goal on each side to increase collisions dramatically and see collision impact better.

Pathfinding is a global setting and can be set to compass pathfinding or global pathfinding following the user's preference. Furthermore, the update method can be chosen as well as the tile size for the tree structure. Other settings that can freely be chosen are the agent size and speed, including size and speed distributions instead of a fixed size and speed. The last setting of importance is the number of parallel threads. The number can be any positive number, and when a number is chosen, the operating system divides the computational power of all cores over all the threads. The number of threads can be more than the number of threads on the CPU, and that will not increase speed further since there are no extra cores available for those threads.

3.15 Chapter Summary

This chapter gave an extensive description of the simulator development and how the tests have been performed. Various techniques have been used to make sure the pathfinding and collisions work correctly and quickly. Some techniques are made from scratch, such as compass pathfinding, while others modify exiting techniques, such as the customized Dijkstra algorithm. All combined, they form a simulator that allows for all the desired settings and can be error-tolerant as well as error-preventive. The simulator has been tested extensively to ensure simulations work as intended and calibrate the simulator to be deterministic when it should be. Elements for statistics generation have been added, and statistics are all reported in a heat map and text file in a reader-friendly format. All these statistics are found and reported in the next chapter.

CHAPTER 4: RESULTS AND DISCUSSION

4.1 Overview

This chapter presents all results necessary to answer the research questions. First, the characteristics that describe a simulation are described. Then, some important results from testing different pathfinding methods are presented and explained. After these pathfinding results, the test results that have been included in Appendix A are explained and the chapter ends with a section that discusses design and research choices.

4.2 Characteristics of Simulations

An objective in this study was to identify the characteristics that quantify the accuracy and speed of a simulation. The literature review and the development of the proposed simulator have provided a clear set of such characteristics. These characteristics are the basis for the final conclusion on the usefulness of error-tolerance in parallel crowd simulation.

4.2.1 Heat Map

A heat map is a picture of the size of the simulation environment. The map pixel is coloured with shades of red, marking the presence of agents on that pixel throughout the simulation. The heat map does not have a unit, as it is composed of unitless quantities. Generally speaking, there is almost a bijection between a simulation and its heat map. Choke points and paths are all evident characteristics for a simulation since they give fast information about the paths taken by the crowd and the colour intensity also shows which paths are used the most. As the heat map is normalized, it shows how crowded areas are compared to other areas regardless of how long the simulation runs. There is also an aesthetical aspect to the heat map. It is an elegant and appealing way for a user to see a lot of information quickly.

4.2.2 Walking and Waiting Time

A counter on each agent monitors how long they have been on their way and how much time they have spent waiting. As they should be compared fairly and independent of simulation speed, these times are not measured in $seconds$ but $frames$. These times are separate statistics. It is not true that the shortest path to get from A to B is calculated from the total walking time minus the waiting time. The agents walking around others or that are doing other manoeuvres to get closer to their goal are considered walking and not waiting. The waiting time gives information about agents getting stuck in queues and choke points. In contrast, the total time travelled gives information about waiting, walking straight towards (local) goals and avoiding collisions. Specifically, the latter could change average walking times under circumstances where collision detection and pathfinding are variable. When spawning a fixed number of total agents, the spawn rate can be vastly reduced to create a simulation where the shortest travelling times can be found.

Comparing that number with the normal spawn rate and taking waiting times into account, the average number of frames spent avoiding collisions can be found. These statistics are important, as they can show both averages as well as absolute numbers. A heat map shows how crowded areas are relative to each other, meaning that an increased spawn rate on the same map may result in a very similar heat map, as long as no new choke points occur. The total time spent by agents and the waiting times can therefore assist in further characterizing specific simulations.

4.2.3 Counting Lines and Simulator Performance

Counting lines and `StopWatch` instances to measure accuracy and performance respectively have been discussed in the methodology. The combination of these two and the previously mentioned measures are enough to clearly and extensively test and answer the research

questions.

4.3 Pathfinding

While the objective of this study is not to figure out the most realistic type of pathfinding nor finding out if one would be superior to another, it is good to have a baseline understanding of how different they are when controlling for all other factors. These methods have a clear set of objectives that are the same or different. These different objectives have been described previously, but one part that has not been highlighted is the path length. The global shortest path algorithm finds the shortest path, but that does not mean that compass pathfinding is not meant to be fast. The lifetimes and waiting times of agents are processed to measure the accuracy of simulations.

