Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit Informatik

Parallelisierung eines Modells zur Simulation von Personenströmen

Autoren	Martin Blöchlinger Roman Müntener
Hauptbetreuung	Markus Thaler
Nebenbetreuung	Albert Steiner Manuel Renold
Industriepartner	Savannah Simulations AG

Datum	06.06.2014

Zusammenfassung

Aufgrund immer stärkerer Frequentierung von öffentlichen Anlagen wie Bahnhöfen oder Flughäfen wird es immer wichtiger, grosse Massen von Fussgängern effizient handhaben zu können. Fussgängersimulationen bilden das Verhalten von Fussgängern nach und sind daher ein Instrument um beurteilen zu können, wie gut sich Architektur, Fahrplan oder Engpässe wie Schalter oder Sicherheitskontrollen eignen, mit vielen Personen umzugehen. Weiter können solche Simulationen bei der Planung von Grossanlässen eingesetzt werden. In dieser Arbeit soll Anhand eines vereinfachten Modells zur Fussgängersimulation überprüft werden, ob sich dieses Modell parallelisieren lässt und wie gut es skaliert. Es werden mehrere Parallelisierungsansätze und Algorithmen zur Implementation des Modells diskutiert und in einem Prototypen implementiert. Dabei wurde als Parallelisierungsansatz eine Unterteilung des zu simulierenden Gebietes in Zonen gewählt, die parallel verarbeitet werden können. Anhand dieses Prototypen wurde mittels Messungen analysiert, wie gut der Parallelisierungssansatz mit der Anzahl Fussgänger und der Anzahl zur Verfügung stehender Rechner oder CPUs skaliert. Auf einem Rechner mit mehreren CPUs konnte mit dem Paralleliserungsansatz die Gesamtlaufzeit im Vergleich zu einer Simulation ohne Parallelisierung stark gesenkt werden. Der Prototyp ist zudem in der Lage, die Simulation auf mehrere Rechner in einem Netzwerk zu verteilen. Dadurch konnte eine weitere Reduktion der Gesamtlaufzeit erreicht werden. Der Prototyp wurde mit bis zu einer Million Fussgängern in einem Rechnerverbund bestehend aus 10 Rechnern getestet und konnte im Falle von einer Million Fussgängern einen Simulationslauf 12-mal langsamer als Echtzeit ausführen.

Abstract

Public places such as airports or train stations are increasingly frequented by travelers, thus it is important to be able to handle large numbers of pedestrians. Pedestrian simulations emulate the behavior of pedestrians and are a tool to analyze how well architecture, time-tables or choke points such as counters or security check points are capable of handling a lot of pedestrians. Additionally these tools can be used to plan public events where large crowds of pedestrians are to be expected. This bachelor thesis analyzes the possibility to parallelize pedestrian simulations on the basis of a simplified pedestrian behavior model and collects data of how good the parallelization scales with multiple CPUs and multiple computers. Different parallelization strategies and algorithms to implement the behavioral model were discussed and implemented in a prototype. The chosen parallelization strategy divides the simulation area into zones that can be processed in parallel. Simulations were made and their total runtimes were measured to analyze how well the strategy scales with the amount of pedestrians and available CPUs or computers. On a single computer with multiple CPUs the parallelization strategy allowed for a significant reduction of the total runtime compared to a simulation without parallelization. The prototype is able to distribute the simulation to multiple computers in a network. This also led to an additional reduction of the total runtime. The prototype was tested with up to one million pedestrians in a computer network of 10 computers and in the case of one million pedestrians it was able to run the simulation 12-times slower than real-time.

Vorwort

Im vergangenen Semester haben wir Herrn Thaler gefragt, ob es möglich wäre, unsere Projektarbeit irgendwie als Bachelorarbeit fortzusetzen. Wir dachten uns jedoch bereits, dass es den Anforderungen nicht genügen würde und so war es dann auch. Also haben wir gefragt, ob Herr Thaler eine andere Idee hätte und er schlug uns diese Arbeit im Bereich Parallel Computing vor. Fussgängersimulation ist ein sehr interessantes Thema und bietet viel Spielraum für Ideen und wir hätten uns gerne weiter mit der eigentlichen Simulation des Fussgängerverhaltens beschäftigt. Parallel Computing ist ebenfalls ein interessantes, aber auch ein herausforderndes Thema.
Zu Beginn der Arbeit wurde die Stimmung etwas getrübt, da erste Implementationsversuche schlicht nicht performant genug für eine mögliche Parallelisierung waren. Es hat jedoch Spass gemacht, dieses Problem erfolgreich durch kreative Ansätze zu bewältigen.

Wir danken Herrn Thaler für die Betreuung und dass er es uns ermöglicht hat, diese Arbeit durchzuführen. Ebenso danken wir Herrn Steiner und Herrn Renold für die konstruktiven Diskussionen über mögliche Ansätze und Ideen. Weiter möchten wir Herrn D. Früh für seine Unterstützung bei den Messungen danken. Er ermöglichte es uns, die Messungen auf den Schulrechnern in einem Labor durchzuführen.

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……...

……

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Inhaltsverzeichnis

1. Einleitung

1.1. Ausgangslage

Durch die zunehmende Urbanisierung vieler Regionen verkehren viele Personen auf nur wenig Platz. Dies wird auch durch die zunehmende Mobilität der Bevölkerung begünstigt. Vor allem an Bahnhöfen, Flughäfen und Häfen macht sich dies bemerkbar. Zudem gewinnen Grossanlässe und stark frequentierte Orte an Bedeutung. Die Kapazitätsgrenzen der Systeme des öffentlichen Verkehrs sowie auch des Individualverkehrs sind bald erreicht. Dadurch steigen die Risiken, dass bei Extremereignissen wie zum Beispiel einer Evakuation eines Gebäudes infolge eines ausgebrochenen Feuers Mensch und Umwelt, aber auch Infrastrukturen zu schaden kommen. Um diese Risiken zu reduzieren, sind intelligente Lösungen gefragt. Die Durchführung von Simulationen und die Analyse der Resultate können dazu beitragen, solche Lösungen zu entwickeln. Ausserdem können sie zur Planung von Grossanlässen wichtige Hinweise liefern, wo Flaschenhälse auftreten könnten. Diese können dann eliminiert und somit die Effizienz des Personenflusses gesteigert werden [10, S. 10–11].

Es gibt eine Vielzahl an Simulationssoftware für Fussgängersimulationen. Diese kann spezifisch auf das Anwendungsgebiet zugeschnitten sein wie zum Beispiel die Software *CAST*[1] von der Airport Research Center GmbH zur Simulation eines Flughafens, die ebenfalls eine Fussgängersimulation beeinhaltet.
Es gibt auch Simulationssoftware, welche generell die Simulation von Fussgängern in einer beliebigen Anlage ermöglicht. Dazu gehört insbesondere die Applikation *SimWalk*[2] des Industriepartners Savannah Simulations AG.

Im Rahmen eines KTI-Projektes wird am Institut für Datenanalyse und Prozessdesign in Zusammenarbeit mit dem Industriepartner ein Modell entwickelt, welches das Verhalten von Fussgängern möglichst realistisch simulieren soll. Diese Modelle gehen von einer seriellen Verarbeitung aus. Das Modell wird im Kapitel 2.2.4 auf Seite 13 genauer erläutert.

Durch eine Parallelisierung des Modells erhofft man sich eine effizientere Nutzung der vorhandenen Hardware und eine geringere Laufzeit einer Simulation. Ebenfalls soll durch die Parallelisierung eine grössere Anlage mit einer grösseren Anzahl an Fussgängern simuliert werden können. Das Ziel dieser Arbeit ist die Parallelisierung dieses Modells und wird im folgenden Abschnitt genauer erläutert.

1.2. Aufgabenstellung

Das Ziel dieser Arbeit ist es, das Modell so zu parallelisieren, damit es möglichst gut mit der Anzahl Fussgänger, der Anlagengrösse und der Anlagenkomplexität skaliert. Es soll ein Daten- und Verarbeitungsmodell entworfen und auf verschiedenen Plattformen getestet werden. Als Plattform kann ein Shared-Memory-System mit GPUs oder ein verteiltes System dienen. Diese Anforderungen werden in der offiziellen Aufgabenstellung definiert, die im Anhang auf Seite 84 beigefügt ist.

[1] http://www.airport-consultants.com/
[2] http://simwalk.com/

1.3. Terminologie

Im Rahmen dieser Arbeit werden die Bezeichnungen *Fussgänger* und *Personen* synonym benutzt. Die Bezeichnung Fussgänger beinhaltet sowohl männliche wie auch weibliche Personen. Die Bezeichnung *andere Fussgänger* wird für Fussgänger mit Ausnahme des betrachteten Fussgängers verwendet. Als *statische Objekte* werden Hindernisse wie Wände, Mauern oder andere Objekte bezeichnet, die sich nicht bewegen. Die Umgebung, in welcher eine Simulation erfolgt, wird als *Karte* oder *Anlage* bezeichnet. Weitere Begriffe werden im Glossar auf Seite 82 beschrieben.

2. Grundlagen

2.1. Charakteristiken von Personenströmen

Um den Realismusgrad von Fussgängersimulationen einschätzen zu können, muss zuerst bekannt sein, wie sich Fussgänger in realen Situationen verhalten. Diese Beobachtungen können dann mit einer Computersimulation verglichen werden, um entscheiden zu können, ob diese als realistisch eingestuft werden kann. Zwei bekannte Charakteristiken werden nachfolgend vorgestellt.

Wenn sich zwei Gruppen von Fussgängern in einem Korridor in entgegengesetzter Richtung bewegen, bilden sich gemäss [3, S. 6] spontan Spuren in die jeweilige Richtung. Fussgänger, die sich in einer Gruppe weiter hinten befinden, reihen sich also automatisch hinter Fussgängern ein, die in die gleiche Richtung gehen. Abbildung 2.1 (Seite 10) zeigt eine solche Situation.

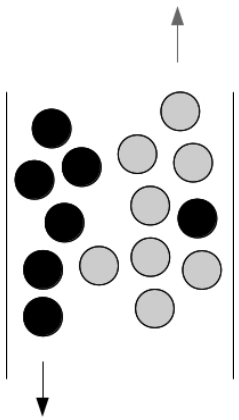


Abbildung 2.1.: Spurbildung bei Fussgängergruppen in entgegengesetzten Richtungen

Ein weiteres typisches Verhalten kann gemäss [3, S. 8] beobachtet werden, wenn sich Personenströme in einem bestimmten Winkel kreuzen. Um sich durch den Strom zu kämpfen, der sich in die andere Richtung bewegt, bilden sich Streifen, die vertikal resp. horizontal verlaufen. Diese Situation ist in den Abbildungen 2.2 und 2.3 (Seite 11) abgebildet.

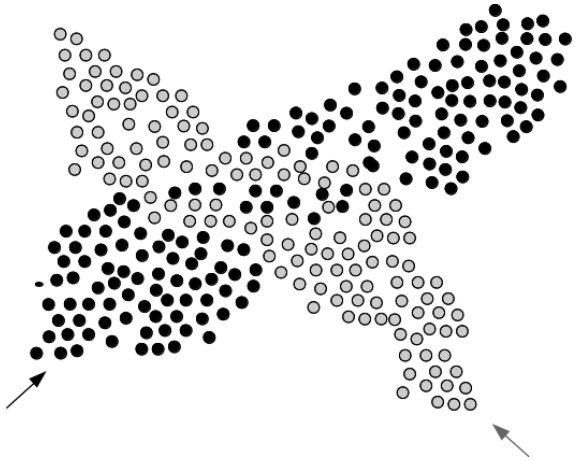


Abbildung 2.2.: Spurbildung bei sich kreuzenden Fussgängern

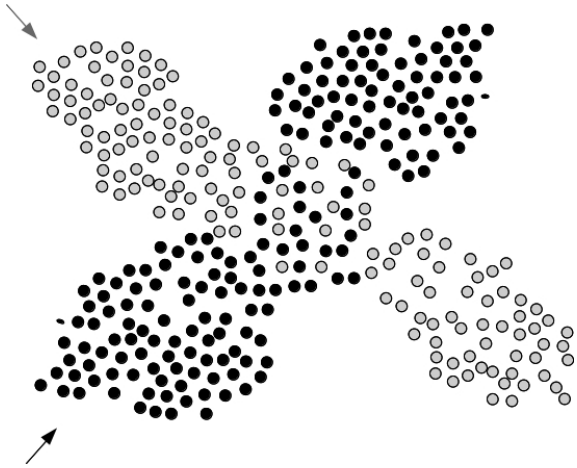


Abbildung 2.3.: Spurbildung bei sich kreuzenden Fussgängern

2.2. Fussgängersimulationen

Es gibt verschiedene Modelle um Fussgänger zu simulieren. Die Grundlegende Struktur um den Fussgängerverkehr zu beschreiben kann gemäss Hoogendoorn et al. [5] und Wagoum [11, S. 11–12] in drei hierarchische Stufen eingeteilt werden.

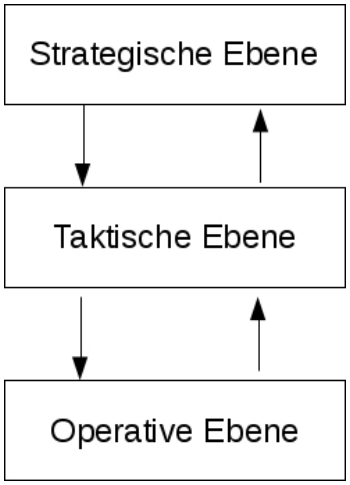


Abbildung 2.4.: Struktur einer Fussgängersimulation

Diese Stufen sind in Grafik 2.4 (Seite 12) abgebildet. Die strategische Ebene beschäftigt sich mit der grundlegenden Routenwahl eines Fussgängers. Dieser kennt seinen Startpunkt und meist auch den Zielpunkt. Mit der strategischen Ebene kann nun der kürzeste Weg vom Start- bis zum Zielpunkt berechnet werden. Je nach Detailgrad des Modells kann hierbei auch unterschieden werden, ob der Fussgänger wirklich die kürzeste Route kennt. Dies wäre zum Beispiel bei Pendlern oder sehr ortskundigen Fussgängern der Fall. Bei Touristen könnte ein anderer Algorithmus verwendet werden.
Ebenfalls auf dieser Ebene können Aktivitäten festgelegt werden, die ein Fussgänger während seiner Reise durchführen soll. So kann ein Fussgänger beispielsweise an einem Kiosk eine Zeitung kaufen, bevor er auf das eigentliche Ziel zusteuert.
Eine Hierarchiestufe tiefer befindet sich die taktische Ebene. Durch die strategische Ebene wurde bereits eine Route berechnet. Während ein Fussgänger dieser Route folgt, können jedoch weitere Ereignisse eintreffen, welche die Entscheidung über die kürzeste Route beeinflussen. Wenn sich viele Fussgänger auf der gleichen Route befinden, verlangsamt sich die Fortbewegungsgeschwindigkeit der Fussgänger, bis ein Stau entsteht. Auf der taktischen Ebene können Fussgänger solche Ereignisse erkennen und Alternativrouten berechnen.
Auf der operativen Ebene wird der unmittelbare Entscheid gefällt, in welche Richtung und mit welcher Geschwindigkeit sich der Fussgänger mit dem nächsten Simulationsschritt bewegen soll. Diese Entscheidung wird durch andere Fussgänger und statische Objekte (z.B. Wände, Abschrankungen, geparkte Autos) im Sichtfeld des Fussgängers beeinflusst.

In den anschliessenden Abschnitten werden verschiedene Ansätze zur Fussgängersimulation kurz besprochen.

2.2.1. Zelluläre Automaten

In [1] wird auf den Einsatz von zellulären Automaten und deren Probleme eingegangen: Zelluläre Automaten basieren darauf, dass die Anlage in ein reguläres Gitter eingeteilt wird. Die Zellen dieses Gitters besitzen Zustände, die in diskreten Zeitschritten ändern können. Die Hindernisse der Anlage und die Fussgänger werden den Zellen zugeordnet. Dadurch ist es einfach möglich, die Hindernisse eines Fussgängers in seiner Umgebung zu bestimmen, da dazu nur die Zustände der umliegenden Zellen abgefragt werden müssen. Eine Simulation kann dadurch sehr effizient durchgeführt werden.
Dieses Modell besitzt aber auch Nachteile. Ein Fussgänger kann sich nur von Zelle zu Zelle bewegen. Beschleunigung und unterschiedliche Geschwindigkeiten lassen sich deswegen nur bedingt modellieren. Ein Problem besteht beispielsweise bereits dann, wenn sich ein Fussgänger eine Zelle in diagonaler Richtung weiterbewegt. In diesem Fall hätte sich der Fussgänger schneller bewegt, als wenn er sich in

vertikaler oder horizontaler Richtung bewegt hätte, da er in einem Zeitschritt mehr Weg zurückgelegt hätte.

2.2.2. Fluiddynamik

Wie Helbing [2, S. 1–2] schreibt, ähneln Fussabdrücke von Menschenmengen im Schnee und in Zeitraffer-Bildern von Fussgängern Strömen in Flüssigkeiten. Formeln aus der Physik können als Basis zur Berechnung von verschiedenen Fussgängergruppen verwendet werden. Die resultierenden Gleichungen ähneln den Gleichungen zur Berechnung von Flüssigkeiten, aber es werden weitere Terme für die Zielrichtung und Interaktion der Fussgänger verwendet. Durch die Anwendung dieser Formeln kann die für Personenströme charakteristische Spurbildung erreicht werden [2, S. 20].

2.2.3. Social-Forces-Modell

Im Social-Forces-Modell [4] wird das Fussgängerverhalten durch Kräfte anderer Fussgänger und Hindernissen beeinflusst. Wenn der Fussgänger seinen Zielpunkt ohne Hindernisse erreichen kann, wird er diesen mit seiner Wunschgeschwindigkeit ansteuern. Sind jedoch andere Personen in Reichweite, wird der Fussgänger durch diese beeinflusst. Je näher der Fussgänger an einer anderen Person oder Wand vorbeigehen muss, desto unwohler fühlt er sich. Diese sozialen Kräfte beeinflussen die Entscheidung des Fussgängers, in welche Richtung und mit welcher Geschwindigkeit er weitergehen will. In diesem Modell können zusätzlich auch anziehende Kräfte einberechnet werden. So können beispielsweise anziehende Kräfte von Schaufenstern einbezogen werden.
In Computersimulationen kann ebenfalls Spurbildung beobachtet werden [4, S. 4].