These stats can also be used to see the quantitative differences between the different types of pathfindings. It is important to note here that the preferred path is based on the environment only, not on other agents. Significantly, waiting times are the only stats that get influenced by choice of pathfinding indirectly because different types of pathfinding may create different choke points on the map. The preferred route agents try to take hardly changes as a result of large amounts of agents. A total of six tests are conducted, and three environments have been created:

1. one with no walls obstructing the way (E1),
2. one with a number of walls, creating a number of preferred paths and potential choke points (E2)
3. and an environment where only one route is optimal for agents to follow, potentially creating one massive choke point (E3).

For each environment, a spawn rate of y is chosen and a total amount of x agents are

spawned and the spawn is capped at that amount. For each environment, two pairs (x, y) are chosen. The spawn rates y used for each environment are 1 and 4. Spawn rate 1 creates a relatively peaceful simulation where not many collisions and choke points occur and spawn rate 4 is bound to create some problems in environments where walls are present. x is then adjusted so that there is a short time, called *busy time*, where agents are present across the environment, from start to end. When this situation is accomplished, the spawn stops so that the simulation comes to an end as the walking agents reach their destinations. The number of agents present in the simulation during busy time could be different for both types of pathfinding, but for a fair comparison, x needs to be the same for both simulations. Therefore the largest x is chosen, meaning both test runs get at least a few seconds of busy time. Busy time is kept short because the length of a simulation and potential choke points do not change the pathfinding and therefore more busy time would add nothing to the simulation for these tests in particular. The results are shown in Table 4.1, where μ_{LC}, μ_{WC}, μ_{LS} and μ_{WS} represent the average lifetime using compass pathfinding, waiting time using compass pathfinding, lifetime using shortest path and waiting time using shortest path, respectively.

Table 4.1: The lifetimes and waiting times including their standard deviations of both types of pathfinding.

Life and waiting times in frames for each simulation *(environment, amount of agents)*						
	(E1, 1400)	(E1, 5500)	(E2, 1500)	(E2, 6000)	(E3, 1500)	(E3, 5750)
μ_{LC}	1385 ± 47	1443 ± 91	1726 ± 341	3106 ± 1555	1607 ± 238	3172 ± 1557
μ_{WC}	0 ± 2	15 ± 56	206 ± 277	1466 ± 1393	165 ± 197	1592 ± 1425
μ_{LS}	1382 ± 49	1419 ± 60	1450 ± 100	2717 ± 1748	1449 ± 64	4417 ± 1938
μ_{WS}	0 ± 0	3 ± 4	20 ± 66	1116 ± 1624	8 ± 18	2743 ± 1806

When using the compass method, a large amount of waiting time results from a phenomenon we call *wall hugging*. When many agents bump into a wall, they try walking to the side. However, the side is occupied due to the number of agents bumping into the wall. When more agents are walking towards the wall than the wall length divided by the agents' diameters, the agents will get stuck on each other. Some will walk around the crowd due to agent avoidance, but many get stuck waiting until they can continue walking. The situation is not a flaw in the pathfinding algorithm but a mere consequence of agents trying to get closer to their goal (or as far away from a health hazard). In the event of a fire, it is not uncommon to see people clump together or run away from the fire without paying much attention to where they are running.

Many of the numbers in the table also have a high standard deviation. The result is likely due to the big difference between agents who do not get stuck in choke points but getting a much lower than average lifetime and waiting time and agents who get stuck in choke points with higher life and waiting times. Therefore, the most crucial factor is the mean since the test runs for each simulation have the same total amount of agents, and each agent has the same spawn position and goal position. The standard deviation is more a measure of the environment and choke points than pathfinding accuracy.

It is clear from the test runs that global shortest pathfinding is not vastly more effective than compass pathfinding in superficial scenarios, however. It is faster, but by margins in the neighbourhood of 15%. The heat maps show that the relatively close travelling and waiting times are not a result of similar paths. For the sake of space and because of the focus of this section, which is not the focus of this thesis, not all heat maps are included. Figure 4.1 shows the heat map for environment 2 with 1,500 agents. A metric quantifies the heat map

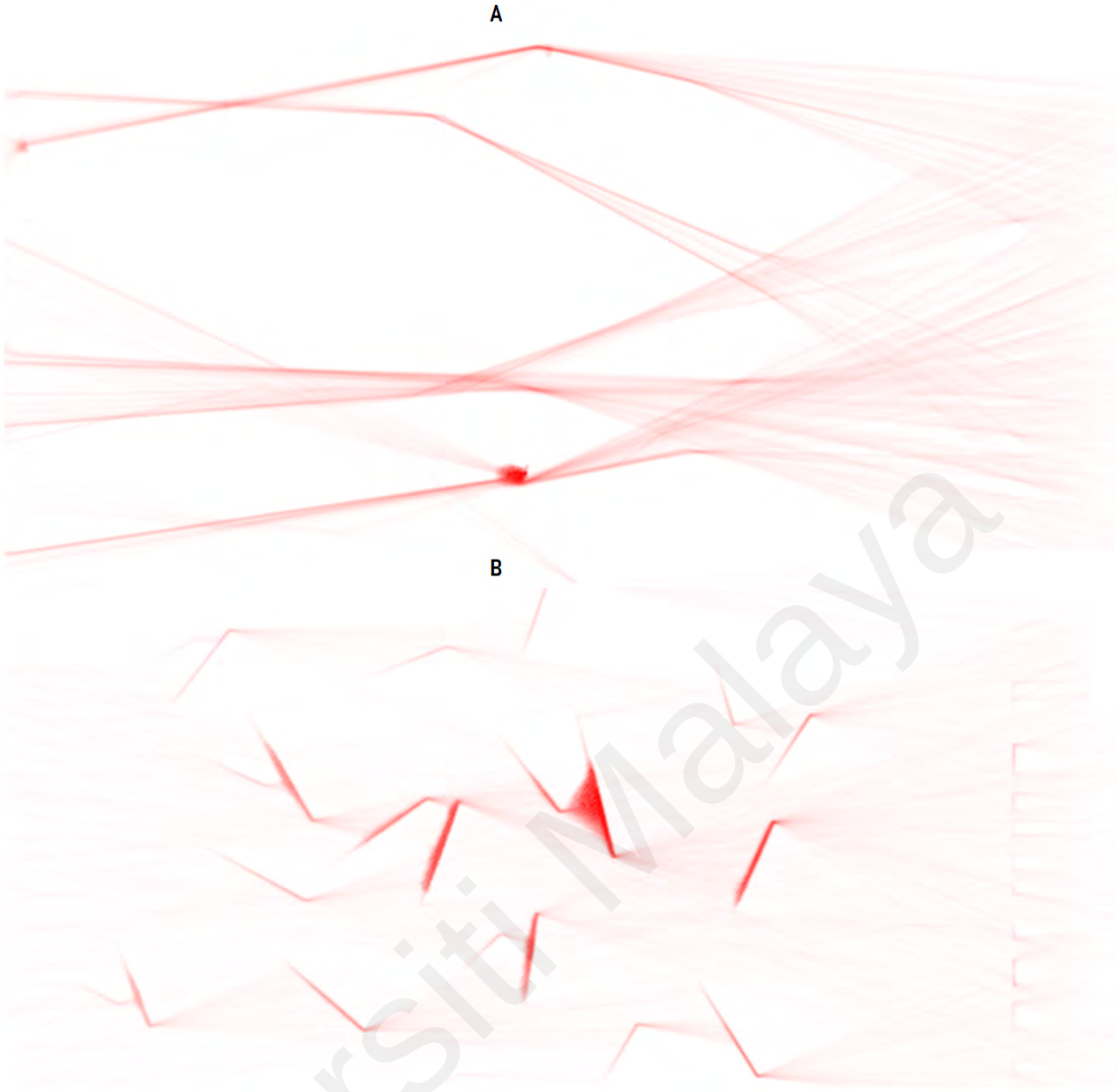

Figure 4.1: The heat maps for two types of pathfinding in the same environment with the same simulation settings. A shows the global shortest path method, and B shows the compass method. Every time step, each agent contributes points to its position on the heat map, resulting in a darker red color on coordinates where a lot of agents pass by.

differences, following research questions 1 and 3. In Figure 4.1, the differences are pretty straightforward. Still, when comparing simulations that should produce the same results if they were deterministic, the heat maps may look similar to the naked eye.