2.2.4. Behavioral heuristics

Im Modell von Moussaïd, Helbing und Theraulaz [7] passt ein Fussgänger seine Geschwindigkeit und Richtung an, je nachdem wo sich im Sichtfeld Hindernisse oder andere Fussgänger befinden. Dabei besitzt jeder Fussgänger wie im Social-Forces-Modell eine Komfortgeschwindigkeit und einen Zielpunkt. Der Fussgänger versucht möglichst direkt mit seiner Komfortgeschwindigkeit den Zielpunkt anzusteuern. Vor jedem Schritt muss der Fussgänger sein Sichtfeld auf Hindernisse und andere Fussgänger abtasten. Dabei wird in einem bestimmten Winkel nach links und rechts ausgehend von der Zielrichtung überprüft, welche Distanz der Fussgänger mit seiner Wunschgeschwindigkeit bis zu einer Kollision gehen kann. Dabei werden auch die Geschwindigkeiten der anderen Fussgänger berücksichtigt.
Ein Fussgänger will sein Ziel möglichst hindernisfrei erreichen. Gleichzeitig will er aber auch nicht zu weit vom direkten Zielpfad abweichen. Diese Entscheidung kann durch eine Kostenfunktion modelliert werden:

$$d(\alpha) = d_{max}^2 + f(\alpha)^2 - 2d_{max}f(\alpha)cos(\alpha_0 - \alpha) \tag{2.1}$$

Die in der Gleichung 2.1 verwendeten Parameter und Funktionen bedeuten:

- α Winkel zwischen der Blickrichtung und der potenziellen Laufrichtung
- d_{max} maximale Sichtdistanz
- $f(\alpha)$ Funktion, die zu gegebenem Winkel die Distanz bis zur Kollision berechnet
- α_0 Winkel zwischen Blickrichtung und Zielrichtung

Wurde eine Richtung bestimmt, muss die Geschwindigkeit berechnet werden, mit der sich der Fussgänger fortbewegen soll. Die Geschwindigkeitsänderung kann durch die Formel 2.2 ausgedrückt werden, wobei $\vec{v}_i$ die aktuelle Geschwindigkeit, $\vec{v}_{des}$ die Wunschgeschwindigkeit und τ die Relaxation Time ist:

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{v}_{des} - \vec{v}_i}{\tau} \tag{2.2}$$

Die Relaxation Time τ ist die Zeit, die ein Fussgänger mindestens benötigt, um rechtzeitig abbremsen zu können. Die Wunschgeschwindigkeit ist davon abhängig, denn es wird immer ein Sicherheitsabstand bewahrt, um innerhalb von τ noch abbremsen zu können. Dies kann ausgedrückt werden als $v_{des}(t) = min(v_i^0, \frac{d_h}{\tau})$. d_h ist dabei die Distanz zwischen dem Fussgänger und dem ersten Objekt in der gewählten Richtung zum Zeitpunkt t. Der Vektor $\vec{v}_{des}$ (Wunschgeschwindigkeit) zeigt mit der Länge von $\|\vec{v}_{des}\|$ in die gewählte Richtung. v_i^0 bezeichnet die Komfortgeschwindigkeit des Fussgängers. Diese Geschwindigkeit versucht der Fussgänger während der Simulation zu erreichen, was allerdings nur möglich ist, wenn sich der Fussgänger frei bewegen kann.

Weitere Formeln [7, S. 2–3] definieren das Verhalten in sehr dichten Situationen (Kollisionen).
Dieses Modell wurde in dieser Arbeit als Basis verwendet. Die konkrete Implementation wird im Abschnitt 5.4.2 auf Seite 36 beschrieben.

2.3. Parallele Programmierung

Die Idee der parallelen Programmierung ist es, mehrere Berechnungsströme zu erzeugen, die gleichzeitig ausgeführt werden können. Dazu wird das Problem in Teilaufgaben zerlegt, die nachfolgend Tasks genannt werden. Diese Tasks werden dann mittels Threads oder Prozessen auf die Recheneinheiten (CPU) abgebildet. Die Zuordnung der Tasks an Prozesse oder Threads wird Scheduling genannt, wobei zwischen statischem und dynamischen Scheduling unterschieden werden kann. Bei statischem Scheduling ist die Zuteilung bereits vor dem Programmstart definiert. Bei dynamischem Scheduling kann die Zuteilung während der Laufzeit des Programms verändert werden. Dies kann dazu verwendet werden, die Rechenlast auf die Ressourcen möglichst gleichmässig zu verteilen [9, S. 22–23].

Nachfolgend werden einige verschiedene Plattformen und Tools zur parallelen Entwicklung von Anwendungen vorgestellt.

2.3.1. Plattformen

Die verschiedenen parallelen Hardware-Architekturen können gemäss der Flynnschen Klassifikation [9, S. 27–29] klassifiziert werden:

SISD (Single Instruction, Single Data) Rechner besitzen jeweils einen Daten- und Instruktionsstrom. Sequentielle Abarbeitung.

MISD (Multiple Instruction, Single Data) Rechner besitzen mehrere Instruktionsströme aber nur einen Datenstrom. Die Rechenressourcen können verschiedene Instruktionen auf die gleichen Daten anwenden. Dies hat jedoch praktisch nur geringe Relevanz.

SIMD (Single Instruction, Multiple Data) Rechner besitzen einen Instruktionsstrom und mehrere Datenströme. Die Rechenressourcen führen jeweils die gleiche Instruktion auf unterschiedliche Daten aus.

MIMD (Multiple Instruction, Multiple Data) Rechner besitzen mehrere Daten- und Instruktionsströme. Jede Rechenressource kann unterschiedliche Instruktionen auf unterschiedliche Daten ausführen.

Eine weitere Unterteilung kann aufgrund der Speicherorganisation gemacht werden:

Rechner mit verteiltem Speicher Die Rechenressourcen haben jeweils einen eigenen zugeordneten Speicherbereich. Will eine Recheneinheit auf einen Speicherbereich zugreifen, der einer anderen Rechenressource zugeordnet ist, muss dies über Programmiermechanismen, wie z.B. Kommunikation über ein Netzwerk, erfolgen.

Rechner mit gemeinsamen Speicher (Shared Memory) Die Rechenressourcen können alle auf den gleichen globalen Speicher zugreifen und teilen sich somit den Speicherbereich.

Nachfolgend werden einige Plattformen vorgestellt.

Multicore-Prozessoren

Multicore-Prozessoren sind Prozessoren, welche mehrere Rechenkerne (Cores) auf dem gleichen Chip besitzen. Diese Rechenkerne bilden die Rechenressourcen. Multicore-Prozessoren sind MIMD-Systeme. Jeder Prozessorkern kann seine eigenen Instruktionen ausführen. Dabei kann der Speicherzugriff separat erfolgen. Multicore-Systeme verfügen über einen gemeinsamen Speicher.
Die Programmierung von Multicore-Systemen ist eng mit der parallelen Thread-Programmierung verbunden. In modernen Programmiersprachen können explizit Threads erzeugt werden, die dann während der Laufzeit den programmierten Code ausführen. Der Programmierer muss allerdings durch Synchronisationsmechanismen sicherstellen, dass Speicherzugriffe in korrekter Reihenfolge erfolgen, weil sich die Recheneinheiten den Speicherbereich teilen. Ansonsten kann es zu unerwünschten Nebeneffekten kommen, weil die Threads durch das Betriebssystem eingeplant werden und der Programmierer darauf keinen Einfluss hat.

Grafikkarte

Grafikkarten sind ursprünglich mit dem Ziel entwickelt worden, die grafische Bildschirmausgabe zu berechnen. Der Prozessor der Grafikkarte wird als GPU (Graphics Processing Unit) bezeichnet. Die Grafikkarten wurden in den letzten Jahren flexibler nutzbar, wodurch auch andere Berechnungen auf der GPU durchgeführt werden können. Durch APIs und eine standardisierte Programmiersprache kann die GPU angesteuert werden. Dies erfordert jedoch ein anderes Programmiermodell als das eines Multicore-Prozessors, denn GPUs sind SIMD-Systeme. GPUs besitzen sehr viele Rechenkerne und wenig lokalen Speicher. Für die Verarbeitung auf der Grafikkarte müssen die zu verarbeitenden Daten auf die Grafikkarte transferiert werden. Danach kann ein Programm auf der Grafikkarte ausgeführt werden. Die Instruktionen dieses Programms sind für alle Rechenkerne gleich. Die veränderten Daten müssen nach der Verarbeitung wieder zurück in den Hauptspeicher zurücktransferiert werden.

Verteilte Systeme

Durch die Verbindung mehrerer Multicore-Systeme kann ein Programm auf mehreren Computern parallel gerechnet werden. Dieser Ansatz ist ebenfalls ein MIMD-System, jedoch handelt es sich dabei nicht mehr um ein Shared-Memory-System. Jeder Computer besitzt einen eigenen Speicher. Will ein Computer auf den Speicher eines anderen zugreifen, ist dazu Kommunikation erforderlich. Ausserdem steigt in einem verteilten System der Kommunikationsaufwand, da üblicherweise Daten über ein Netzwerk ausgetauscht werden müssen. Die Komplexität der Programmierung wird ebenfalls erhöht, da zusätzliche Rahmenbedingungen betrachtet werden müssen. So können beispielsweise heterogene Systeme verwendet werden, welche andere Hardware-Komponenten oder Betriebssysteme nutzen. Ausserdem müssen verschiedene Strategien betrachtet werden, was im Falle eines Systemausfalles oder einer verlorenen Nachricht geschieht.

2.3.2. Tools

Nachfolgend werden einige Tools zur parallelen Programmierung vorgestellt. Dabei werden vor allem die Programmiersprachen C und Java betrachtet. Einerseits darum, weil in diesen Programmiersprachen sehr viele Bibliotheken und Frameworks vorhanden sind und andererseits weil die Autoren dieser Arbeit in diesen Sprachen das grösste Know-how besitzen.

Thread-Programmierung

In vielen Programmiersprachen wird die Thread-Programmierung unterstützt. Bei Programmiersprachen wie Java gehören Threads zur Programmiersprache dazu. In C kann für die Thread-Programmierung unter Linux beispielsweise die Bibliothek *Pthreads*[1] verwendet werden. Diese Bibliotheken befinden sich auf einem tiefen Abstraktionsniveau. Es gibt aber auch Frameworks, die eine einfachere, abstraktere Verwaltung von Threads erlauben:
In C, C++ und Fortran existiert beispielsweise die API *OpenMP*[2] für die Shared-Memory-Programmierung. Durch diese API können Direktiven in den Programmcode eingefügt werden, welche bei der Kompilierung durch OpenMP in parallelen Code umgewandelt wird. Somit lassen sich beispielsweise ohne viel Aufwand Schleifen durch eine Zeile Code parallelisieren.
Die Programmiersprache Java hat Frameworks in der Sprache eingebaut, welche den Umgang mit Threads erleichtern. So können Thread Pools[3] erstellt werden, welche die effiziente Nutzung mehrerer Threads erlauben. Anstatt einen Thread zu erzeugen wenn er gebraucht wird, wird zu einem Zeitpunkt eine gewisse Anzahl Threads gestartet. Diese werden in einem Pool verwaltet. Wenn nun ein Thread benötigt wird, kann ein Thread aus dem Pool genommen werden, wodurch die aufwändige Thread-Erzeugung entfällt. Ausserdem wird verhindert, dass zu viele Threads erstellt werden. Wenn viele Tasks abgearbeitet werden müssen, werden diese in eine Warteschlange eingefügt und vom Framework auf Threads abgebildet. Zusätzlich wird von Java ein Fork/Join Framework[4] angeboten, welches die Umsetzung des gleichnamigen Parallel Programming Pattern erleichtert.

Message Passing

Es existiert ein standardisiertes und portables Message Passing Interface (MPI[5]), um parallele Programme zu schreiben. Einige Open-Source-Implementationen wie *OpenMPI*[6] bieten Anbindungen an verschiedene Programmiersprachen an. Mittels MPI lassen sich parallele Programme erstellen, die aus mehreren Prozessen bestehen und mittels Message Passing Daten austauschen. Viele Implementationen erlauben auch den Nachrichtentransfer über das Netzwerk. Message Passing bezeichnet den Datenaustausch zwischen Sender und Empfänger, ohne dass ein gemeinsamer Speicher vorhanden ist. Die ausgetauschten Datenpakete werden dabei als Message bezeichnet.

Verteilte Systeme

Um eine Applikation über mehrere Computer zu verteilen, könnte MPI verwendet werden. Für gewisse Programmiersprachen gibt es jedoch keine oder keine offizielle Implementation von MPI. Für Java gibt es aber einige Alternativen, die das Erstellen einer verteilten Applikation erlauben. Mit *JPPF*[7] lässt sich ein Problem aufteilen und gleichzeitig auf verschiedenen Computern rechnen. Typischerweise werden Master/Slave-Architekturen erstellt, wobei der Master die Arbeit auf die Slaves verteilt. Diese Arbeit wird als Job bezeichnet und von den Slaves verarbeitet. Mittels JPPF können verteilte Anwendungen

[1] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html
[2] http://openmp.org/
[3] http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html
[4] http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
[5] http://www.mcs.anl.gov/research/projects/mpi/
[6] http://www.open-mpi.org/
[7] http://www.jppf.org/

einfach, plattformunabhängig und skalierbar erstellt werden. Eine Integration in eine Cloud-Umgebung ist ebenfalls möglich.
Ein weiteres populäres Framework für Simulationen ist das Java-Framework *JADE*[8] (Java Agent Development Framework). Mittels JADE lassen sich Multiagentensystem entwickeln. Agenten sind dabei mehrere gleichartige oder unterschiedlich spezialisierte Einheiten, die kollektiv ein Problem lösen. Jeder Agent wird von JADE als Thread abgebildet. Agenten lassen sich auch über Systemgrenzen hinweg verschieben.

Cloud

In der Cloud lassen sich die im vorherigen Kapitel erwähnten Frameworks ebenfalls verwenden, wenn der Zugriff auf die Rechenressourcen gewährleistet ist. Einige Frameworks bieten ebenfalls eine Cloud-Integration an. Ein populäres Cloud-Framework ist *Hadoop*[9]. Mittels Hadoop lassen sich ebenfalls skalierbare, verteilte Anwendungen entwickeln. Es lässt sich auch in die freie Cloud-Computing-Umgebung *OpenStack*[10] einbinden. Hadoop basiert auf dem MapReduce-Algorithmus und ermöglicht es, intensive Rechenprozesse mit grossen Datenmengen durchzuführen.

Aktorenmodell

Für die Umsetzung des Aktormodells gibt es ebenfalls Frameworks. Für Java- und Scala-Applikationen gibt es das Akka-Framework. Durch Akka[11] lassen sich Aktoren programmieren, welche Nachrichten in Form von Java-Objekten verschicken können. Aktoren werden als Klasse implementiert und können eine anwendungsspezifische Schnittstelle haben (Typed Actor) oder generell alle Arten von Nachrichten empfangen (Untyped Actor). Die Kommunikation erfolgt standardmässig asynchron, kann aber auch synchron erfolgen. Innerhalb der Aktoren kann das Programm sequentiell programmiert werden, denn die Parallelisierung wird dadurch erreicht, dass mehrere Aktoren gleichzeitig eine Nachricht bearbeiten können und nicht dadurch, dass ein Aktor mehrere Nachrichten abarbeiten kann.
Akka kümmert sich um das Scheduling der Aktoren. Es können auch verschiedene Scheduling-Mechanismen für bestimmte Aktoren konfiguriert und implementiert werden. Ausserdem lässt sich festlegen, wie Akka die Aktoren auf Threads abbildet. So können Fork/Join oder einfache Threadpools festgelegt werden. Zusätzlich kann festgelegt werden, wie viele Threads im Minimum und wie viele maximal erstellt werden dürfen.
Das Zustellen von Nachrichten wird durch sogenannte Router vereinfacht. Ein Router kann gewisse Nachrichten beispielsweise an alle ihm bekannten Aktoren mitteilen (Broadcast). Ein anderer Routertyp erlaubt die Zustellung von Nachrichten der Reihe nach (Round-Robin-Router). So wird die erste Nachricht am ersten Aktor zugestellt, die nächste dem zweiten und so weiter.
Akka erlaubt auch sogenannte Remote-Aktoren. Diese Aktoren werden auf einem anderen Computer erstellt und belegen dessen Systemressourcen. Somit lässt sich mit Remote-Aktoren auch ein verteiltes System realisieren. Akka ermöglicht die Konfiguration der verwendeten Transportprotokolle der Nachrichten, Fehlerbehandlungsmechanismen und Serialisierungsmechanismen. Letzteres wird benötigt, wenn Nachrichten über ein Netzwerk ausgetauscht werden sollen. Dazu muss eine Nachricht, welche als Java-Objekt implementiert ist, persistiert werden und kann erst dann verschickt werden.
Aktoren können fehlertolerant Implementiert werden. Das heisst, es kann definiert werden, was mit einem Aktor geschieht wenn ein Fehler auftritt. Standardmässig startet der Aktor einfach neu. Der Aktor bleibt erreichbar und kann in einen gültigen Zustand überführt werden.

[8] `http://jade.tilab.com/index.html`
[9] `http://hadoop.apache.org/`
[10] `https://www.openstack.org/`
[11] `http://akka.io/`

Grafikkarte

Für die Programmierung der GPU existieren ebenfalls mehrere Möglichkeiten. Die standardisierte *OpenCL*[12] Schnittstelle erlaubt die Programmierung der GPU mit einer C-ähnlichen Sprache. Dabei können Berechnungen durchgeführt werden, die nichts mit dem eigentlichen Zweck der Grafikkarte - der Berechnung der Bildschirmausgabe - zu tun haben müssen. Diese Verwendung der Grafikkarte wird auch als *GPGPU* (General Purpose Computation on Graphics Processing Unit) bezeichnet.
Es gibt Sprachanbindungen für OpenCL für eine Vielzahl von Programmiersprachen, unter anderem auch für C und Java. Für Java gibt es beispielsweise die Bibliothek *JOCL*[13], welche sich nahe an der originalen C-API orientiert. Es gibt aber auch das *Aparapi*[14] Projekt, welches es ermöglicht, Java-Code zu schreiben, der zur Laufzeit von Java Bytecode zu OpenCL konvertiert wird.
CUDA[15] (Compute Unified Device Architecture) ist eine ähnliche Schnittstelle wie OpenCL, die jedoch nur für Grafikkarten des Herstellers *NVIDIA* funktioniert.

2.3.3. Skalierung

Von einer parallel implementierten Applikation wird meist Skalierbarkeit gefordert. Dabei ist Skalierbarkeit ein Mass für die Eigenschaft, einen Leistungsgewinn proportional zur Anzahl der verwendeten Prozessoren zu erreichen [9, S. 46]. Dabei wird noch zwischen starker und schwacher Skalierung unterschieden. Um die Leistung und Effizienz eines Programms zu beziffern, werden die in der Literatur [6, S. 56] üblichen Formeln für Speedup (Formel 2.3) und Effizienz (Formel 2.4) verwendet, wobei P die Anzahl Prozessoren, T_1 die Laufzeit mit einem Prozessor und T_P die Laufzeit des Programms mit P Prozessoren bezeichnet.

$$S_P = \frac{T_1}{T_P} \tag{2.3}$$

$$E_P = \frac{S_p}{P} = \frac{T_1}{PT_P} \tag{2.4}$$

Starke Skalierung bedeutet, dass die Problemgrösse konstant bleibt, aber der Speedup und die Anzahl der Prozessoren variiert. Dies wird auch im *Amdahlschen Gesetz* verwendet.
Schwache Skalierung hingegen bedeutet, dass sich die Problemgrösse mit der Anzahl Prozessoren vergrössert. Dies wird in *Gustafson-Barsis' Gesetz* angenommen [6, S. 57].

Im nachfolgenden Abschnitt wird das Amdahlsche Gesetz näher beschrieben.

Amdahlsches Gesetz

Das Gesetz von Amdahl besagt, dass die mögliche Verringerung der Laufzeit eines Programms begrenzt wird. So stellt die Anzahl Prozessoren die obere Grenze für den Speedup dar. Weitere Begrenzungen entstehen durch den verwendeten Algorithmus. Einige sequentielle Teile lassen sich nicht parallelisieren und gewisse Teile sind durch Datenabhängigkeiten nicht parallelisierbar [9, S. 37–38].
Die Laufzeit eines Programms lässt sich in zwei Kategorien einteilen: Zeit, die für die serielle Abarbeitung benötigt wurde (W_{ser}) und Zeit, die für die parallelisierbaren Teile (W_{par}) benötigt wurde. Die Laufzeit eines Programms, das auf nur einer Recheneinheit ausgeführt wird, kann also mit der Formel 2.5 ausgedrückt werden.