Environment 3, abbreviated by E3, with the singular choke point, shows that the compass method has lower lifetimes with large amounts of agents than the shortest path method. This scenario is included for completeness, but it hardly measures differences between the

two types of pathfinding. The choke point is made by creating a funnel through which the agents need to reach the goal. Such a scenario results in roughly the same paths for agents, as there is only one way to get from the spawn to the goal. Therefore, the difference seen in lifetimes is more a product of favourable interactions between the pathfinding method and the collision avoidance algorithm when using the compass method.

4.4 Parallelization Errors

The nature of parallelization errors is always the same, but is so fundamental that it pops up in various applications. The errors are almost always related to different processes reading and/or writing to the same memory, thus reading outdated information or overwriting another process' work. There are three ways this generic error can harm a crowd simulation when not taken into account:

1. Creating the tree structure takes quite some time and cannot be done in parallel because agents could be overwritten.
2. The tree structure cannot have too small tiles because if two agents move towards each other simultaneously, they could walk into each other.
3. When using the nearby agents method, it is not necessary to have a grid at all, but the parallel simulation still needs to be planned to be error-free.

For all three errors, tests have been conducted to see whether the simulation benefits from breaking these rules while the simulation itself stays more or less the same.

The errors are measured by counting lines that measure flow at a specific area, waiting and walking times, and heat map differences. For the heat map differences, the absolute values for each pixel in the heat map are subtracted, and the resulting number is then given, as well as the total absolute value of the heat map. The error is then presented by dividing the

heat map difference by the total heat map value. This means a small change in route or choke point can change the heat map drastically and therefore these differences are almost always relatively large, percentage-wise. A choke point has a huge impact on the heat map and if a choke point changes position, a heat map difference of more than 50% is not unlikely.

Having said that, the simulator is fully calibrated with many test runs to ensure it is deterministic and accurate when it needs to be. Aside from intentional non-deterministic elements, the same simulator settings will always result in the same statistics, even for the very sensitive heat map. For all tests, an Intel i5-6200U processor has been used.

4.4.1 Creating and filling the tree structure in parallel

The tree structure has many tiles, as many as one million for an environment the size of one screen. Of course tiles the size of a single pixel are not realistic, but real-world maps can easily be big enough to make many tiles a feasible option. When making so many tiles, creating the tree can take up to 330 *ms*, which is preferable. If possible, this should be shortened to an amount lower than the calculations done for all agents. An almost empty map was used to simulate various agents to test the impact of parallelizing this process. The map is almost empty to prevent the simulation environment from getting very crowded when running simulations with large crowds. Since the tree structure is completely invariant of the walls and other environmental elements, using an almost empty map is sufficient to find general answers for all environments regarding the tree structure. A large amount of agents is necessary to see whether this has a significant impact on the potential speed-up of the tree structure in the parallel set-up.

Only the tree structure method was used here. The nearby agents method could, in theory, be used to assist in collision detection as well. Still, it is important to note that the

nearby agents method does not require a tree structure at all and therefore a faster way to build the tree structure is of no value to that method. If parallelizing the building of the tree structure would prove effective, that would not mean the analysis of the tree structure versus the nearby agent method becomes invalid. Since this parallelization is non-deterministic, it is unfair to compare it to the deterministic nearby agents method. Therefore, the comparison in this section is strictly between the tree structure in parallel and serial form.

Simulations on an empty map tend to have around 1,400 agents on a busy map for each spawn per frame. Therefore, this will be the starting point for a measurement; frame rates before and after the scene are busy are not taken into account. The amount of time it takes to render a frame with no agents can vary significantly. The simulator becomes more of a background application and gets whatever sliver of processing power Windows assigns to idle applications. Spawn rates of 0.5, 1, 2, 4 and 8 are tested. A total of 5,000 agents test the spawn rates below 4, while 10,000 agents test the spawn rate of 4. The remaining spawn rate is tested for a total of 15,000 agents.