[12] `https://www.khronos.org/opencl/`
[13] `http://www.jocl.org/`
[14] `https://code.google.com/p/aparapi/`
[15] `http://www.nvidia.com/object/cuda_home_new.html`

$$T_1 = W_{ser} + W_{par} \tag{2.5}$$

Bei P Recheneinheiten kann die maximale Laufzeit mittels Formel 2.6 ausgedrückt werden.

$$T_P \geq W_{ser} + \frac{W_{par}}{P} \tag{2.6}$$

Dabei werden superlineare Speedups nicht berücksichtigt, die durch Caching-Effekte oder durch parallele Algorithmen, die schneller als der serielle Algorithmus sind, ermöglicht werden [6, S. 57].
W_{ser} und W_{par} kann durch den Anteil an der Gesamtlaufzeit ersetzt werden, wobei f der nicht parallelisierbare Anteil darstellt:

$$W_{ser} = fT_1 \tag{2.7}$$
$$W_{par} = (1-f)T_1 \tag{2.8}$$

Dadurch erhält man das Gesetz von Amdahl:

$$S_P \leq \frac{1}{f + \frac{1-f}{P}} \tag{2.9}$$

Selbst wenn unendlich viele Recheneinheiten zur Verfügung stehen würden, würde der Speedup durch den nicht parallelisierbaren Anteil begrenzt:

$$S_\infty \leq \frac{1}{f} \tag{2.10}$$

Durch Amdahls Gesetz lässt sich ein Eindruck gewinnen, wie die Skalierbarkeit einer Applikation aussehen kann. Dabei spielt der nicht parallelisierbare Anteil der Applikation eine wichtige Rolle. In Abbildung 2.5 (Seite 20) sind verschiedene hypothetische Applikationen oder Algorithmen eingezeichnet. Einer dieser Algorithmen wird als sehr gut parallelisierbar angenommen, wodurch der nicht parallelisierbare Anteil nur 0.1% beträgt. Bei einem anderen Algorithmus beträgt der Anteil 50%. In der Grafik lässt sich ablesen, wie unterschiedlich diese Applikationen oder Algorithmen skalieren. Bereits ab zwei Recheneinheiten unterscheiden sich die Speedups. Ab acht Recheneinheiten kann die blaue Kurve nur noch wenig mehr Speedup erreichen, während der Algorithmus der violetten Kurve fast linear skaliert.

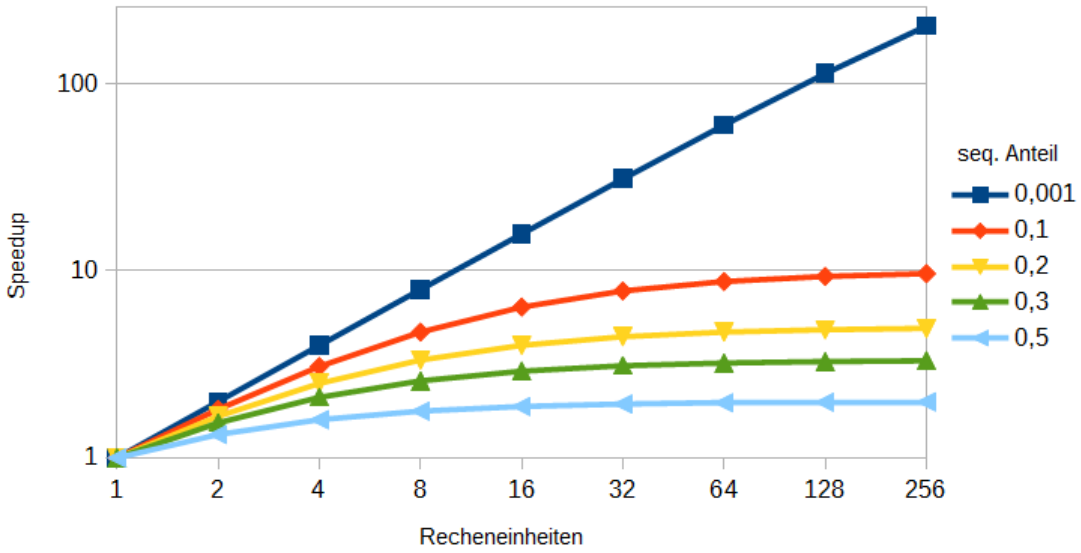


Abbildung 2.5.: Speedup von hypothetischen Algorithmen gemäss Amdahlschen Gesetz

Abbildung 2.6 (Seite 21) zeigt zu den gleichen Applikationen/Algorithmen wie in Abbildung 2.5 die Laufzeit in Abhängigkeit der Anzahl Recheneinheiten. Dabei wird angenommen, dass die Algorithmen/Applikationen auf einer Recheneinheit jeweils eine Laufzeit von 1000 Sekunden haben. Auch hier lässt sich beobachten, dass sich ab einer bestimmten Anzahl Recheneinheiten die Rechenzeit nicht mehr Signifikant verkürzt.

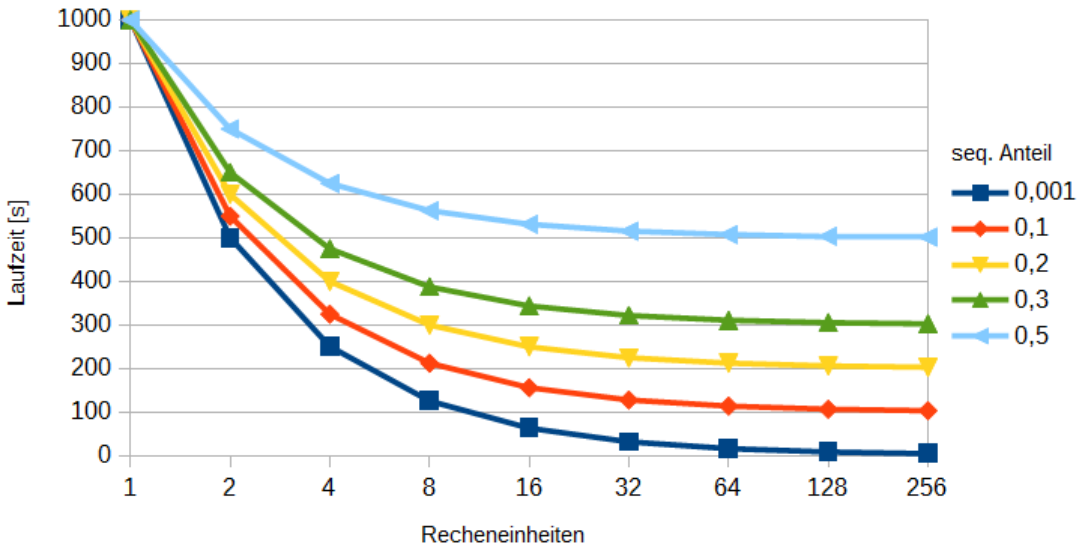


Abbildung 2.6.: Laufzeit von hypothetischen Algorithmen gemäss Amdahlschen Gesetz

3. Vorgehen

Zu Beginn der Arbeit wurden Überlegungen gemacht, welche Art von Fussgängersimulationsmodell implementiert werden soll. Seitens Betreuer und Industriepartner wurde gewünscht, sich am Modell von Moussaïd, Helbing und Theraulaz [7] zu orientieren. Es folgten anschliessend Überlegungen, wie sich dieses Modell implementieren lässt. Nebenbei entstanden im Laufe der Arbeit einige andere experimentelle Prototypen um alternative Ideen und Ansätze zu testen (siehe Abschnitt A.4 auf Seite 86). Für die Implementation der operativen Ebene wurde schliesslich auf einen Strahlabtastungs-Algorithmus (siehe Abschnitt 5.4.2 auf Seite 37) gesetzt. Währenddessen wurde nach möglichen Parallelisierungstrategien gesucht. Dabei wurde der Fokus auf zwei Möglichkeiten gelegt. Diese sind das Zonenmodell sowie der Gruppenansatz (siehe Abschnitt 4 auf Seite 23). Im Rahmen dieser Arbeit wurde das Zonenmodell schliesslich verwendet. Gemäss Betreuer zieht es der Industriepartner in Betracht, eine Cloud-basierte Version ihrer Fussgängersimulation anzubieten. Im Hinblick darauf wurde deshalb in dieser Arbeit ein verteiltes System mit mehreren Rechnern als Zielplattform gewählt. Erste Versuche mit dem Prototypen zeigten jedoch ein schwerwiegendes Problem auf. Bei einigen hundert dicht stehenden Fussgängern war die Simulation bereits sehr langsam und eine Parallelisierung schien erst unmöglich, da es sich nicht vermeiden lässt, dass sich in bestimmten Gebieten Fussgänger dicht drängeln und aufgrund des Zonenmodells ist die langsamste Zone die dominierende Zone. Dieses Problem musste also zwingend gelöst werden. Das Problem konnte dank eines alternativen Algorithmus und einer besseren Fussgängerfilterung mit adaptiver Sichtweite gelöst werden (siehe Abschnitt 5.4.2 auf Seite 36). Im nächsten Schritt konnten nun Messungen mit dem Prototypen getätigt werden. Dazu wurden verschiedene Karten und Szenarien erstellt und jeweils die Gesamtlaufzeit einer Simulation gemessen. Es wurden verschiedene Messungen mit unterschiedlicher Anzahl Zonen, Anzahl Fussgängern und Anzahl Rechner auf verschiedenen Karten getätigt. Die Messungen wurden analysiert und aufgrund der festgestellten geringen CPU-Auslastung wurde nach Optimierungspotenzial im Ansatz, der Implementierung und des Codes gesucht und wo möglich gleich optimiert. Anschliessend wurden neue Messungen mit den vorgenommenen Optimierungen erfasst.

3.1. Testing

Das Testing der operativen Ebene fand durch einen Prototypen statt, der die operative Ebenen visuell Schritt für Schritt aufzeigen kann. Weiter konnte zu Testzwecken die Simulation mit der implementierten GUI beobachtet werden, um unerwartetes Verhalten zu entdecken. Ein automatisiertes Testen wäre nur mit erheblichem Mehraufwand möglich gewesen und deshalb wurde darauf verzichtet.

3.2. Phasen

Die Arbeit wurde in zwei Phasen unterteilt. Diese Aufteilung wurde jedoch erst gegen Ende der Arbeit getätigt. Dies deshalb, da es gegen Ende der Arbeit gelungen war, den Prototypen so zu optimieren, dass neue Messungen gemacht werden mussten.

In dieser Arbeit wird der Teil bis und mit zu den Messungen mit der ersten Version des Prototypen (unoptimiert) als Phase 1 bezeichnet. Die Optimierungen und die mit der optimierten Version durchgeführten Messungen werden als Phase 2 bezeichnet.

4. Parallelisierungsansätze

Um eine Applikation parallelisieren zu können, muss zunächst untersucht werden, welche Bestandteile und Algorithmen sich parallelisieren lassen.
In einer Fussgängersimulation können verschiedene Teile parallel gerechnet werden. Zum Beispiel könnte die Routenberechnung für einen Fussgänger parallel zur operativen Ebene gerechnet werden. Der erwartete Nutzen wäre dabei allerdings vergleichsweise klein, da die Routenberechnung nicht nach jedem Simulationsschritt neu berechnet werden muss. Im Gegensatz dazu muss die operative Ebene für jeden Simulationsschritt berechnet werden. Der Aufwand der Berechnung steigt in Abhängigkeit der Anzahl Fussgänger an. Somit bietet es sich an, die Berechnung der operative Ebene zu parallelisieren, um eine möglichst gute Skalierung der Applikation in Abhängigkeit der Anzahl Fussgänger und der Anlagengrösse zu erreichen.

Zu diesen Überlegungen gibt es auch in der Literatur Theorien, welche diesen Ansatz stützen:

Es gibt zwei Strategien zur Parallelisierung: Datenparallelität verteilt die Daten auf die Recheneinheiten, welche diese jeweils mit dem gleichen Algorithmus verarbeiten. Im Gegensatz dazu werden bei der funktionalen Dekomposition auf den Recheneinheiten verschiedene Algorithmen parallel ausgeführt. Datenparallelität ist dabei zu bevorzugen, denn mittels funktionaler Dekomposition kann nur ein konstanter Speedup erreicht werden. Datenparallelität hingegen wird als Schlüssel zur Erreichung von Skalierbarkeit angesehen. [6, S. 24]

In Fussgängersimulationen müssen verschiedene Daten in unterschiedlichen Zeitabständen und Zeitauflösungen berechnet werden. Die Kernfunktionalität besteht darin, für jeden Fussgänger die nächste Richtung und Geschwindigkeit zu bestimmen und daraus dann die Position zu bestimmen. Dieser Vorgang kann für Fussgänger grundsätzlich parallel durchgeführt werden. Zu Beachten gilt es allerdings, welche Daten zur Verfügung stehen müssen, damit die Berechnung korrekt durchgeführt werden kann:

- Aktuelle Blickrichtung
- Aktuelle Geschwindigkeit
- Zielpunkt, Zwischenziel oder ganze Route
- Fussgänger im Blickfeld der Person
- statische Hindernisse (z.B. Wände) im Blickfeld der Person

Die Berechnung der neuen Parameter des Fussgängers ist also von dessen Zustand und der Zustände der umliegenden Fussgänger sowie der Hindernisse abhängig. Hindernisse werden als statische Objekte angenommen, welche sich nicht bewegen. Deshalb muss die Information über Hindernisse während der ganzen Simulation dem Fussgänger nur einmal mitgeteilt werden. Die restlichen Informationen sind abhängig von der Berechnung des vorherigen Schrittes im Umkreis des Fussgängers. Dies bedeutet, dass verschiedene Fussgänger unabhängig voneinander berechnet werden können und kein Datenaustausch zwischen ihnen notwendig ist. Dies bildet die Grundlage für die nachfolgend vorgestellten Ideen.
Der Gruppenansatz beschäftigt sich mit der Überlegung, ob dicht beieinanderstehende Fussgänger (Gruppe) einzeln betrachtet werden und somit parallelisiert werden können.
Der implementierte Ansatz der Aufteilung des simulierten Geländes in Zonen wird anschliessend thematisiert.

4.1. Gruppenansatz

Aus den vorherigen Überlegungen liegt der Schluss nahe, dass nahe beieinanderstehende Fussgänger als Einheit betrachtet werden können. Diese Einheiten können dann unabhängig voneinander berechnet werden. Diese Einheit wird nachfolgend als Gruppe bezeichnet. Damit die Gruppen isoliert berechnet werden können, darf kein Fussgänger innerhalb dieser Gruppe einen anderen Fussgänger im Sichtfeld haben, der zu einer anderen Gruppe gehört. In diesem Fall müssten wieder nach jedem Schritt die neuen Positionsdaten der Fussgänger zwischen den Gruppen ausgetauscht werden.
Es gilt also zu bestimmen, welche Fussgänger zu einer Gruppe gehören. Diese Zuordnung muss nach jedem Schritt aktualisiert werden. Um die Gruppen zu bestimmen, muss gewartet werden, bis jede Gruppe berechnet wurde. Erst dann kann eine neue Zuordnung eines Fussgängers zu einer Gruppe geschehen. Dies bedeutet, dass die Zuordnungsfunktion vor jedem Berechnungsschritt ausgeführt werden muss. Es ist essentiell, dass diese Funktion sehr effizient durchgeführt werden kann, damit die Parallelisierung einen positiven Effekt auf die Performance der Simulation erzielen kann.

Um die Gruppen zu bestimmen muss allerdings für jeden Fussgänger überprüft werden, ob dieser einen Fussgänger im Sichtbereich sieht, der sich nicht in der gleichen Gruppe befindet.

4.1.1. Nachteile des Ansatzes

Es wird vermutet, dass das Bestimmen der Gruppenzugehörigkeit aufwändig ist, da hierzu der Abstand zwischen allen Fussgängern berechnet werden muss. Zusätzlich gibt es einen weiteren Nachteil. Grosse Menschenmengen, die nahe beieinanderstehen, werden in eine Gruppe zusammengefasst. Innerhalb einer Gruppe werden die Berechnungen sequentiell ausgeführt. Darum wird die Berechnung einer grossen Menschenmenge viel Rechenzeit beanspruchen. Da nach einem Berechnungsschritt auf alle anderen Gruppen gewartet werden muss, muss die Berechnung einer grossen Gruppe abgewartet werden, bevor der nächste Schritt gerechnet werden kann. Die restlichen Gruppen können beliebig klein sein, die Rechenzeit für einen Schritt wird dadurch aber nicht kürzer werden. Die grösste Gruppe definiert also die minimale Rechenzeit eines Schrittes.
Die folgenden Abbildungen zeigen auf, dass bereits ein Fussgänger ausreicht, dass aus einer grossen Gruppe und einer kleinen Gruppe eine grosse Gruppe entsteht. Die Gruppen sind dabei mit roten Kreisen gezeichnet und die blauen Kreise stellen die einzelnen Fussgänger dar.
Abbildung 4.1 (Seite 24) zeigt zwei Gruppen. Die Gruppen können parallel berechnet werden. Die grössere Gruppe (im Bild links) bestimmt dabei die Laufzeit zur Berechnung eines Schrittes. Dabei wird im konkreten Beispiel angenommen, dass die Berechnung der Gruppe mit mehr Fussgängern länger dauert als die Berechnung der kleineren Gruppe.

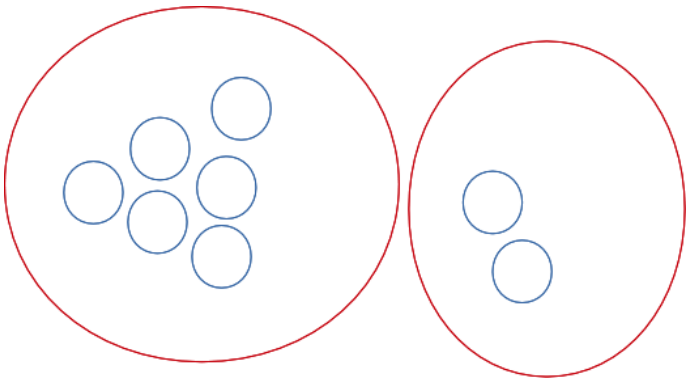


Abbildung 4.1.: Fussgänger, die so weit voneinander weg stehen, dass sie in zwei Gruppen aufgeteilt werden

Abbildung 4.2 (Seite 25) zeigt die gleiche Situation, bei der allerdings ein Fussgänger (mit x gekennt-zeichnet) nahe bei der zweiten Gruppe steht. Im grün gezeichneten Sichtfeld dieses Fussgängers befindet sich ein Fussgänger der anderen Gruppe. Die Information, dass sich im Sichtfeld dieser Person ein anderer Fussgänger aufhält, muss der Person bekannt sein. Bei der eingezeichneten Gruppenaufteilung wäre dies aber nicht der Fall. Deshalb wäre diese Gruppenaufteilung unzulässig.

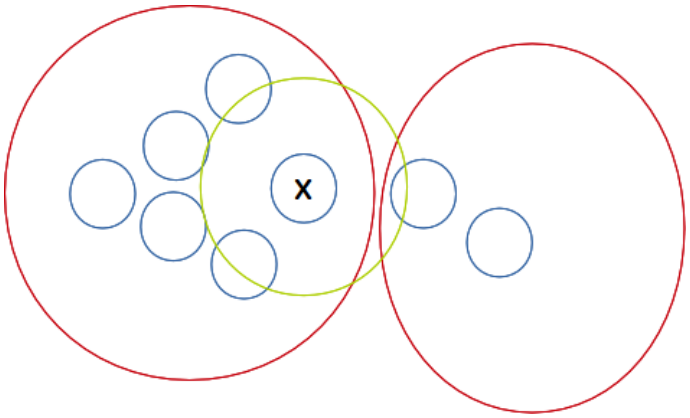


Abbildung 4.2.: Der Fussgänger x verhindert, dass die Fussgänger in zwei Gruppen aufgeteilt werden können

Abbildung 4.3 (Seite 25) zeigt die Gruppenaufteilung, die nötig wäre, um eine korrekte Berechnung durchführen zu können. Durch einen einzelnen Fussgänger kann es passieren, dass Gruppen zusammengeschlossen werden müssen. Im konkreten Beispiel wäre nur noch eine Gruppe vorhanden, was die Parallelisierung verunmöglicht.

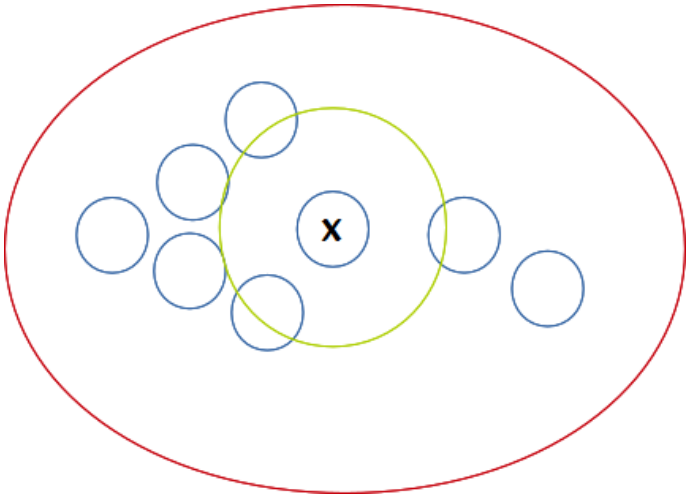


Abbildung 4.3.: Die korrekte Gruppenaufteilung der vorherigen Situation

4.1.2. Verbesserung des Ansatzes

Eine Verbesserung des Ansatzes könnte dadurch erreicht werden, dass es überlappende Bereiche zwischen den Gruppen gibt. In diesen Bereichen müssen die Informationen über die Fussgänger beiden Gruppen zur Verfügung stehen, wobei eine Gruppe verantwortlich für die Berechnung des Fussgängers ist und die andere Gruppe diesen nur als Zusatzinformation verwendet. Dieses Konzept wird im nachfolgenden Abschnitt noch genauer vorgestellt, weshalb an dieser Stelle nicht näher darauf eingegangen wird.

4.2. Zonenmodell

Das Zonenmodell beschreibt einen Ansatz, bei dem der Grundriss der Anlage in Bereiche geteilt wird. Diese Bereiche werden im Rahmen dieser Arbeit als Zonen bezeichnet. Im einfachsten Fall wird die Anlage fix in gleich grosse Zonen eingeteilt. Die Idee zur Parallelisierung basiert ebenfalls auf der Erkenntnis, das Fussgänger unabhängig voneinander berechnet werden können, sobald sie so weit entfernt sind, dass sie sich nicht mehr sehen. Die Zoneneinteilung wird vor dem Start der Simulation vorgenommen. Abbildung 4.4 (Seite 26) zeigt die Aufteilung eines Beispiel-Gebäudegrundrisses. Die schwarzen Rechtecke stellen dabei Wände dar. Die violetten Linien definieren die Zonengrenzen.

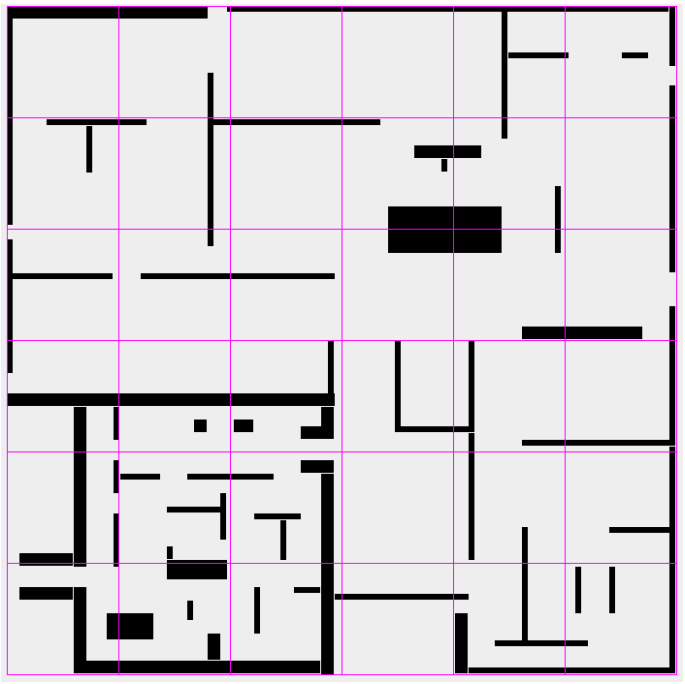


Abbildung 4.4.: Gebäudegrundriss mit Zoneneinteilung

Eine Zone ist für die Berechnung der in ihr enthaltenen Fussgänger zuständig. Zonen können voneinander unabhängig berechnet werden, was die Grundlage für die Parallelisierung bildet. Da Fussgänger sich über Zonengrenzen hinweg bewegen können, muss ein Austausch zwischen den beteiligten Zonen stattfinden. Da die Übergabe an die Nachbarzone zu einem bestimmten Zeitpunkt passiert, muss sich auch die Nachbarzone im selben Simulationsschritt befinden. Dies bedeutet, dass Zonen auf alle umliegenden Zonen warten müssen, bevor sie den nächsten Simulationsschritt berechnen dürfen. Somit besteht eine Barriere nach jedem Simulationsschritt. Die umliegenden Zonen werden in dieser Arbeit als Nachbarzonen bezeichnet. Die Abbildung 4.5 (Seite 27) zeigt eine Zone (rot) mit ihren Nachbarzonen (grün).

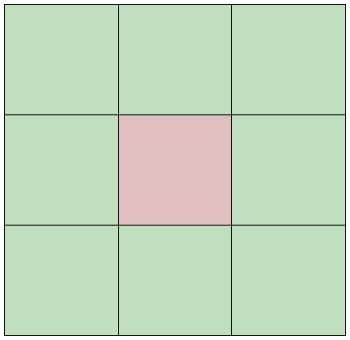


Abbildung 4.5.: Eine Zone und ihre Nachbarzonen

4.2.1. Sichtbarkeitsbereich

Ein Sonderfall entsteht, wenn ein Fussgänger an einer Zonengrenze steht und in die Richtung einer Nachbarzone blickt. In diesem Fall muss die Information vorhanden sein, wo sich in seinem Sichtbereich andere Fussgänger aufhalten. Diese Information liegt allerdings in der Nachbarzone, da diese für die Berechnung dieser Fussgänger zuständig ist. Probleme dieser Art treten bei der Parallelisierung häufig auf, weshalb es auch ein Parallel Computing Pattern gibt, das sich mit diesem Problem befasst. Dieses Pattern nennt sich 'Geometric Decomposition' und beschreibt eine Lösung, indem überlappende Bereiche definiert werden. Innerhalb dieser Bereiche werden die Informationen zwischen den Tasks kommuniziert. [6, p. 100]
Im Bezug auf das Zonenmodell bedeutet dies, dass es einen Bereich innerhalb der Zone gibt, dessen Inhalt nach jedem Berechnungsschritt den Nachbarzonen zur Verfügung stehen muss. Wenn sich also ein Fussgänger in diesem so genannten Sichtbarkeitsbereich befindet, wird nach der Berechnung der neuen Position diese an die Nachbarzone mitgeteilt.
Die Abbildung 4.6 (Seite 27) zeigt den grün eingefärbten Sichtbarkeitsbereich der Zone x. Der Sichtbarkeitsbereich der Zone betrifft alle Nachbarzonen. In diesem konkreten Fall werden die Informationen der Fussgänger, die sich innerhalb des grünen Bereichs der betrachteten Zone x befinden, nach jedem berechneten Schritt an die Zone x geschickt.

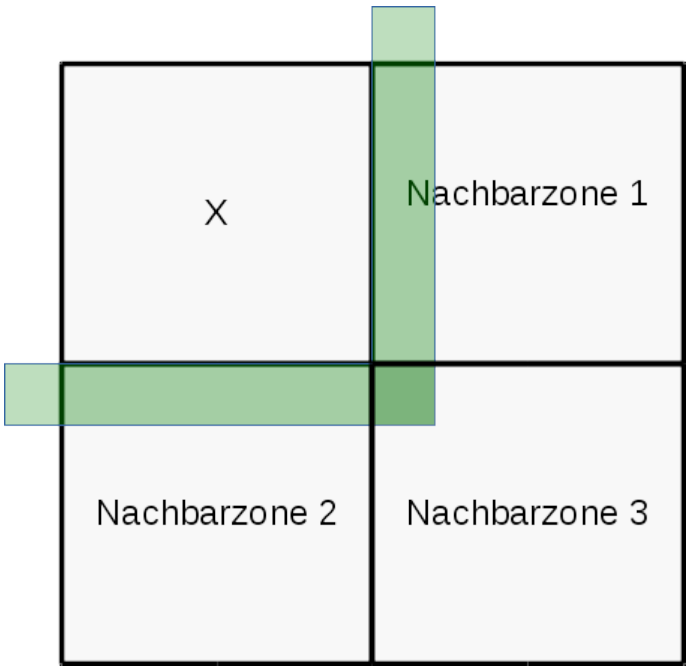


Abbildung 4.6.: Sichtbarkeitsbereich (grün dargestellt) der Zone x

4.3. Weitere Ansätze

Zu Beginn der Arbeit wurden weitere Ansätze geprüft, jedoch schnell wieder verworfen. Ein Ansatz beruht darauf, alle Fussgänger, die auf den gleichen nächsten Zielpunkt zulaufen, isoliert zu betrachten. Die Problematik wird aber schnell ersichtlich. Zwei Fussgängergruppen, die sich in entgegengesetzter Richtung bewegen und dessen Fussgänger jeweils unterschiedliche Zielpunkte besitzen, können sich kreuzen. Sie befinden sich dann jeweils im Sichtfeld des Anderen, wodurch eine parallele Verarbeitung erschwert wird. Die betroffenen Informationen müssten kommuniziert werden. Bei Kreuzungen würde so ein grosser Kommunikationsaufwand entstehen.
Ein anderer Ansatz befasst sich mit den Kanten des Weggraphs. Die Idee ist, alle Fussgänger, die sich entlang der gleichen Wegkante bewegen, parallel zu rechnen. Diese Idee wurde auch schnell wieder verworfen. Zeichnet man einen Weggraph einer Karte, wird ziemlich schnell ersichtlich, dass sich viele Kanten überlagern. Dies ist nicht nur bei Kreuzungen der Fall, sondern kann auch in einfachen Gängen auftreten. Auch hier müsste ein erheblicher Kommunikationsaufwand betrieben werden, um die benötigten Informationen auszutauschen.

5. Implementation

Das Zonenmodell, das im Abschnitt 4.2 auf Seite 26 beschrieben ist, wurde in einem Prototyp implementiert.

Nachfolgend sind verschiedenste Aspekte der Implementation wie Implementationsdetails oder aufgetretene Probleme beschrieben. Als Programmiersprache wurde Java gewählt. Diese Wahl beruht auf der grössten Überschneidung an vorhandenen Sprachkenntnissen der Autoren. Ausserdem bietet Java Plattformunabhängigkeit und ist sehr verbreitet.

5.1. Parallele Programmstruktur

Es gibt verschiedene unterstützende Strukturen für die Formulierung paralleler Algorithmen. Zum Beispiel kann durch Loop Parallelism die Abarbeitung einer Schleife parallelisiert werden.

Das Zonenmodell lässt sich durch verschiedene Strukturen beschreiben. Eine Implementation könnte zum Beispiel Master/Worker für die Aufteilung und Abarbeitung der Arbeit verwenden. Verschiedene Workers könnten dabei Arbeit vom Master anfordern und abarbeiten.
In dieser Arbeit wurde aber der Aktor-Ansatz verwendet. Aktoren sind nebenläufige Einheiten, die über Nachrichten miteinander kommunizieren können. Dabei verfügen die Aktoren nicht über einen geteilten Speicherbereich. Die empfangenen Nachrichten werden pro Aktor in eine Warteschlange gelegt. Sobald eine Nachricht für einen Aktor verfügbar ist und er Rechenzeit erhält, kann diese Nachricht verarbeitet werden. Aktoren ähneln dem Konzept der objektorientierten Programmierung, da sie ihren internen Zustand kapseln. Normalerweise erfolgt die Kommunikation zwischen den Aktoren asynchron.
Das Aktorenmodell schien sich für die Fussgängersimulation zu eignen, da jeder Fussgänger über einen eigenen Zustand verfügt und ein individuelles Verhalten aufweisen kann. Somit könnte jeder Fussgänger als Aktor implementiert werden. In der Implementation wurde jedoch darauf verzichtet, für jeden Fussgänger einen Aktor zu erzeugen, da dadurch der Kommunikationsaufwand wahrscheinlich zu gross würde. Stattdessen wird eine Zone als Aktor implementiert. Die Simulation der Fussgänger innerhalb einer Zone geschieht sequentiell.

5.2. Akka

Zur Implementation der Anwendung mittels Aktoren wurde das Framework Akka (in der Version 2.2.3) verwendet. Akka ist ein Framework, um parallele, verteilte und fehlertolerante Event-basierte Applikationen auf der JVM zu implementieren. Es gibt Programmierschnittstellen zu den Programmiersprachen Java und Scala. Es erlaubt parallele Programme zu entwickeln, ohne sich um die Verwaltung der Threads kümmern zu müssen. Dies wird dadurch erreicht, dass Aktoren vom Framework jeweils auf einen Thread eingeplant werden. Ein Programmierer muss nun die anwendungsspezifische Logik in einer Aktor-Klasse implementieren. Die Kommunikation zwischen den Aktoren erfolgt durch Messages. Diese können ebenfalls mithilfe des Frameworks übermittelt werden.

5.2.1. Untyped Actors

Normalerweise werden in einer Akka-Applikation untypisierte Aktoren verwendet. Diese Art von Aktor besitzt nur eine minimale Schnittstelle, deren einzige Methode dem Zweck dient, eine Message zu verarbeiten. Die Implementation dieser Methode sowie die Messages sind applikationsspezifisch programmierbar. Diese Methode kann nicht von mehreren Threads gestartet werden, deshalb sind Synchronisationsmechanismen nicht notwendig. Dieser Ansatz unterscheidet sich grundlegend vom objektorientierten Ansatz, weshalb es auch einen typisierten Aktor gibt, mit dem sich eine Integration eines Aktors in eine objektorientierte Umgebung einfacher vollziehen lässt. Der Zustand eines Aktors wird wie im objektorientierten Ansatz üblich im Objekt gespeichert.

Die nachfolgende Skizze 5.1 (Seite 30) veranschaulicht das Empfangen einer Nachricht eines Aktors. Die Nachricht wird von einem Aktor gesendet (nicht in Skizze eingezeichnet). Das Akka-Framework legt diese Nachricht in die Message Queue des Empfänger-Aktors, da dieser jeweils nur eine Message verarbeiten kann. Sobald der Aktor vom Framework aktiviert wird, kann dieser die nächste Message aus der Message Queue entfernen und verarbeiten.

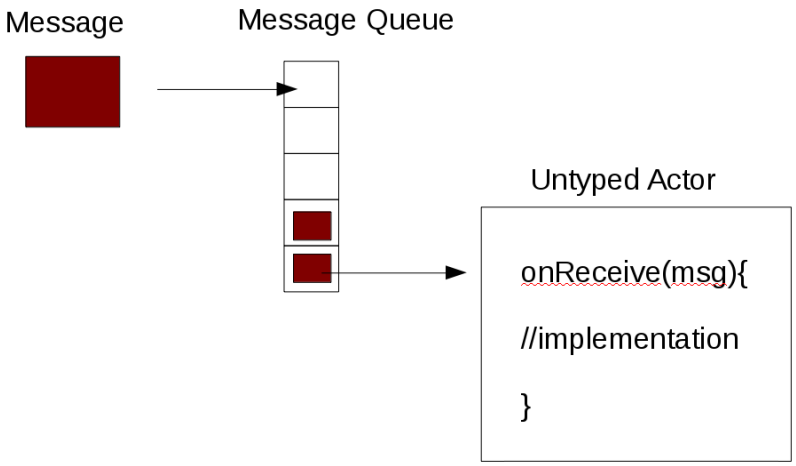


Abbildung 5.1.: Empfangen einer Message in einem Aktor

5.2.2. Typed Actors

Typed Actors bieten die gleiche Funktionalität wie Untyped Actors, mit dem Unterschied, dass sie über eine applikationsspezifische Schnittstelle verfügen. Dadurch kann ein Typed Actor wie ein normales Objekt behandelt werden. Die Integration in die objektorientierte Umgebung kann somit transparent erfolgen. Dies bedeutet, dass ein Subsystem oder Teilmodul der Applikation, das keine Aktoren verwendet, mit Aktoren kommunizieren kann, ohne dass Methoden des Frameworks verwendet werden müssen. Dadurch wird die Kopplung an das Framework verringert. Die Implementation der Aktormethoden müssen ebenfalls nicht synchronisiert werden, da das Framework die Methodenaufrufe in Messages umwandelt, die an den Aktor gesendet werden.

5.2.3. Verteiltes Rechnen

Aktoren werden innerhalb eines Aktorensystems verwaltet. Innerhalb des Aktorensystems können Aktoren miteinander über Messages kommunizieren. Dazu müssen sie den gewünschten Aktor adressieren können. Dies wird in Akka über Aktorreferenzen (ActorRef) ermöglicht. Diese Aktorreferenzen können Computerübergreifend verwendet werden. So können Aktoren erzeugt werden, die sich auf einem anderen Computer befinden. Die Kommunikation zwischen den Aktoren ändert sich im Programmcode nicht. Das Framework erkennt, dass die Message an einen Aktor auf einem anderen Computer gesendet werden soll, serialisiert die Nachricht und schickt sie dann über das Netzwerk an den Ziel-Computer. Die

Konfiguration dieser Remote-Aktoren kann in der Akka-Konfiguration oder direkt im Programmcode vorgenommen werden.

5.3. Softwarearchitektur

Die Simulations-Software ist in mehrere Applikationen gegliedert. Durch das Aktorenmodell und die Möglichkeit durch das Akka-Framework, Aktoren transparent im Netzwerk zu verteilen, kann die Kopplung der Applikation gering gehalten werden. Ausserdem erleichtert die Message-basierte Kommunikation zwischen Aktoren die Kommunikation der Applikationen untereinander.

Die Software besteht aus den folgenden Applikationen:

Master Ist für die Synchronisierung und Lastverteilung der Simulation verantwortlich

Worker Ist für die Anmeldung eines Remote-Computers am Master verantwortlich. Der Computer, von dem sich ein Worker meldet, steht sodann als Rechenressource für die Simulation zur Verfügung.

Client Ist für die grafische Darstellung und die Steuerung der Simulation verantwortlich

Die folgende Grafik 5.2 (Seite 31) zeigt die Applikationen mit ihren Komponenten:

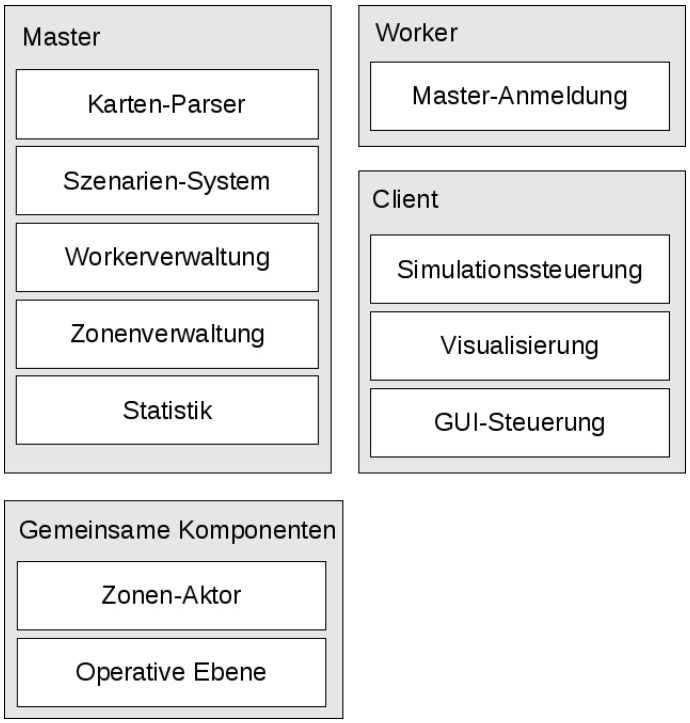


Abbildung 5.2.: Gliederung der Software in Applikationen und Komponenten

Die Komponenten die in der Grafik 5.2 unter *Gemeinsame Komponenten* aufgeführt werden, werden sowohl in der Master- wie auch in der Worker-Applikation verwendet. Die Komponenten werden in den nächsten Abschnitten genauer erläutert.