On top of that, two tile widths are tested; 7 and 20. As the tree is two dimensional, there is a factor 8 between the amounts of tiles in these trees. Tile width 7 makes for a vastly faster simulation than tile width 1 already, so a tiny difference could already be a convincing speed-up for this test. It is evident that calculating the tree structure in parallel gets a much stronger speed-up in absolute numbers as the tree structure gets larger, so a positive conclusion for tile widths 7 and 20 would imply the same or amplified values and conclusions for smaller tiles. Since the environment does not affect this process, pathfinding does not affect the test outcomes. The pathfinding used was the global shortest pathfinding, but results are invariant of pathfinding methods so that compass pathfinding

would yield the same results. Only a few small walls are included to force collisions and see whether the simulation's accuracy is affected much.

The results of these tests are not in parallelization's favour. When comparing all these simulations, a clear pattern of faster serial runs is visible. All runs combined, the average serial simulation is 8.7% faster than the average parallel run. Two reasons were identified for this result.

1. The overhead for starting multiple threads can be relatively large when parts of the tree structure are small. This can outweigh the processing power boost received from using multiple threads.
2. When two processes start writing agents into lists, the agent object can get corrupted. A try-catch construction had to be implemented to prevent this from happening. Letting the simulator run with these errors is no option, as they trigger a null reference exception. If this happens often, the overhead of a try-catch construction gets quite large.

It could still be possible that for smaller tiles, a speed-up can be realized this way, but as the simulator in its current state would get non-deterministic when doing so, it becomes hard to test accuracy.

The speed-down from building the tree structure in parallel alone is enough reason to say this method is not effective. For completeness, however, the accuracy has also been measured. Firstly, life times and waiting times are almost the same for both methods, within an error margin of 0.2% for all test runs. Similar results were observed for the several counting lines that measure the flow of people in specific areas. These all have a deviation of less than 1%, except for the simulation with spawn rate 8, which had a difference of 2%.

These percentages represent the difference between the test runs following the formula

$$\frac{|V_1 - V_2|}{V_1},$$

where V_i represents the value of that metric to run i. This method works fine for small differences since the denominator will not change so much about the percentage. For very large differences, e.g. >100%, this method would not be sufficient. When encountering such numbers, the conclusion is that there are large and significant differences. In this work, the result is sufficient, because a speed-up at the cost of massively different outcomes is not feasible. The heat maps differ 3.5 to 13.6%, which seems large at first, but given the sensitivity, the heat map differences are not that large. It is too large to call insignificant, however.

4.4.2 Tiles-size Related Errors

So far, when simulating in parallel, the simulator uses the already available tiles to calculate the next frame error-proof in parallel. In order to guarantee that this is error-proof , the tiles need to have a width and length of at least $2r + 3s$, where r is an agent's radius, and s is an agent's speed. When there are multiple speeds and radii, the largest is chosen. In serial, however, one agent is considered at a time and can only move a maximum of $1s$ per frame, so the distance between two agents only needs to be $1s$ from edge to edge and $2r+s$ from centre to centre. Of course, if two agents happen to be updated simultaneously and are less than $2s$ away from each other, they could run into each other. Using the tree structure method instead of the nearby agents method needs to be accounted for and potential rounding errors.

A smaller tile size results in a faster simulation for a larger number of agents. Figure 3.5 in Chapter 3 showed this very clearly. This test determined whether the simulation gets

significantly faster by moving from tile width $2r + 3s$ to $2r + s$ and whether that results in a simulation that yields significantly different results from the error-proof simulation. The tests include different amounts of agents and spawn rates, similar to the previous section, multiple maps, and pathfinding algorithms. The three maps used are a map without walls, one with a few walls, and many walls. The walls are placed strategically in the way of potential shortest paths, so they create preferred routes and choke points. Since agents number needs to be medium to large, if any such errors occur, the case with a spawn rate of 0.5 is excluded from the tests.