5.4. Gemeinsame Komponenten

Die folgenden Abschnitte beschreiben die Komponenten genauer, welche in der Master- wie auch in der Worker-Applikation verwendet werden. Dazu gehört die operative Ebene und der Aktor, der für eine Zone zuständig ist.

5.4.1. Zonen-Aktor

Eine Zone wird als Akka-Aktor implementiert. Dieser Aktor ist verantwortlich, dass die Fussgänger, die sich in der zugehörigen Zone befinden, simuliert werden. Um dies zu erreichen, muss auch die Kommunikation mit den Nachbarzonen sichergestellt werden. Ebenfalls muss beachtet werden, dass die Nachbarzonen den vorherigen Simulationsschritt bereits berechnet haben, bevor die Simulation für die betrachtete Zone berechnet wird (Synchronisation).
Für den Ablauf eines Simulationsschrittes sind in den folgenden Grafiken 5.3, 5.4 und 5.5 Flussdiagramme abgebildet. Der gesamte Ablauf wurde auf drei Diagramme unterteilt, um die Übersichtlichkeit zu bewahren.
Abbildung 5.3 (Seite 33) zeigt den groben Ablauf eines Simulationsschrittes. Damit ein Simulationsschritt berechnet werden kann, muss zunächst die Synchronisation mit den restlichen Zonen sichergestellt werden. Danach kann der Rechnungsschritt für alle Fussgänger durchgeführt werden. Nach dem Rechnungsschritt müssen die Daten an die Nachbarzonen gesendet werden. Diese Daten enthalten die Fussgänger, welche sich nach dem Simulationsschritt im Sichtbarkeitsbereich einer Nachbarzone befinden und die Fussgänger, welche die Zone verlassen haben und sich nun in einer der Nachbarzonen befinden. Falls über den Client die GUI aktiviert wurde, werden zusätzlich alle Personendaten an den Client geschickt, wodurch eine Visualisierung der Simulation ermöglicht wird.
Wenn der Simulationsschritt beendet ist, wird dem Master mitgeteilt, dass der Simulationsschritt für diese Zone abgeschlossen ist (Heartbeat).
Als letzte Aktion muss noch der aktuelle Zustand des Aktors neu gesetzt werden. Es kann sein, dass der Aktor bereits Daten von Nachbarzonen bekommen hat, die für den nächsten Simulationsschritt nötig sind. Diese werden nun als Ausgangszustand für den nächsten Simulationsschritt verwendet.

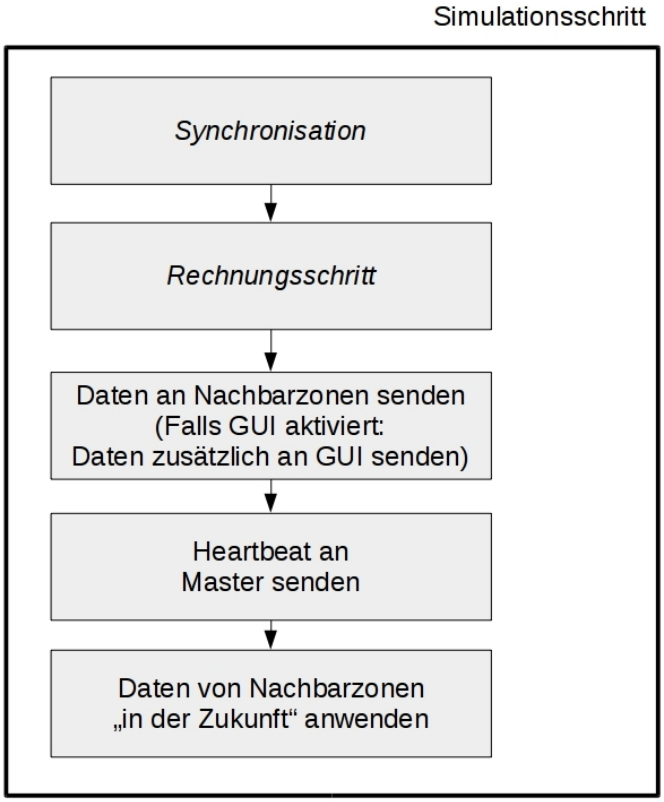


Abbildung 5.3.: Ablauf Simulationsschritt berechnen

In der Abbildung 5.4 (Seite 34) wird die Synchronisation der Zonenaktoren aufgezeigt. Mit der Synchronisation wird sichergestellt, dass der Rechnungsschritt erst dann berechnet wird, wenn alle Daten der umliegenden Zonen des letzten Simulationsschrittes empfangen wurden. Zusätzlich wird gewartet, bis der Master eine Nachricht schickt, welche die Zonen auffordert, den nächsten Simulationsschritt zu berechnen (Master Heartbeat).
Es können auch Daten für den übernächsten Simulationsschritt empfangen werden. Diese müssen speziell behandelt werden. Diese Nachrichten werden zwischengespeichert und erst nach dem Rechnungsschritt berücksichtigt.

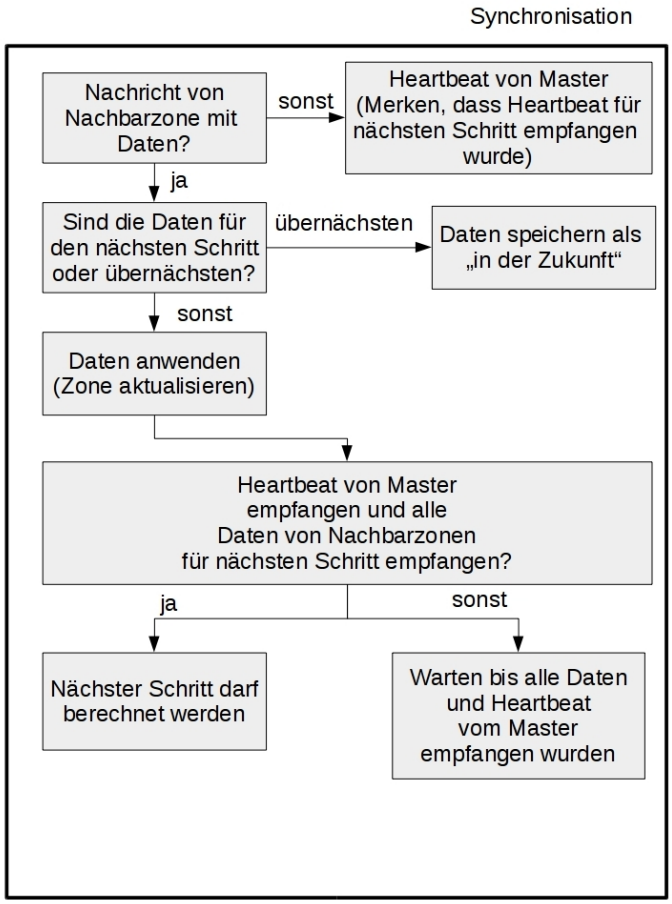


Abbildung 5.4.: Synchronisation des Zonen-Aktors

Der eigentliche Rechenschritt, bei dem für jeden Fussgänger der nächste Zustand berechnet wird, wird in Abbildung 5.5 (Seite 35) aufgezeigt. Zunächst findet eine Vorfilterung statt, die aufgrund der Position der Fussgänger diese in Subzonen einordnet. Für die Berechnung der operativen Ebene müssen dann nur noch Fussgänger in dieser Subzone betrachtet werden, was den Aufwand zur Berechnung pro Fussgänger senkt. Diese Vorfilterung ist im Abschnitt 5.4.1 auf Seite 35 näher beschrieben.
Nach der Subzonen-Filterung wird nochmals gefiltert. Alle Personen, die sich nicht im Sichtradius des Fussgängers befinden, können herausgefiltert werden, damit diese in der operativen Ebene nicht mehr berechnet werden müssen.
Anschliessend kann die operative Ebene gerechnet werden. Diese berechnet für einen Fussgänger die nächste Position, Geschwindigkeit und Richtung. Die Details der operativen Ebene werden im Abschnitt 5.4.2 erläutert.
Nach der Berechnung der neuen Position muss überprüft werden, ob eine Kollision mit einem Objekt stattgefunden hat. In diesem Fall wird die Position des Fussgängers nicht aktualisiert. Anschliessend wird überprüft, ob die Route des Fussgängers neu berechnet werden muss. Wenn dies der Fall ist, wird eine neue Route berechnet.
Am Ende eines Rechnungsschritts wird die Zonenzugehörigkeit des Fussgängers neu bestimmt. Dies beinhaltet eine Überprüfung, ob die Zone verlassen wurde und eine Nachbarzone für die Berechnung des nächsten Simulationsschritts verantwortlich ist und ob sich der Fussgänger nun in einem Sichtbarkeitsbereich einer Nachbarzone befindet.

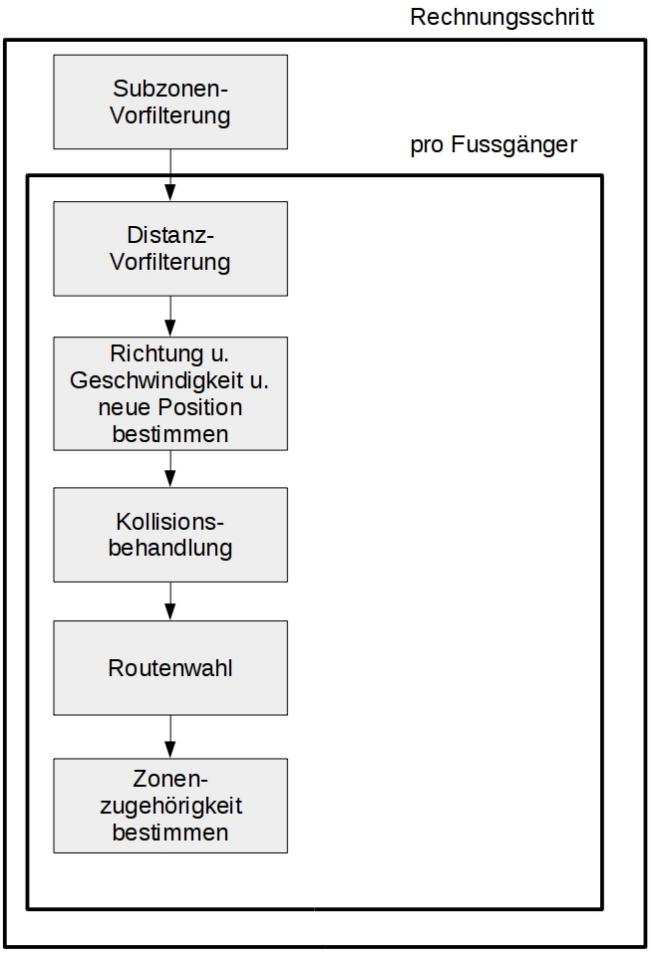


Abbildung 5.5.: Ablauf eines Rechnungsschritts

Subzonen-Filterung

Die Subzonen-Filterung ist eine Filterstufe innerhalb einer Zone. Dazu wird das Prinzip der Zonenaufteilung auf einer weiteren Ebene angewendet. Bei der Aufteilung der Simulationswelt in Zonen wurde dieses Prinzip angewendet, um Daten parallelisiert rechnen zu können. Es kann nun aber auch innerhalb einer Zone angewendet werden, um eine Einteilung der Fussgänger zu ermöglichen. Das Ziel der Filterung ist, dass pro Fussgänger nur die Fussgänger, die sich im Sichtradius der Person befinden herausgefiltert werden, damit die Berechnung der operativen Ebene effizienter durchgeführt werden kann.
Statt die Distanz zwischen jedem Fussgänger zu berechnen, die sich in der Zone befinden, kann dies weiter eingeschränkt werden. Die Zone kann in Subzonen aufgeteilt werden. Danach muss nur noch die Distanz zwischen den Fussgängern, die sich innerhalb dieser Subzonen befinden, berechnet werden. Wie bei den Zonen muss ein Bereich definiert werden, in dem ein Fussgänger auch der Nachbarsubzone bekannt sein muss.

Das Raster wird beim Initialisieren der Zone definiert. Dabei wird die Fläche der Zone durch die Subzonengrösse geteilt.

Die Abbildung 5.6 (Seite 36) zeigt die Zonen Z_1 bis Z_4. In Z_1 wurde die Unterteilung in Subzonen Sz_1 bis Sz_9 (grau hinterlegt) eingezeichnet. Der Sichtbarkeitsbereich der Subzone Sz_5 wurde grün eingezeichnet.

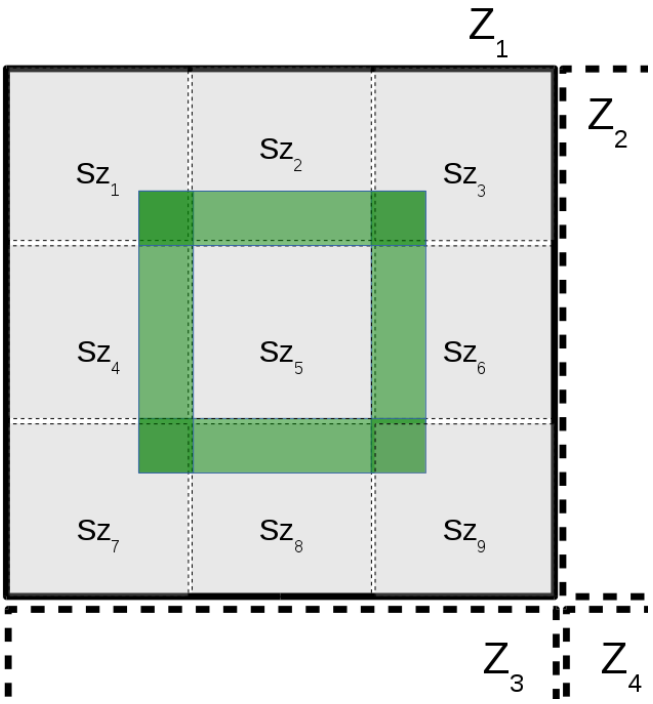


Abbildung 5.6.: Einteilung einer Zone in Subzonen

Das Einteilen der Fussgänger in die Subzonen erfolgt vor jedem Berechnungsschritt. Für jeden Fussgänger wird überprüft, in welcher Subzone sich die Position befindet. Analog zur Zone haben auch Subzonen einen Sichtbarkeitsbereich. Also muss zusätzlich bestimmt werden, in welchen Zonen sich der Fussgänger im Sichtbarkeitsbereich befindet.

Zu jedem Fussgänger wird die Subzone gespeichert, in der er sich befindet. Zur Berechnung der operativen Ebene werden nun die Fussgänger verwendet, die sich in der gleichen Subzone befinden.

Diese zusätzliche Unterteilung in Subzonen vermindert die Laufzeitkomplexität. Für das Filtern der anderen Fussgängern aus dem Sichtbereich würde diese bei $O(N^2)$ liegen, wobei N die Anzahl der Fussgänger in der Zone sind. Dies begründet sich dadurch, dass für jeden Fussgänger eine Distanzberechnung zu jedem anderen Fussgänger durchgeführt werden muss. Erst wenn diese Distanzberechnung durchgeführt wurde, kann entschieden werden, ob sich ein anderer Fussgänger im Sichtbereich aufhalten kann.
Durch die Aufteilung der Zone in k Subzonen, kann die Komplexität auf $O((\frac{N}{k})^2 * k + N * k)$ reduziert werden, wenn man annimmt, dass die Fussgänger innerhalb der Zone gleichmässig verteilt sind. Die Problemgrösse wird damit auf k Subzonen aufgeteilt. Die Distanzberechnungen müssen trotzdem noch gemacht werden. Da es k Subzonen sind, müssen diese k-mal durchgeführt werden. Für das Einteilen der Fussgänger in die richtigen Subzonen muss dann für jeden Fussgänger überprüft werden, ob er sich innerhalb einer der Subzonen befindet.

5.4.2. Operative Ebene

Die operative Ebene muss das Umfeld eines Fussgängers abtasten können, um die Distanz und den Winkel zu Hindernissen wie anderen Fussgängern oder Gebäuden zu ermitteln. Fussgänger werden als kreisrund mit einem gegebenen und den anderen Fussgängern bekannten Körperradius betrachtet. Jeder Fussgänger besitzt ein Sichtfeld und eine Sichtweite. Als Standard wird ein Sichtfeld von 180 Grad und eine Sichtweite von 10 Metern verwendet.

Vorfilterung

Die Vorfilterung bestimmt alle anderen Fussgänger die potenziell im Sichtfeld eines Fussgängers liegen. Diese Vorfilterung senkt den Berechnungsaufwand der Strahlabtastung, da entsprechend weniger Kollisionsüberprüfungen nötig werden. Die Vorfilterung filtert alle anderen Fussgänger mit einer Distanz grösser als der Sichtweite des Fussgängers heraus.

Strahlabtastung

Das Sichtfeld eines Fussgängers wird mittels Strahlen mit Länge des Sichtfelds des Fussgängers vom Mittelpunkt sowie den äussersten Punkten links und rechts in Zielrichtung abgetastet. Kollidiert der Strahl mit einem Hindernis, wird die Distanz zur Kollision vermerkt und um einen bestimmten Winkel δ abweichend von der Zielrichtung je links und rechts der Zielrichtung erneut mit einem Strahl abgetastet. Dieser Winkel wird jeweils so lange erhöht, bis das Sichtfeld des Fussgängers überschritten oder ein kollisionsfreier Winkel gefunden wurde. Anschliessend wird mittels einer Kostenfunktion die neue Laufrichtung des Fussgängers bestimmt. Der Abtastwinkel δ bestimmt die Auflösung des Sichtfelds eines Fussgängers. Beobachtungen der Simulation haben gezeigt, dass 10 Strahlen und somit ein Abtastwinkel $\delta = 18°$ ausreichend ist.

Die folgenden Grafiken zeigen zwei unterschiedliche Situationen, in der die Strahlabtastung durchgeführt wird. Dabei ist der betrachtete Fussgänger rot markiert. Die blauen Kreise stellen die Fussgänger im Sichtbereich des betrachteten Fussgängers dar. Der Fussgänger möchte den Zielpunkt Z (grün eingezeichnet) erreichen. Seine aktuelle Laufrichtung ist mit einem schwarzen Pfeil markiert.

Abbildung 5.7 (Seite 38) zeigt eine Situation, in der fünf Strahlen ausgesendet werden. Die Strahlen sind als rote, schwarze und grüne Linien eingezeichnet und chronologisch von s_1 bis s_5 nummeriert. Als erster Strahl wird s_1 in Zielrichtung ausgesendet. Ein Schnittpunkt mit einem Objekt (Fussgänger oder Gebäude) wird festgestellt und die Kosten für diese Laufrichtung durch die Kostenfunktion bestimmt. Da eine Kollision festgestellt wurde, müssen weitere Strahlen ausgesendet werden. Die nachfolgenden Strahlen werden jeweils im Winkel δ links und rechts vom ersten Strahl ausgesendet. Auch die Strahlen s_2 und s_3 treffen auf Objekte auf. Nach diesem Schritt wird der Winkel α zwischen dem ausgesendeten Strahl und der Zielrichtung um δ erhöht. Bei s_5 wird keine Kollision festgestellt. Dies bedeutet, dass es keinen besseren Weg mehr geben kann, wenn weitere Strahlen ausgesendet werden, weil dazu der Winkel zwischen dem Strahl und der Zielrichtung grösser würde. Dieser Winkel würde höhere Kosten verursachen. Darum kann das Aussenden von Strahlen beendet werden, sobald ein Strahl ausgesendet wurde, der mit keinem Objekt und keinem Fussgänger kollidiert.
Dies ist aber nicht das einzige Abbruchkriterium. Da ein Fussgänger ein bestimmtes Sichtfeld hat, ist es nicht nötig, Strahlen zu senden, die ausserhalb dieses Sichtfelds gesendet würden. Deshalb wird bei jeder Vergrösserung des Winkels geprüft, ob der Winkel zwischen dem Strahl und der aktuellen Laufrichtung im Sichtfeld liegt.

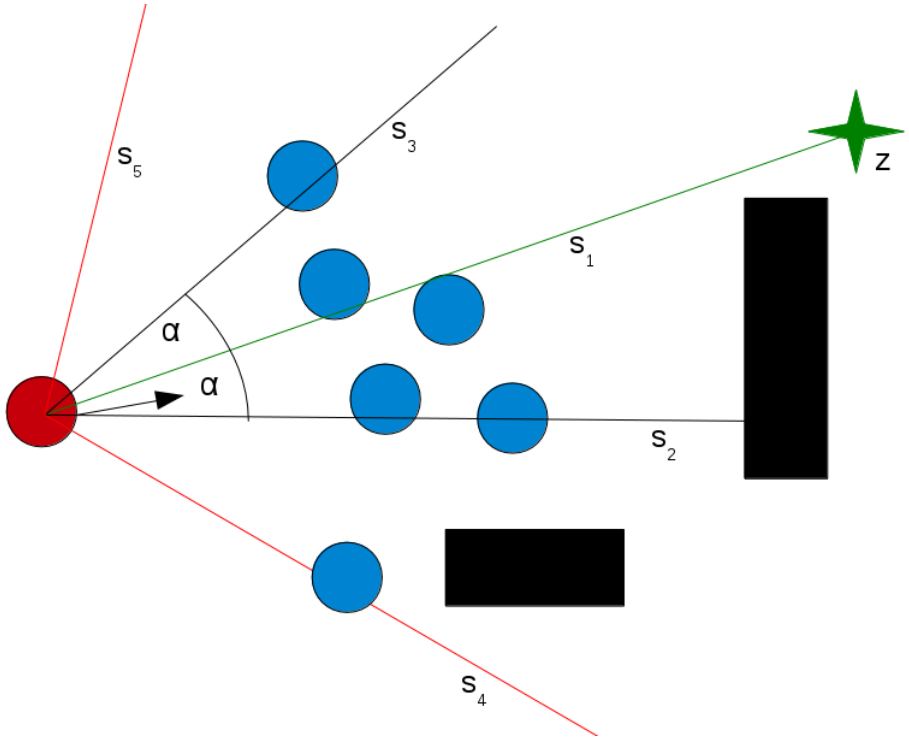


Abbildung 5.7.: Strahlabtastung des Sichtfeldes eines Fussgängers

Bei der zweiten Situation, die in Abbildung 5.8 (Seite 38) dargestellt ist, wird bereits bei s_3 kein Schnittpunkt mehr gefunden. Deshalb kann hier bereits nach drei ausgesendeten Strahlen abgebrochen werden.
Dies bedeutet, dass der Algorithmus besonders dann effizient ist, wenn sich wenige Hindernisse zwischen dem Fussgänger und seinem Zielpunkt befinden. Wenn die Zielrichtung frei ist, muss nur dieser eine Strahl ausgesendet werden. Wenn sich hingegen viele Hindernisse um eine Person herum befinden, muss im schlimmsten Fall der komplette Sichtbereich der Person abgetastet werden.

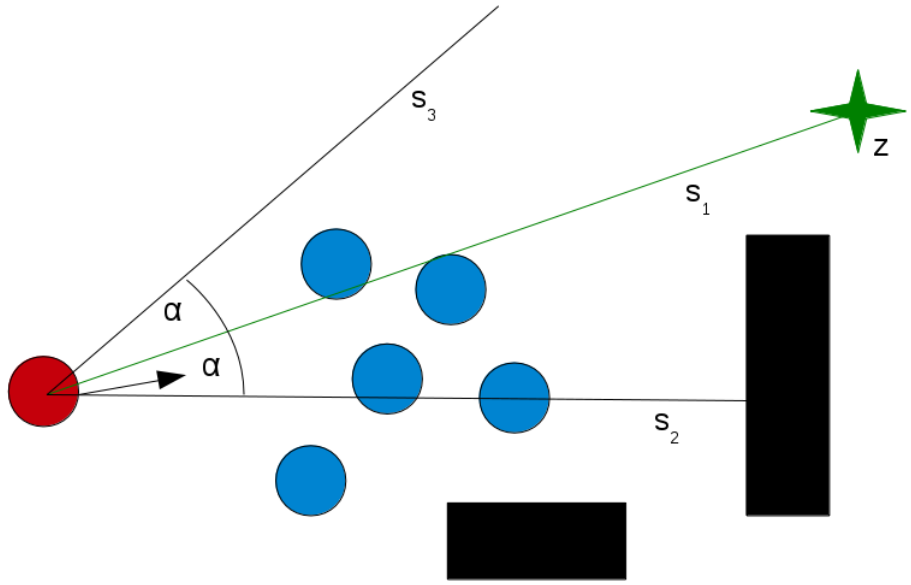


Abbildung 5.8.: Strahlabtastung kann bereits nach dem dritten Strahl beendet werden.

Die Abbildungen 5.7 und 5.8 verwenden eine vereinfachte Darstellung der Strahlabtastung. In der Implementation werden für einen Strahl jeweils drei Strahlen ausgesendet. Dies ist notwendig, weil Fussgänger einen Umfang haben und somit auch überprüfen müssen, ob ein Fussgänger auch mit seinem Körperumfang den eingeschlagenen Weg ohne Kollisionen begehen kann.

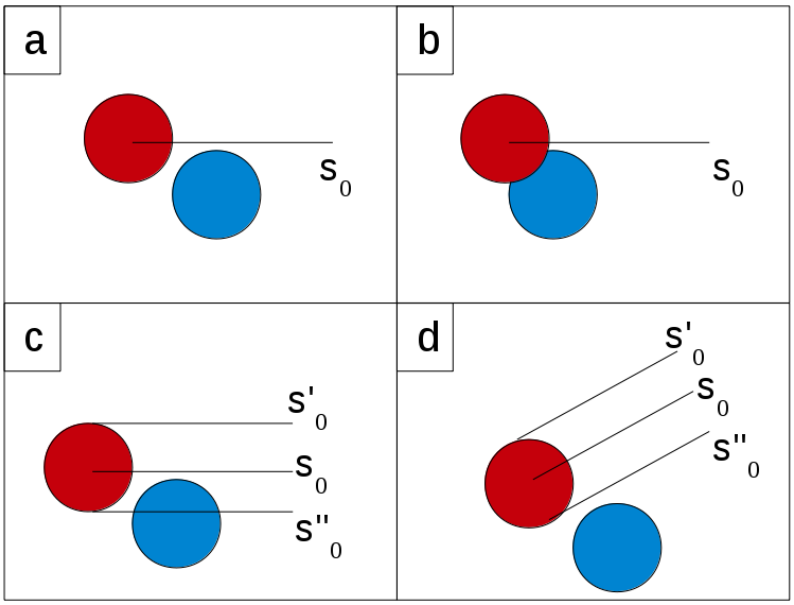


Abbildung 5.9.: Vergleich zwischen Strahlenabtastung mit einem bzw. drei Strahlen

Die Abbildung 5.9 (Seite 39) zeigt die Problematik anhand einer einfachen Situation. Die Ausgangssituation ist in Bild A gezeichnet. Der rote Fussgänger möchte sich in horizontaler Richtung bewegen. Durch die Strahlabtastung mit s_0 würde die Kollision mit dem nahestehenden blauen Fussgänger nicht erkannt. Würde sich der rote Fussgänger nun fortbewegen, würde er wie im Bild B dargestellt mit dem blauen Fussgänger kollidieren.
Darum werden in der implementierten Version (Bild C) in der Ausgangssituation drei Strahlen gesendet. Dabei werden die zwei zusätzlichen Strahlen jeweils vom linken bzw. rechten Rand des Körpers ausgesendet. Die Kollision würde in s''_0 erkannt und ein kollisionsfreier Weg gefunden werden. Der rote Fussgänger würde somit seine Richtung ändern und an der blauen Person vorbeilaufen.

Die Kollsionsprüfung eines Strahls mit einem Fussgänger wird mittels eines Gleichungssystem gelöst. Dabei werden die Schnittpunkte einer Gerade (dem Strahl) und eines Kreises (des Fussgängers) bestimmt.

Dichteproblem

Der Berechnungsaufwand der Strahlabtastung steigt, je mehr andere Fussgänger im Sichtfeld eines Fussgängers sind. Die Wahrscheinlichkeit, dass der Weg in Zielrichtung frei ist, sinkt mit der Dichte der Fussgänger. Ist ein Fussgänger komplett von anderen Fussgängern umgeben, muss immer das gesamte Sichtfeld abgetastet werden, da kein kollisionsfreier Winkel gefunden und der Algorithmus daher nicht frühzeitig abgebrochen werden kann. Es hat sich gezeigt, dass bei vielen dicht stehenden Fussgängern der Berechnungsaufwand erheblich steigt. Da der gewählte Parallelisierungsansatz den Nachteil hat, dass die langsamste Zone die Gesamtrechenzeit bestimmt, wäre es verheerend, wenn in einer Zone eine kritische Dichte erreicht würde. Eine einzelne Zone mit einigen hundert dicht stehenden Fussgängern würde die ganze Simulation ausbremsen, selbst wenn die anderen Zonen tausende von weniger dicht stehenden Fussgängern schnell simulieren könnten.

Adaptive Sichtweite

Je weniger andere Fussgänger zur Berechnung der operativen Ebene eines Fussgängers in Frage kommen, desto weniger Berechnungsaufwand benötigt sie. Es darf jedoch nicht grundsätzlich die Sichtweite aller Fussgänger eingeschränkt werden, denn dann würde auch der Grad an Realismus eingeschränkt werden. Bei dicht stehenden Gruppen von Fussgängern wäre jedoch auch real die Sichtweite eines Fussgängers eingeschränkt, da die Sicht durch die anderen Fussgänger versperrt wird. Es bietet sich daher an, die Sichtweite von dicht stehenden Fussgängern zu reduzieren, während die Sichtweite der weniger dicht stehenden Fussgängern nicht eingeschränkt wird. Beobachtungen der Simulation haben gezeigt, dass keine Einbusse an Realismus aufgrund der eingeschränkten Sichtweite bei dicht stehenden Fussgängern beobachtbar ist.

Dichtefaktor und Sichtweite

Die Sichtweite wird anhand des Dichtefaktors ρ bestimmt. Die Berechnung von ρ betrachtet die anderen Fussgänger in einem kleinen Umkreis (Standardradius 2 Meter) und innerhalb der Standardsichtweite S. Fussgänger, die innerhalb der Standardsichtweite sind, werden jedoch nur leicht gewichtet. ρ berechnet sich wie folgt ($N =$ Anzahl anderer Fussgänger), wobei d_i der Abstand der Mittelpunkte zwischen dem Fussgänger und dem anderen Fussgänger und B der Standardkörperradius ist:

$$\rho = \text{min}(\frac{\sum_{i=1}^{N}(c(d_i)) \cdot B^2 \cdot \pi}{\pi \cdot 2}, 1.0)$$

$$c(d) = \begin{cases} 1 + 0.01 & \text{für } (d \leq 2 \wedge d < S) \\ 0.01 & \text{für } (d > 2 \wedge d < S) \\ 1 & \text{für } (d \leq 2 \wedge d \geq S) \end{cases}$$

Ein Dichtefaktor von $\rho > 0.8$ wird bei ca. 26 Fussgängern erreicht. Die Berechnung der effektiven Sichtweite s_e eines Fussgängers ist wie folgt:

$$s_e = s_m + ((S - s_m) \cdot e^{-\rho * 4})$$

s_m ist die notwendige Mindestsichtweite, die ein Fussgänger haben muss, damit die operative Ebene noch korrekt rechnen kann und beträgt $2.1 \cdot B$. Da jedoch $e^{-\rho * 4}$ für $\rho = 1$ nicht 0 ist, ist die effektive Sichtweite s_e mit den so gewählten Faktoren immer grösser als die notwendige Mindestsichtweite und beträgt ca. 0.7 Meter.

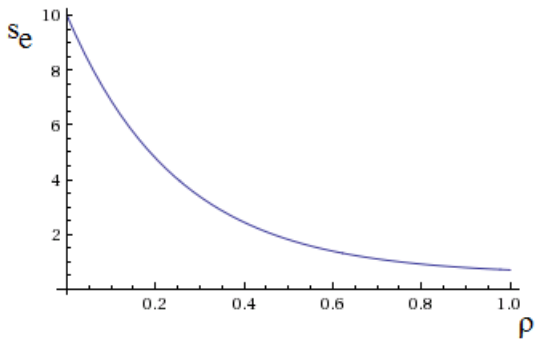


Abbildung 5.10.: Sichtweite in Abhängigkeit von ρ ($B = 0.25, S = 10$)

Algorithmuswechsel bei kritischer Dichte

Obwohl der Berechnungsaufwand bei dicht stehenden Fussgängern aufgrund der adaptiven Sichtweite massiv gesenkt werden konnte, war er immer noch hoch. Es wurde deshalb versucht, auf einen anderen Algorithmus anstelle der Strahlabtastung zu wechseln. Es hat sich jedoch gezeigt, dass der alternative Algorithmus bei wenig dicht stehenden Personen kaum einen Vorteil bringt, jedoch bei dicht stehenden Fussgängern schneller als die Strahlabtastung ist. Eine kritische Dichte gilt als erreicht, wenn der Dichtefaktor ρ grösser als 0.8 ist.

Alternativer Algorithmus

Der alternative Algorithmus wurde inspiriert vom sog. Z-Buffer[1] aus der Computergrafik. Anstatt Strahlen ausgehend vom Fussgänger zu benutzen, wird der Winkel zusammen mit der Distanz und dem Winkel den ein anderer Fussgänger im Sichtfeld einnimmt berechnet. Diese Informationen werden auf einen 1-D Buffer projiziert, der von $-\frac{G}{2}$ bis $\frac{G}{2}$ geht, wobei G dem Sichtfeld in Grad entspricht. Jedes Element im Buffer deckt eine Winkelbreite von δ ab, wobei δ der Auflösung des Sichtfelds entspricht. Für jeden anderen Fussgänger wird – abhängig davon, unter welchem Winkel er zum Fussgänger in Referenz zur Blickrichtung steht und wie viel Winkel er vom Sichtfeld einnimmt – dessen Distanz zum Fussgänger im Buffer abgetragen. Ist der eingenommene Winkel grösser als die Auflösung, wird die Distanz entsprechend in mehrere Elemente im Buffer eingetragen. Befindet sich an einer Stelle bereits ein Distanzwert im Buffer, so wird jeweils das Minimum genommen. Wurde dies für jeden anderen Fussgänger getätigt, wird der Buffer einmalig durchlaufen, um den Winkel mit den niedrigsten Kosten zu bestimmen, den der Fussgänger dann als neue Richtung einnimmt. Da ein Element im Buffer mehrere Winkelgrade umfasst, wird der Mittelwert aus dem Startwinkel und dem Endwinkel des Elements im Buffer gebildet. Die Abbildung 5.11 (Seite 42) zeigt dieses Verfahren graphisch auf. Die Blickrichtung ist als grün, die Richtung zum anderen Fussgänger als rot gezeichnet. α ist der Winkel unter dem der andere Fussgänger sichtbar ist und β ist die halbe Winkelbreite, die der andere Fussgänger im Sichtfeld einnimmt. Die Grafik nimmt an, dass beide Fussgänger gleich grosse Körperradien haben.

Berechnung der Winkel

Der Winkel β lässt sich mittels $tan^{-1}(\frac{s \cdot 2r}{d})$ berechnen, wobei mit $s \cdot 2r$ für $s > 1.0$ erreicht werden kann, dass der Fussgänger versucht, einen grösseren Abstand einzuhalten als notwendig wäre. Sind die Körperradien der Fussgänger nicht identisch, müssen die beiden Radien addiert werden und der Abstand ergibt sich somit mit $s \cdot (r_1 + r_2)$, wobei r_i die Körperradien der Fussgänger sind. Der Winkel α berechnet sich aus dem um $90°$ rotierten Richtungsvektor $\vec{V}$ des Fussgängers und dem Richtungsvektor $\vec{W} = P_2 - P_1$ vom Fussgänger zum anderen Fussgänger, wobei P_2 die Position des anderen Fussgängers und P_1 die Position des Fussgängers ist. α berechnet sich dadurch wie folgt:

$$\alpha = 90° - cos^{-1}(\frac{\vec{V}}{|\vec{V}|} * \frac{\vec{W}}{|\vec{W}|})$$

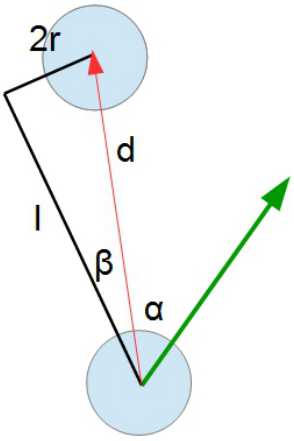
Berechnung des Buffer-Indexes und der Anzahl Elemente

Der Index i im Buffer berechnet sich aus dem Index der Mitte des Buffers $z_m = ceil(\frac{G}{\delta} \cdot 0.5)$, wobei G das Sichtfeld in Grad und δ die Winkelauflösung ist. i ist dadurch gegeben als $i = z_m + (\frac{\alpha}{\delta})$. Die Anzahl Elemente n_β, welche die Winkelbreite β einnehmen, berechnen sich als $\frac{\beta}{\delta}$. Ausgehend vom Index i wird anschliessend im Buffer in die nächsten n_β Elemente sowohl nach links als auch nach rechts die minimale Distanz eingetragen, die ein Fussgänger unter jenem Winkel laufen kann.

Einschränkungen

Der alternative Algorithmus findet nur ab einer kritischen Dichte Anwendung. Entsprechend nimmt er keine Anpassungen an der Laufgeschwindigkeit eines Fussgängers vor und weicht nicht aktiv Mauern aus. Dies begründet sich dadurch, dass Fussgänger die von diesem Algorithmus gerechnet werden sich kaum bis gar nicht bewegen können (nur jeweils die äussersten Personen besitzen Bewegungsfreiheit).

[1] `https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Z-buffering.html`



1-D Buffer

-90..-72	-72..-54	-54..-36 (α)	-36..-18	-18..0	0..18	18..36	36..54	54..72	72..90
∞	∞	d	∞	∞	∞	∞	∞	∞	∞

Abbildung 5.11.: Der alternative Algorithmus der operativen Ebene.

Es ist möglich, alle Fussgänger von diesem Algorithmus rechnen zu lassen, jedoch bemerkt man eine Einbusse an Realismus insbesondere deshalb, da sie Mauern nicht aktiv ausweichen.

Kollisionsverhinderung

Es hat sich gezeigt, dass Kollisionen nicht gänzlich vermieden werden können. Wenn die Kosten für ein Ausweichmanöver von der Kostenfunktion als zu hoch geschätzt werden, würden Fussgänger durch andere Fussgänger hindurchlaufen. Um dies zu verhindern, wurde eine Prüfung eingebaut, ob die berechnete Position eines Fussgängers zu einer Überlappung mit einem anderen Fussgänger oder einer Mauer führen würde. Ist dies der Fall, bleibt der Fussgänger stehen. Falls dieser Fall eintrifft, versucht der Fussgänger, eine zufällige Richtung einzunehmen, da Beobachtungen zeigten, dass so ein realistischeres Verhalten erreicht werden kann. Kann der Fussgänger in der zufällig gewählten Richtung nicht laufen, bleibt er ebenfalls stehen.