The results show that tile width 7 results in a faster simulation for smaller numbers of agents than tile width 5. As the smaller tiles mainly accommodate for large amounts of agents, the observation is expected. With more agents joining the environment, it gets more expensive to do all calculations for agents within a certain radius. The smaller tiles provide a solution. They decrease the number of agents to be checked quadratically. Therefore, with many agents, a smaller tile size helps the simulation speed a lot. For the simulations with lower spawn rates, resulting in less than 6,000 agents during the busy time, a very consistent difference of two to three *ms* is visible, favouring a tile width of 7. The tree structure creation analysis can explain this consistency. Whenever the tree structure is made using a specific tile width, it will always take a measurable amount of time to create and fill the tree structure. The time is constant and is longer when the tiles are smaller because there must be more tiles. As more agents get spawned, the speed-up per agent from using smaller tiles starts to outweigh the speed-down from building a larger tree structure. It is not the case for small amounts of agents, and the speed-up cannot be measured so well. The largest spawn rate further supports this explanation. The average speed-up for the spawn rate of 8 is 8.7 to 14.0% for an empty map; a significant increase in

speed. Extrapolating these results, a speed-up of more than 15% is not unlikely to occur for even higher spawn rates, creating a more dense crowd.

The waiting times and lifetimes are still quite similar, only varying up to 7 frames on a total of more than 1,500 frames to reach the goal. Similarly, the counting lines have errors of 0.1 to 2.5%, which is undoubtedly measurable, but not significant to the extent that the conclusions to be drawn from the simulation would become different. The heat maps are less promising. With differences of 6.3 to 19.7%, these could prove less accurate. The outcome does not mean they will look much different, but it is high enough a difference to doubt their precision.

4.4.3 Nearby Agents Method Parallelization Errors

When using the nearby agents method, it is unnecessary to have a grid of agents. That means the overhead of using the grid and some overhead in parallelizing the simulator can be omitted. It means the agents can be simulated simultaneously, even when they are right next to each other, causing problems, as mentioned previously. The nearby agents method was tested in an error-proof simulation and error-tolerant simulation to test the impact of omitting all planning. It would not be a fair comparison if the error-proof method used the grid from the tree structure. After all, that grid is far more complicated and extensive than necessary for update planning alone. Therefore, a column structure was used where alternating columns are assigned to different threads, making sure agents close to each other are not updated simultaneously. The same simulation settings were tested as for the previous section. Of course, the difference is that instead of tile size, the parallelization method is altered between test runs to match error-proof and error tolerance simulations.

There is a small but consistent speed-up visible for the error-tolerant approach. This

speed-up is in the range of 2.4 to 3.2% for each test run, which is significant but not large. This small speed-up comes at a price, but similar to the speed-up the inaccuracy is not large despite being significant, at 7.6 to 12.4%. The counting lines show a difference between runs with the same settings other than the error-tolerance of less than 4%. A few counting lines measure the same flow on both approaches and the differences seem not to scale with numbers, meaning the percentage is probably capped at this 4% for even larger crowds as well. The waiting times have been consistently approximately 1.4% higher for the error-tolerant approach. The consistency can be explained by the errors that occur, where agents walk into each other, meaning they will have to wait one frame before getting fixed.

4.5 Discussion

Although the research encompasses a large chunk of the topic at hand, not everything could be tested or taken into account.

4.5.1 Realistic Maps

As mentioned before, no real-world maps have been scanned or otherwise given as input to the simulator because the current test worlds contain the same elements such as walls and exits while keeping the maps within reasonable bounds. It would be amiss on this work, however, if this would be omitted from the discussion. The case could be made that real-world maps contain situations that have been overlooked in creating the environments used in this research. It is unlikely with the current simulator because agents only need to stop bumping into a wall or each other. Both events happen many times in all test runs, and any potentially overlooked scenario would at most change how often agents get stopped. Although that can lead to different simulations and outcomes related to those simulations, it is doubtful that those events would change the dynamics of parallelization errors.

4.5.2 Impact of Different Metrics

The results for each metric are given, and no particular valuation is given to metrics relative to each other. While some metrics could give more information than others, the value of a metric highly depends on the use case. When simulating an emergency evacuation, one may be more interested in the flow of people through specific doors, even if a heat map gives more information about the people density overall. For this reason, the metrics are all measured and reported as equals.

4.5.3 Artificial Intelligence

The simulator contains no elements of artificial intelligence. While artificial intelligence could, in theory, give different pathfinding and even prevent some collisions, it would still be likely to have some collisions and be prone to parallelization errors. Therefore, some of the results and conclusions in this dissertation may also apply to simulations based on artificial intelligence.