Einschränkungen

Da Fussgänger sofort stehen bleiben um im letzten Moment eine Kollision zu verhindern, die nicht vom Abtasten und der Kostenfunktion verhindert werden konnte, entsteht ein unrealistisch schnelles Bremsen, sollte ein Fussgänger zu schnell unterwegs sein. Die Standardgeschwindigkeit eines Fussgängers in der Simulation beträgt 1 Meter pro Sekunde bei einem Zeitschritt von 100 Millisekunden. Dies entspricht 10 cm pro Schritt. Ein Fussgänger kann so 10 cm vor einem anderen Fussgänger stehen bleiben. Bei sehr schnellen Fussgängern (10 Meter pro Sekunde) würde der Fussgänger 1 Meter vor einem anderen Fussgänger sofort stehen bleiben. Um solche Situationen korrekt zu simulieren, müsste auf ein realistischeres Modell gewechselt werden, welches unter anderem auch Bremswege von Fussgängern berücksichtigt und auch nur Ausweichmanöver in Betracht zieht, bei denen die Anforderungen an den Bremsweg eingehalten werden. Ebenso müsste die Geschwindigkeit abhängig von der Distanz zu anderen Fussgängern gemacht werden, damit ein Fussgänger in jedem Fall noch rechtzeitig bremsen kann.

Abdrängungsproblem

Fussgänger können durch Ausweichmanöver in eine Position gelangen, von der sie keine Sicht mehr auf das Ziel haben und alleine durch die operative Ebene nicht mehr zum Ziel finden können. Abbildung 5.12 (Seite 43) zeigt eine solche Situation auf. Zielrichtung und Zielpunkt sind grün gezeichnet. Die operative

Ebene kann keine kollisionsfreie Richtung mehr finden und eine 180° Drehung widerspräche der Funktion der operativen Ebene, möglichst direkt auf das Ziel zulaufen zu wollen. Solche Situationen müssen entsprechend erkannt und gelöst werden. Besteht kein Sichtkontakt mehr zum nächsten Zielpunkt, werden neue sichtbare Wegpunkte bestimmt und von denen aus eine Route zum Endziel gesucht.

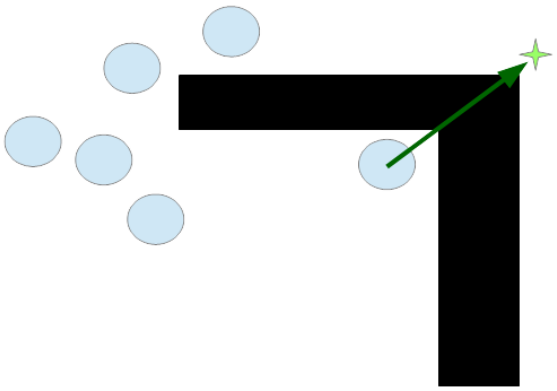


Abbildung 5.12.: Abgedrängter Fussgänger.

Probleme beim parallelen Berechnen

Wird die operative Ebene in Zonen unterteilt und jede Zone für sich parallel berechnet, kann es bei Zonenübergängen zu Kollisionen kommen. Wird sequentiell gerechnet, so hat man eine eindeutige Reihenfolge, in der die Fussgänger ihren Schritt tätigen. Hat ein Fussgänger seinen Schritt getätigt, ist seine neue Position sofort jedem anderen Fussgänger bekannt. Entsprechend wissen im selben Simulationsschritt berechnete Fussgänger bereits diese neue Position. Wird parallel gerechnet, ist diese neue Position nicht während dem Simulationsschritt, sondern erst nach dem Schritt bekannt, da sich Zonen erst nach dem Schritt wieder synchronisieren. Aus diesem Grund kann es vorkommen, dass zwei Fussgänger, die an einem Zonenübergang in jeweils entgegengesetzte Richtungen laufen möchten, den Weg als frei erkennen und deshalb ihren Schritt tätigen. Tun dies beide Fussgänger, überlappen sie sich beim nächsten Simulationsschritt. Abbildung 5.13 (Seite 44) stellt diese Problematik grafisch dar. Schraffiert ist die ungefähre Wunschposition der beiden Fussgänger gezeichnet. Im sequentiellen Fall (links) unternimmt der blaue Fussgänger den ersten Schritt. Da der Weg für ihn frei ist, darf er an seine Wunschposition vorrücken. Anschliessend kommt der magentafarbene Fussgänger zum Zug, der nicht so weit vorrücken darf, da der Weg vom blauen Fussgänger versperrt wird. Im parallelen Fall (rechts) unternehmen beide Fussgänger den Schritt zur selben Zeit. Beide erkennen den Weg als frei und dürfen daher auf ihre Wunschposition vorrücken. Dies führt schliesslich zu einer Überlagerung.

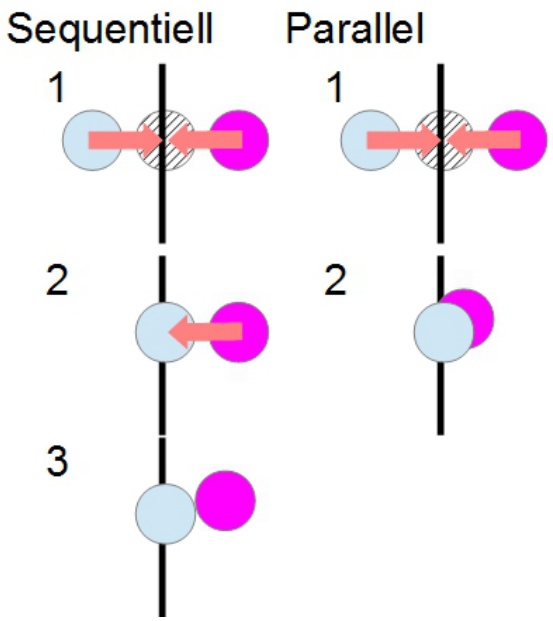


Abbildung 5.13.: Sequentielle und parallele Kollisionserkennung

Lösung des Problems
Für dieses Problem gibt es verschiedene Lösungsansätze. Ein möglicher Ansatz wäre zu spekulieren wie sich Fussgänger verhalten. Dazu könnte man aus Sicherheitsgründen immer davon ausgehen, dass ein Fussgänger in der Nähe des Zonenübergangs im nächsten Schritt überlappen wird und einen zusätzlichen Sicherheitsabstand zu diesem einhalten. Dies hätte jedoch zur Folge, dass Fussgänger in Zonenübergangsnähe nicht mehr dicht aufschliessen und daher eine unrealitisch aussehende Lücke lassen. Andererseits könnte man darauf spekulieren, dass keine Überlappung stattfindet, den Schritt berechnen und den Berechnungsschritt anschliessend verwerfen, sollte die Spekulation falsch gewesen sein. Allerdings bietet dies keine Lösung für den Fall, falls eine Spekulation tatsächlich falsch war. Die einfachste Lösung ist, eine Ordnung zwischen den Fussgängern einzuführen. Jedem Fussgänger wird eine eindeutige und aufsteigende Nummer zugeordnet. Würde ein Fussgänger einem anderen Fussgänger mit einer niedrigeren Nummer bei einem Zonenübergang potenziell zu nahe kommen, verzichtet dieser präventiv um eine potenzielle Überlagerung zu vermeiden. Ändert der andere Fussgänger mit der niedrigeren Nummer jedoch plötzlich seine Richtung, hätte der Fussgänger nicht verzichten müssen. Es ist deshalb nicht ausgeschlossen, dass mit dieser Lösung keine Lücken entstehen, jedoch ist die Wahrscheinlichkeit dazu wesentlich kleiner.

5.5. Master-Komponenten

Nachfolgend werden die Komponenten der Master-Applikation beschrieben. Ausserdem wird die Synchronisation der Zonen erklärt, da dies eine der Hauptaufgaben des Masters ist.

5.5.1. Szenarien

Die Software ermöglicht es, mehrere verschiedene Szenarien zu erstellen und zu simulieren. Ein Szenario beinhaltet:

- Karte
- Strategie zur Erzeugung der Startpositionen der Fussgänger

- Zielpunkt und nächster Punkt auf der Route der Fussgänger
- Individuelle Eigenschaften der Fussgänger (Umfang, Geschwindigkeit, Sichtfeld)

Ein Szenario wird durch eine Java-Klasse repräsentiert. Die Szenarien können über die Konfigurationsdatei ausgetauscht werden. Sie wird beim Start der Master-Applikation geladen.

5.5.2. Karten-Parser

Der Master ist für das Einlesen der Karte verantwortlich. Zudem erstellt er den dazugehörigen Weggraph. Genauere Angaben sind im Abschnitt 5.8 auf Seite 50 zu finden. Der Master teilt die Karte und den Weggraph beim Initialisieren den Zonen-Aktoren mit.

5.5.3. Workerverwaltung

In der Konfigurationsdatei kann spezifiziert werden, auf wie viele Worker gewartet werden soll, bis die Simulation gestartet wird. Ist dieser Wert grösser als Null, wartet der Master, bis er eine Nachricht von allen Workern erhalten hat. Sobald sich genügend Worker gemeldet haben, erstellt der Master die Zonen-Aktoren auf den Computern, von denen sich ein Worker gemeldet hat. Über die Konfiguration kann festgelegt werden, ob auf dem Computer, auf dem sich der Master befindet, auch Zonen-Aktoren erstellt werden sollen.

Für die Zuweisung, welche Zonen auf welchen Workern berechnet werden sollen, wird ein einfacher Algorithmus verwendet. Die Anzahl Zonen wird durch die Anzahl Worker geteilt, dies ergibt die Anzahl Zonen pro Worker. Jedem Worker wird anschliessend diese Anzahl Zonen zugewiesen. Dabei werden die Zonen zeilenweise von links nach rechts nummeriert. Wenn die Workeranzahl kein Teiler der Zonenanzahl ist, erhält der letzte Worker so viele Zonen wie noch übrig bleiben.

5.5.4. Zonenverwaltung

Der Master teilt die Karte in rechteckige Bereiche (Zonen) ein. Dazu existieren in der Konfigurationsdatei zwei Optionen, um festlegen zu können, in wie viele horizontale und vertikale Teile die Karte aufgeteilt werden soll. Für jede Zone wird ein Zonen-Aktor erstellt, der sich je nach Konfiguration auf dem gleichen Computer wie der Master, oder auf einem der Worker befindet.

5.5.5. Statistik

Um die Performance der Applikation zu messen, ist es notwendig, statistische Auswertungen über die Laufzeit zu ermitteln. Dabei wird ein Statistik-Aktor auf dem Master-Computer erstellt. Die Zonen-Aktoren sammeln während der Simulation die Laufzeitinformationen und senden diese Informationen periodisch an den Statistik-Aktor.

Die ermittelten Werte sind der folgenden Aufzählung zu entnehmen:

Gesamtlaufzeit Wie lange dauerte die Simulation von der Start-Message des Masters, bis zur letzten Bestätigungsmeldung des letzten Zonenaktors.

Durchschnittsminimum Pro Schritt wird über alle Zonen die minimale Berechnungszeit berechnet und dann über alle Schritte der Durchschnitt dieser Minimalwerte gebildet

Durchschnittsmaximum Pro Schritt wird über alle Zonen die maximale Berechnungszeit berechnet und dann über alle Schritte der Durchschnitt dieser Maximalwerte gebildet

Durchschnittlicher Durchschnitt Pro Schritt wird über alle Zonen die durchschnittliche Berechnungszeit berechnet und dann über alle Schritte der Durchschnitt dieser Durchschnittswerte gebildet

Summe der Maximalwerte Pro Schritt wird über alle Zonen die maximale Berechnungszeit berechnet und dann über alle Schritte die Summe dieser Maximalwerte gebildet

Anzahl Schritte Wie viele Simulationsschritte wurden berechnet

Zudem werden die Durchschnittswerte und die Summe der Maximalwerte einmal inklusive der Wartezeit und einmal exklusive Wartezeit berechnet. Inklusive Wartezeit bedeutet, dass eine Messung gestartet wird, sobald der Aktor mit der Berechnung eines Simulationsschritts beginnt und endet, wenn dies wieder passiert. Bei der exklusiven Messung wird zum gleichen Zeitpunkt gestartet, aber bereits die Messung beendet, wenn die Berechnung des Simulationsschritts abgeschlossen wurde.

In der Konfigurationsdatei kann die Anzahl Schritte festgelegt werden, die gemessen werden sollen. Wenn die Simulation diese Anzahl erreicht, wird die Simulation beendet und die Statistik wird in der Konsole des Masters ausgegeben.

5.5.6. Synchronisation

Der Master ist für die Synchronisation der Zonen verantwortlich. Dazu schickt der Master für jeden Simulationsschritt eine Message (Master Heartbeat) an alle Zonen. Diese Message erlaubt den Zonen den nächsten Simulationsschritt zu berechnen, wenn bereits alle Informationen der Nachbarzonen eingegangen sind. Sobald eine Zone einen Simulationsschritt berechnet hat, schickt sie dem Master eine Bestätigungsnachricht. Der Master zählt alle Nachrichten der Zonen im aktuellen Schritt. Hat er alle erhalten, wird wiederum ein Master Heartbeat an alle Zonen verschickt.
Für die Synchronisation der Zonen wurde in einer ersten Version der Applikation auf eine zentrale Synchronisation verzichtet. Im nachfolgenden Abschnitt wird auf die Problematik dieses Ansatzes eingegangen.

Probleme ohne zentrale Synchronisation der Zonen

Eine Zone könnte mit dem Berechnen des nächsten Schrittes bereits fortfahren, sobald alle Daten der Nachbarzonen empfangen wurden. Für eine einfache Simulation würde dies auch funktionieren. Das Problem bei diesem Ansatz ist jedoch, dass gewisse Zonen bereits etliche Simulationsschritte mehr berechnet haben als andere.
Dieses Phänomen tritt insbesondere bei Randzonen auf. Abbildung 5.14 (Seite 47) zeigt vier Simulationsschritte in den Bildern 1 bis 4. Die Zonen sind grau gefärbt und beinhalten die Nummer für den aktuellen Simulationsschritt. Anfangs (Bild 1) sind alle Zonen am Berechnen des ersten Schrittes. Sobald eine Zone diesen fertig berechnet hat, schickt sie eine Nachricht an alle Nachbarzonen. Meist können nicht alle Zonen gleichzeitig berechnet werden, da die Anzahl Zonen die Anzahl Recheneinheiten übersteigt. Sobald eine Zone die Nachrichten aller Nachbarzonen erhalten hat, darf sie den nächsten Simulationsschritt durchführen. Dadurch haben Randzonen und insbesondere Eckzonen einen Vorteil, weil diese weniger Nachbarzonen haben und somit weniger lang warten müssen. Betrachtet man die mittlere Zone stellt man fest, dass diese warten muss, bis alle Zonen ihren Schritt berechnet haben. Die Eckzonen hingegen müssen nur auf jeweils drei Nachbarzonen warten. Wenn nicht alle Zonen gleichzeitig berechnet werden können, ergibt sich dadurch eine Situation wie im Bild 2 zu sehen ist. Die Eckzone hat bereits alle Nachrichten der Nachbarzonen erhalten und befindet sich bereits im Simulationsschritt 2. Sobald die umliegenden Zonen der rechten unteren Zone den Simulationsschritt zwei berechnet haben (Bild 3), kann diese bereits den dritten Schritt berechnen. Dabei kann beispielsweise die linke obere Zone noch immer im ersten Simulationsschritt sein. Diese Situation ist in Bild 3 ersichtlich.
Durch die Bedingung, dass jede Zone auf die Nachbarzone warten muss, kann gesagt werden, dass der Simulationsschritt höchstens um eins grösser sein darf als der Simulationsschritt der Nachbarzonen. Dies heisst aber nicht, dass sich alle Zonen höchstens um einen Simulationsschritt unterscheiden.
Die Simulationsschritte können sich maximal um die Anzahl Zonen in der Diagonalen unterscheiden.

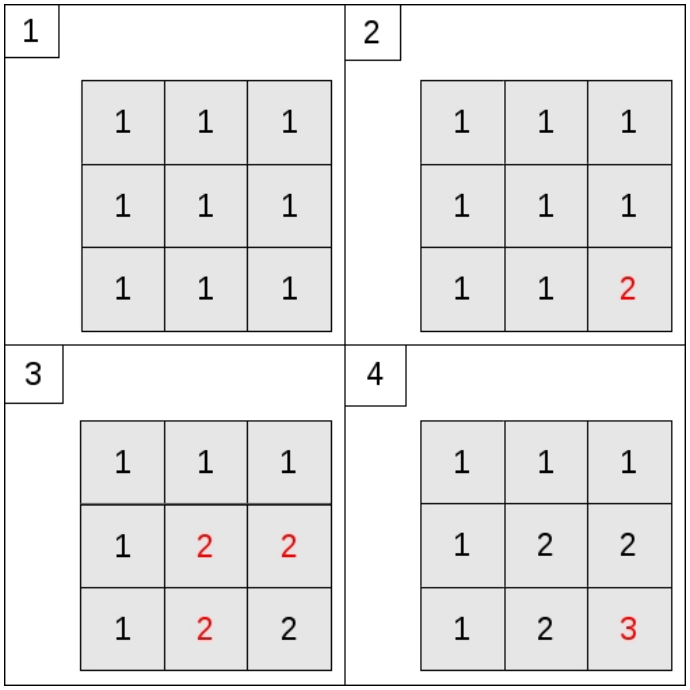


Abbildung 5.14.: Problem bei Synchronisation ohne zentrale Instanz

Auswirkungen
Dieses Verhalten hat Auswirkungen auf die gesamte Simulation. Für eine einfache Fussgängersimulation, welche nur die operative Ebene beinhaltet und keine Ereignisse während der Simulation auftreten können, wäre dieses Verhalten kein Problem.
Wenn die Simulation aber globale Informationen aus den Zonen verwalten soll gibt es Probleme. Denn die Zonen können sich in verschiedenen Simulationsschritten befinden.
Ein Beispiel für solche globale Informationen wäre die Implementation einer taktischen Ebene. Für die Berechnung einer Route für einen Fussgänger unter Berücksichtigung von Staus müsste bekannt sein, an welchen Orten sich wie viele Fussgänger momentan befinden. Diese Information lässt sich aber nur konsistent bestimmen, wenn sich alle Zonen im gleichen Simulationsschritt befinden. Dieses Problem liesse sich lösen, was allerdings zusätzlichen Speicherplatz und/oder Ungenauigkeiten verursacht. Ein einfacher Ansatz würde darin bestehen, die Tatsache zu ignorieren, dass sich Zonen in verschiedenen Simulationsschritten befinden. Dadurch würde es zu Ungenauigkeiten kommen. Wie gross diese Auswirkungen sind, hängt von der Zonenanzahl und der spezifischen Situation ab. Je mehr Zonen verwendet werden, desto mehr Zonen werden sich in der Diagonalen befinden. Dadurch erhöht sich der maximale erlaubte Unterschied, wie viele Simulationsschritte sich Zonen unterscheiden dürfen.
Eine Situation mit negativen Auswirkungen wäre beispielsweise, wenn sich in der Zone, die sich in der Zukunft befindet, innerhalb von wenigen Zeitschritten ein grosser Stau gebildet hat. Ein Fussgänger würde in die Richtung des Staus laufen. Die taktische Ebene würde berechnet und dabei würden die Daten verwendet, die in der Zukunft vorhanden sind. Der Fussgänger wüsste also bereits, dass es in ein paar Simulationsschritten stauen wird.
Ein weiterer Lösungsansatz des Problems würde die Sammlung von Daten beinhalten. Jede Zone könnte pro Simulationsschritt ihren Zustand speichern. Dadurch würde jedoch der Speicherplatzbedarf enorm erhöht. Um dies zu verringern wären weitere Strategien notwendig. Eine Art Mittelwertbildung der Zustände wäre dabei eine Möglichkeit. Eine einfache Möglichkeit könnte darin bestehen, nur bis zu einer maximalen Anzahl Zustände zu speichern und anschliessend die ältesten Zustände zu überschreiben.