4.5.4 Static Environments

In real situations, events can change the environment and the corresponding preferred paths, e.g. a car passing by or a tree falling on the road. While this situation could, in theory, still be simulated using compass pathfinding without too much change, the current global pathfinding algorithm would slow down drastically as a result of dynamic environments. As there is no direct reason to assume such dynamic environments would change the impact of parallelization errors. Because of the primarily increased complexity of implementing such events, this project's scope is limited to static simulation environments.

4.6 Chapter Summary

Although not every possible environment and pathfinding could have been tested, many tests have been run to determine the accuracy of error-tolerant crowd simulations. Tests had

varying degrees of success; error tolerance produced 2.4 to 14.0% simulations faster than their error-preventive counterparts, except tests where not the agents, but data structures supporting the agent updates were calculated in parallel. The error in waiting and lifetimes did not exceed 1.5% for all simulations, and the error in flow measured by counting lines did not exceed 4%. The heat maps differ more percentage-wise, but the next section will elaborate a bit more on that. In conclusion, all these results are examined and explained further.

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 Conclusion

The main goal of this research is to quantify the impact of tolerating errors in large scale parallel crowd simulations. Two sub-goals had to be achieved first. One is to develop a crowd simulator capable of simulating crowds with optional error tolerance for various features. The other is to find metrics that can be used to measure these errors well. Crowd simulation is a highly competitive field, and speed can be a deciding factor for event organizers, building planners and emergency caregivers in choosing their preferred simulator. A feature with the potential to speed up a crowd simulation significantly at a relatively low cost is a very relevant feature to explore.

All objectives have been achieved successfully. It turns out that the combination of agent waiting times, agent walking times, local agent flow and heat maps are very effective at measuring simulation accuracy. Naturally, the Windows StopWatch class is excellent to measure performance in terms of speed. A simulator tracking all these measurements is built from scratch so every feature could be controlled and modified. Different types of errors were tested in isolation using this simulator. For most metrics, it is pretty simple to see whether a deviation is significant. If agents walk for 10% less time, they are considerably faster; if the simulator runs 5% faster, that is a significant speed-up, and so on. For heat maps, however, this is not necessarily the case. The heat map, as implemented, is very sensitive to change. If an agent makes a single change in its path, the rest of the path will likely not line up with the original path, changing all values in the heat map. That means when subtracting the original heat map from the changed heat map, the deviation can easily be 150% of the original heat map. In larger-scale simulations, some path

changes may negate each other on the heat map, but if errors were to change choke points or preferred paths for many agents, the subtracted heat map would still likely be over 100% different. The maximum difference is 200% since the heat maps have zero overlaps and every agent in both simulations produces a full deviation..

The data indisputably shows a significant increase in simulation speed when tolerating errors in simulations where the agents are simulated in parallel. The specific speed-up depends on the type of error that is tolerated. The simulation statistics deviate significantly from their error-preventive counterparts but are percentage-wise significantly smaller than the speed-up. The most basic error-tolerant approach, where no agent planning is applied, seems to have the most predictable errors as the statistics deviate within a relatively small range of numbers.

Furthermore, it is not beneficial to apply error tolerance elsewhere in the simulation. Only when the speed-up scales with the number of agents it can be valuable as a feature. Heat maps are the only statistic that percentage-wise can exceed the speed-up consistently, but as mentioned before, the heat map is much more sensitive to small changes. The heat maps in the error-tolerant simulations still look similar to those in the error-preventive simulations, to the naked eye. Distinctive features such as choke points and busy paths are still seen in the same areas, and therefore even the less accurate heat maps can be of value.

The outcome is large and consistent enough that the chance of this outcome or deviation being the result of noise in the data is close to zero. It does not mean that an error is large enough to render a simulation useless due to its significant inaccuracy. Multiple factors need to be considered to determine whether an error in a specific statistic is too large. For

example, when simulating an emergency evacuation, it may be of great importance to have accurate people flow through emergency exits. As every situation requires a different analysis of the importance of accuracy, and this analysis largely depends on the expertise of people working with simulators in practice, deciding whether a simulation is accurate enough based on these statistics is out of this dissertation's scope.

5.2 Future Work

Research following this dissertation could focus on dynamic maps, where error tolerance is tested in a map that changes over time and how this complex environmental behaviour could be simplified and sped up using approximations rather than precise simulations. This research could be extended with a study on the usefulness of simulations known to be inaccurate to a specified extent. In line with that, 'wrong' simulations could still prove useful in finding rough results quickly in the process of converging to a more detailed environment that should be examined more thoroughly. This use and the potential role error-tolerant simulations can play in finding environments worth testing with high accuracy could be researched as well.

REFERENCES

Chen, D., Wang, L., Tian, M., Tian, J., Wang, S., Bian, C., & Li, X. (2013). Massively parallel modelling & simulation of large crowd with gpgpu. *Journal of Supercomputing*, *63*, 675–690.

Datta, S., & Behzadan, A. (2019). Modeling and simulation of large crowd evacuation in hazard-impacted environments. *Advances in Computational Design*, 91–118.

Dou, W., & Li, Y. (2018). A fault-tolerant computing method for xdraw parallel algorithm. *Journal of Supercomputing*, *74*(6), 2776–2800.

Gardner, M. (1970). Mathematical games - the fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, *223*, 120–123.

Gorrini, A., Crociani, L., Vizzari, G., Bandini, S., Franzoni, V., Milani, A., . . . Vallverdú, J. (2019). Stress estimation in pedestrian crowds: Experimental data and simulations results. *Web Intelligence*, *17*(1), 85–99.

Karamouzas, I., Sohre, N., Narain, R., & Guy, S. (2017). Implicit crowds: Optimization integrator for robust crowd simulation. *ACM Transactions on Graphics*, *36*(4), 136:1–136:13.

Kim, H., Han, J., & Han, S. (2019). Analysis of evacuation simulation considering crowd density and the effect of a fallen person. *Journal of Ambient Intelligence and Humanized Computing*, *10*, 4869–4879.

Liu, L., Jin, Y., Liu, Y., Ma, N., Huan, Y., Zou, Z., & Zheng, L. (2018). A design of autonomous error-tolerant architectures for massively parallel computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *26*(10), 2143–2154.

Malinowski, A., & Czarnul, P. (2019). Multi-agent large-scale parallel crowd simulation with nvram-based distributed cache. *Journal of Computational Science*, *33*, 83–94.

Mattsson, P. (2014). Why haven't cpu clock speeds increased in the last few years? *COMSOL Blog*.

Peymanfard, J., & Mozayani, N. (2018). A data-driven method for crowd simulation

using a holonification model. *Journal of Artificial Intelligence and Data Mining*, *7*, 403–409.

Sillapaphiromsuk, S., & Kanongchaiyos, P. (2019). Explicit energy-minimal short-term path planning for collision avoidance in crowd simulation. *Engineering Journal*, *23*(2), 23–53.

Sun, L., & Badler, N. (2018). Exploring the consequences of crowd compression through physics-based simulation. *Sensors*, *18*, 1–16.

Wang, J., S. Lo, Q. W., Sun, J., & Mu, H. (2013). Risk of large-scale evacuation based on the effectiveness of rescue strategies under different crowd densities. *RISK ANALYSIS*, 1553–1563.

Wang, Q., Liu, H., Gao, K., & Zhang, L. (2019). Improved multi-agent reinforcement learning for path planning-based crowd simulation. *IEEE Access*, *7*, 73841–73855.

Weiss, T., Litteneker, A., Jiang, C., & Terzopoulos, D. (2019). Position-based real-time simulation of large crowds. *Computers & Graphics-UK*, *78*, 12–22.

Wong, S., Wang, Y., Tang, P., & Tsai, T. (2017). Optimized evacuation route based on crowd simulation. *Computational Visual Media*, *3*(3), 243–261.

Yilmaz, E. (2010). *Massive crowd simulation with parallel processing* (Unpublished doctoral dissertation). Middle East Technical University, Ankara, Turkey.

Zhang, L., Lai, D., & Miranskyy, A. (2019). The impact of position errors on crowd simulation. *Simulation Modelling Practice and Theory*, *90*, 45–63.