Nachrichten aus der Zukunft
Durch das Verhalten ohne zentrale Synchronisation können Nachrichten von Nachbarzonen empfangen werden, die erst für den zukünftigen Simulationsschritt notwendig sind. Deshalb müssen die Nachrichten

mit dem Simulationsschritt gekennzeichnet werden, für welche die darin enthaltenen Informationen gültig sind. Dass solche Nachrichten aus der Zukunft möglich sind, lässt sich durch Bild 2 in Abbildung 5.14 (Seite 47) erklären. Beispielsweise kann die mittlere Zone den nächsten Schritt berechnen, sobald die Nachricht der linken oberen Zone empfangen wurde. Da diese aber sehr viele Fussgänger beinhaltet, dauert es sehr lange bis die Nachricht eintrifft. Gleichzeitig kann die rechte untere Zone aber bereits Simulationsschritt 2 fertig berechnet haben. Die Nachricht der rechten unteren Zone trifft also früher ein. Diese Nachricht wird benötigt um Simulationsschritt 3 zu berechnen. Aber die mittlere Zone hat noch nicht Schritt 2 gerechnet. Darum müssen die Nachrichten aus der Zukunft zwischengespeichert werden. Sobald die Nachricht der linken oberen Zone eingetroffen ist, kann die mittlere Zone Schritt 2 berechnen und anschliessend die Nachrichten, die sie aus der Zukunft bereits erhalten hat, verarbeiten.

5.6. Worker

Die Worker-Applikation dient nur dazu, sich am Master anzumelden. Dazu erstellt der Worker ein Aktorensystem. Anschliessend meldet er sich periodisch beim Master. Dieser kann nun beim Start einer Simulation Zonen-Aktoren auf dem Aktorensystem dieses Computers erzeugen.

5.7. Client

Der Client besteht aus einer grafischen Benutzeroberfläche (GUI). Im oberen Teil der GUI wird die Simulation grafisch dargestellt (Visualisierung). Im unteren Bereich befindet sich ein Textfeld, das als Eingabefeld für Steuerbefehle sowohl für die GUI selbst, wie auch für die Simulation verwendet werden kann. Die Abbildung 5.15 (Seite 49) zeigt die GUI nach dem Start einer Simulation.

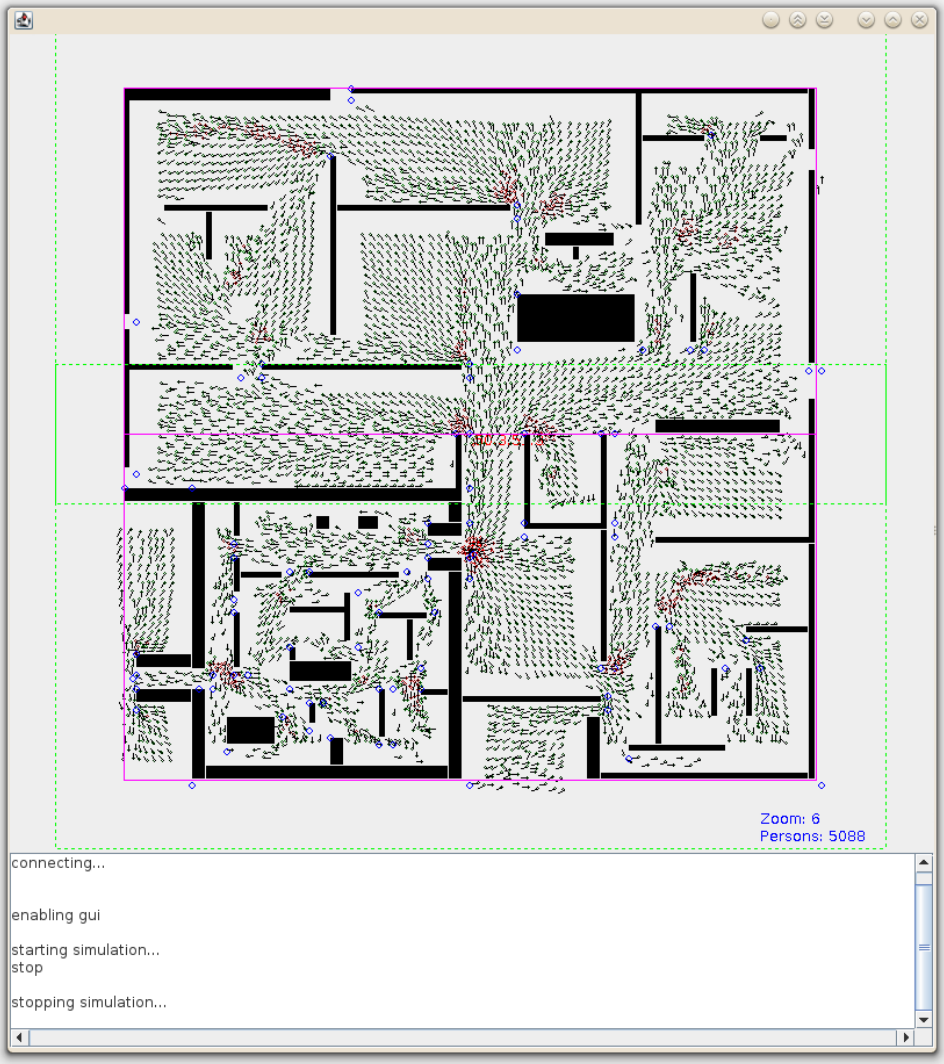


Abbildung 5.15.: Grafische Benutzeroberfläche nach dem Start einer Simulation

Starten einer Simulation

Zum Starten einer Simulation müssen folgende Befehle eingegeben werden:

```
connect
gui enable
start
```

Die vollständige Auflistung der Befehle und deren Bedeutung befindet sich im Anhang auf Seite 85.

5.7.1. Dichteprofil

Alternativ kann die Darstellung der GUI auf eine Dichteprofil-Darstellung eingestellt werden. Das Dichteprofil ermöglicht es zu erkennen, wo und wie stark sich Fussgänger gestaut haben oder welche Pfade oft genommen wurden. Der Vorteil des Dichteprofils ist es, dass es während der Simulation im Hintergrund angefertigt werden kann und es somit nicht erfordert, dass jemand die GUI während der Dauer der Simulation beobachten muss, um Staustellen zu erkennen. Abbildung 5.16 (Seite 50) zeigt ein Beispiel-Dichteprofil einer Simulation. Grüne Stellen sind kaum bis gar nicht von Fussgängern frequentiert, während rote Stellen hoch frequentiert sind. In der Abbildung ist gut ersichtlich, welche Pfade von Fussgängern genommen wurden (dunkelrote Stellen).

Abbildung 5.16.: Dichteprofil einer Simulation

Die Anwendungskomponenten wurden nun vollständig beschrieben. Die nächsten Abschnitte befassen sich mit dem implementierten Raummodell und der taktischen Ebene. Anschliessend werden die Grenzen des Prototypen thematisiert.

5.8. Raummodell

Das Raummodell beschreibt die statischen Hindernisse. Im Prototypen gibt es nur Mauern als statische Hindernisse. Diese Mauern unterliegen der Einschränkung, dass sie achsenorientiert und rechteckig sein müssen. Ein Gebäude oder eine Anlage besteht somit aus einer Menge von Rechtecken.

5.8.1. DXF

DXF (Drawing Interchange File Format) ist ein Dateiformat im CAD-Bereich und wird von einigen Gratis- oder als Demo vorliegenden CAD-Programmen unterstützt [2] [3] . Der Prototyp unterstützt das Laden von DXF-Dateien (der Version R12), die Rechtecke als Polylines enthalten. Diese Rechtecke können verwendet werden, um Wände und andere statische Objekte in einem CAD-Programm zu zeichnen. Der Prototyp benutzt die *kabeja*-Library[4], welche die DXF-Dateien parst.

5.8.2. Weggraph

Der Weggraph wird von der taktischen Ebene benötigt und beschreibt alle möglichen Pfade die ein Fussgänger nehmen kann. Dabei beschränkt er sich nur auf mögliche kürzeste Wege und besteht deshalb aus Verbindungen zwischen Eckpunkten von Rechtecken. Der Graph wird aufgebaut, indem von jedem Eckpunkt zu jedem anderen Eckpunkt eine Linie gelegt wird. Schneidet diese Linie eine Mauer, wird sie verworfen. Schneidet die Linie keine Mauer, wird sie in den Weggraph als mit der Länge gewichtete Kante aufgenommen. Um zu verhindern, dass Fussgänger durch die operative Ebene zu nahe um Ecken laufen, wird ein Sicherheitsabstand zur Ecke verwendet. Aufgrund der Zusammensetzung von Gebäuden oder Hindernissen mittels Rechtecken und des oben genannten Verfahrens, werden innere Ecken teilweise als gültige Eckpunkte für eine Verbindung gewertet und teilweise nicht. Die Abbildung 5.18 (Seite 52) zeigt diesen Sachverhalt grafisch auf. Rot eingezeichnet sind jeweils die Ecken inkl. Sicherheitsabstand des schwarzen Rechtecks. Im linken Fall liegt der Eckpunkt des inneren Ecken links im grünen Rechteck und kommt daher für eine Verbindung nicht in Frage. Ändert man die Zusammensetzung aus Rechtecken wie im rechten Fall, so kann eine Verbindung zum inneren Ecken links gemacht werden. Ein weitere Ungenauigkeit entsteht, wenn zwei Eckpunkte im selben Rechteck liegen und deshalb keine Seite eines Rechtecks schneiden. Diese werden als gültige Kante im Graph eingetragen, welche bei der Routenwahl jedoch nie verwendet werden.

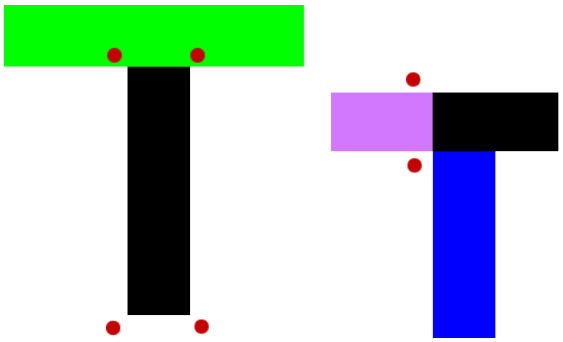


Abbildung 5.17.: Unterschiedliche Zusammensetzung aus Rechtecken kann zu anderen Verbindungen führen.

5.9. Taktische Ebene

Die taktische Ebene beschränkt sich auf die Bestimmung des kürzesten Weges. Dazu wird ein A*-Algorithmus ohne Heuristik [8] verwendet. Ein Fussgänger kennt seinen Zielpunkt, einen nächsten und einen übernächsten Kontrollpunkt (Zwischenziele). Ein Fussgänger steuert durch die operative Ebene solange auf den nächsten Kontrollpunkt zu, bis der übernächste Kontrollpunkt sichtbar ist. Ist dies der Fall, wird der übernächste Kontrollpunkt der nächste Kontrollpunkt und es wird via Wegfindung ein neuer übernächster Punkt bestimmt. Es wäre möglich, einem Fussgänger die vollständige Route

[2] QCAD: http://www.qcad.org/de/qcad-downloads-trial
[3] LibreCAD: http://librecad.org/cms/home.html
[4] http://kabeja.sourceforge.net/

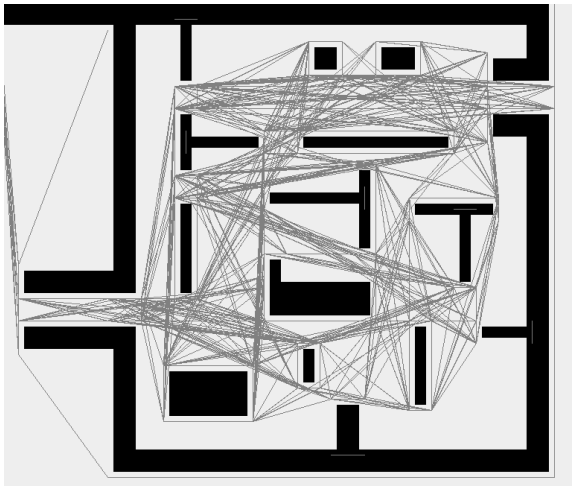


Abbildung 5.18.: Beispiel eines Weggraph.

mitzuteilen. Der oben beschriebenen Ansatz wurde jedoch gewählt, um besser auf sich während der Simulation ändernde Graphen reagieren zu können, sollte diese Funktion implementiert werden. Ändert sich der Graph, müsste dadurch nur der Cache der Routen geleert werden (siehe nächster Abschnitt).

Die Wegfindung wird pro Zone und somit pro Zonen-Aktor durchgeführt. Damit findet bis zu einem gewissen Grad Parallelisierung statt. Innerhalb einer Zone ist die Wegfindung jedoch sequentiell pro Fussgänger.

5.9.1. Routen Caching

Da viele Fussgänger jeweils eine Route vom selben Startpunkt zum selben Zielpunkt benötigen, bietet es sich an, eine gefunden Route zu speichern und bei einer erneuten Anfrage die gespeicherten Routen zu durchsuchen, bevor eine neue Wegfindung berechnet wird. Die Route jedes Mal neu zu berechnen dauert bei grossen Graphen sehr lange (länger als die operative Ebene) und würde eine Simulation in vernünftiger Zeit verunmöglichen.

5.10. Grenzen des Prototypen

Der Prototyp unterliegt Grenzen in der Skalierbarkeit und Stabilität.

5.10.1. Speicher

Damit grössere Weggraphen via Java-Serializer serialisiert und deserialisiert werden konnten, musste explizit die erlaubte Stackgrösse auf mindestens 4MB hochgesetzt werden, denn ansonsten würde die Deserialiserung mit einer Stack-Overflow-Fehlermeldung fehlschlagen. Ebenso benötigt jeder Fussgänger Speicherplatz. Bei grösseren Mengen an Fussgängern muss der Java-Heap vergrössert werden.

5.10.2. Akka

Das Akka-Framework verwendet für die Kommunikation ein Kommunikationsprotokoll, welches unter anderem auch feststellen kann, ob ein Aktor abgestürzt oder unerreichbar ist. Dazu verwendet Akka Heartbeats und einen Transport Failure Detector um fehlgeschlagene Nachrichtenübermittlungen zu detektieren. Ebenso ist konfigurierbar, was die maximale Grösse (Payload Size) einer Nachricht sein

darf. Die Standardwerte für Timeouts und Payload Size waren für den Prototypen ungeeignet. So wurden Aktoren falsch als unerreichbar markiert oder der Verbindungsaufbau zu Aktoren funktionierte nicht. Die Payload Size wurde deshalb massiv erhöht und Kommunikations-Timeouts wurden ebenso erhöht. Die entsprechenden Konfigurationseinstellungen sind im Anhang (Abschnitt A.5.5 auf Seite 93) dokumentiert.

5.10.3. Stabilität

Der Prototyp ist darauf angewiesen, dass keine Nachricht bzw. Daten zwischen Aktoren verloren gehen. Ebenso darf kein Aktor während der Simulation abstürzen oder vom Netzwerk getrennt werden. Eine unterbrochene Simulation kann nicht fortgeführt, sondern muss wiederholt werden.

5.11. Code-Optimierungen am Prototypen

Nachdem erste Messergebnisse vorlagen, wurde nach Optimierungspotenzial gesucht. Dies geschah einerseits manuell und andererseits mittels des Profilers VisualVM[5] für die JVM (Java Virtual Machine). Die getätigten Optimierungen am Code sind nachfolgend dokumentiert.

5.11.1. Veränderbare Vektoren

Viele Berechnungen finden mittels Vektoren statt. Da diese Vektoren teils via Nachrichten mit anderen Aktoren ausgetauscht werden müssen, wurden diese Vektoren als unveränderbar implementiert. Dies, da Akka empfiehlt, nur unveränderbare Daten auszutauschen. Diese Empfehlung beruht darauf, dass keine Garantie besteht, ob das Objekt kopiert wird, oder ob nur eine Referenz des Objekts ausgetauscht wird. Es könnten deshalb Race Conditions auftreten, wenn angenommen wird, dass man auf einer Kopie eines Objektes arbeitet, tatsächlich aber nur eine Referenz und keine Kopie erhalten hat. Mit unveränderbaren Vektoren zu rechnen hat den Nachteil, dass bspw. eine Addition zweier Vektoren einen neuen Vektor erzeugt. Dadurch werden oft viele Objekte erzeugt und wieder verworfen, was unnötige Rechenzeit beansprucht. Wo möglich wurden daher in Phase 2 unveränderbare Vektoren durch veränderbare Vektoren ersetzt.

5.11.2. Benutzung von Bibliotheken

Für einige Geometrie-Probleme wurden, wo möglich, Java-Bibliotheken benutzt, die z.B. Funktionen wie Rotation eines Vektors anbieten. Diese Bibliotheken haben jedoch oft ein objektorientiertes Interface, welches es erforderte, einen Vektor der jeweiligen Bibliothek zu erzeugen, zu rotieren, die Werte auszulesen und in die eigene Vektorklasse zu lesen. Dies ist ineffizient, da so viele Objekte erzeugt und wieder verworfen werden. Die entsprechenden Funktionen wurden daher in Phase 2 selbst implementiert.

5.11.3. Benutzung einer Tabelle zur Berechnung von *acos*

acos (bzw. cos^{-1}) wird zur Berechnung des Winkels zwischen zwei Vektoren benötigt. Es hat sich jedoch gezeigt, dass es schneller ist, statt der eingebauten $Math.acos$ Funktion eine vorausberechnete Tabelle mit einigen Werten zu benutzen. Diese Tabelle umfasst den Wertebereich von -1 bis $+1$ mit einem Schritt von 0.01. Der Index in der Tabelle wird bestimmt durch eine Multiplikation mit $100 = 0.01^{-1}$ plus Addition von 100 (Verschiebung da Wertebereich von -1 bis 1) und anschliessend gerundet auf eine ganzzahlige Zahl.

[5] `http://visualvm.java.net/`

6. Resultate

6.1. Verwendete Hardware

Nachfolgend ist die bei den Messungen verwendete Hardware dokumentiert. Die Kürzel finden im Text und in Legenden von Diagrammen Verwendung.

i5

- Kürzel: i5
- CPU: i5-2520M @ 2.6 GHz
- CPU Cores: 2 (2 Threads pro Core)
- RAM: 8 GB
- OS: Ubuntu 14.04, 64bit
- Java: 1.7.0_55, 64bit

i7

- Kürzel: i7
- CPU: i7s-3632QM @ 2.2 GHz
- CPU Cores: 4 (2 Threads pro Core)
- RAM: 8 GB
- OS: Windows 8.1, 64bit
- Java: 1.7.0_51, 32bit

DSK

- Kürzel: DSK
- CPU: i7-3770 @ 3.4 GHz
- CPU Cores: 4 (2 Threads pro Core)
- RAM: 8 GB (Verfügbar 3.5 GB)
- OS: Windows 7 Enterprise, 32bit
- Java: 1.7.0_51, 32bit

6.2. Verwendete Karten

In diesem Abschnitt sind die verwendeten Karten dokumentiert. Die Abbildungen sind jeweils Screenshots aus der laufenden GUI. Für die Bedeutung der einzelnen Farben, Kreise, Linien etc. siehe Abschnitt A.2 (Seite 85).

6.2.1. Karte a

Karte a ist ein kleines einfaches Gebäude mit einer Fläche von 40x50 Quadratmeter. Zielpunkt der Karte ist der Ausgang oben rechts.

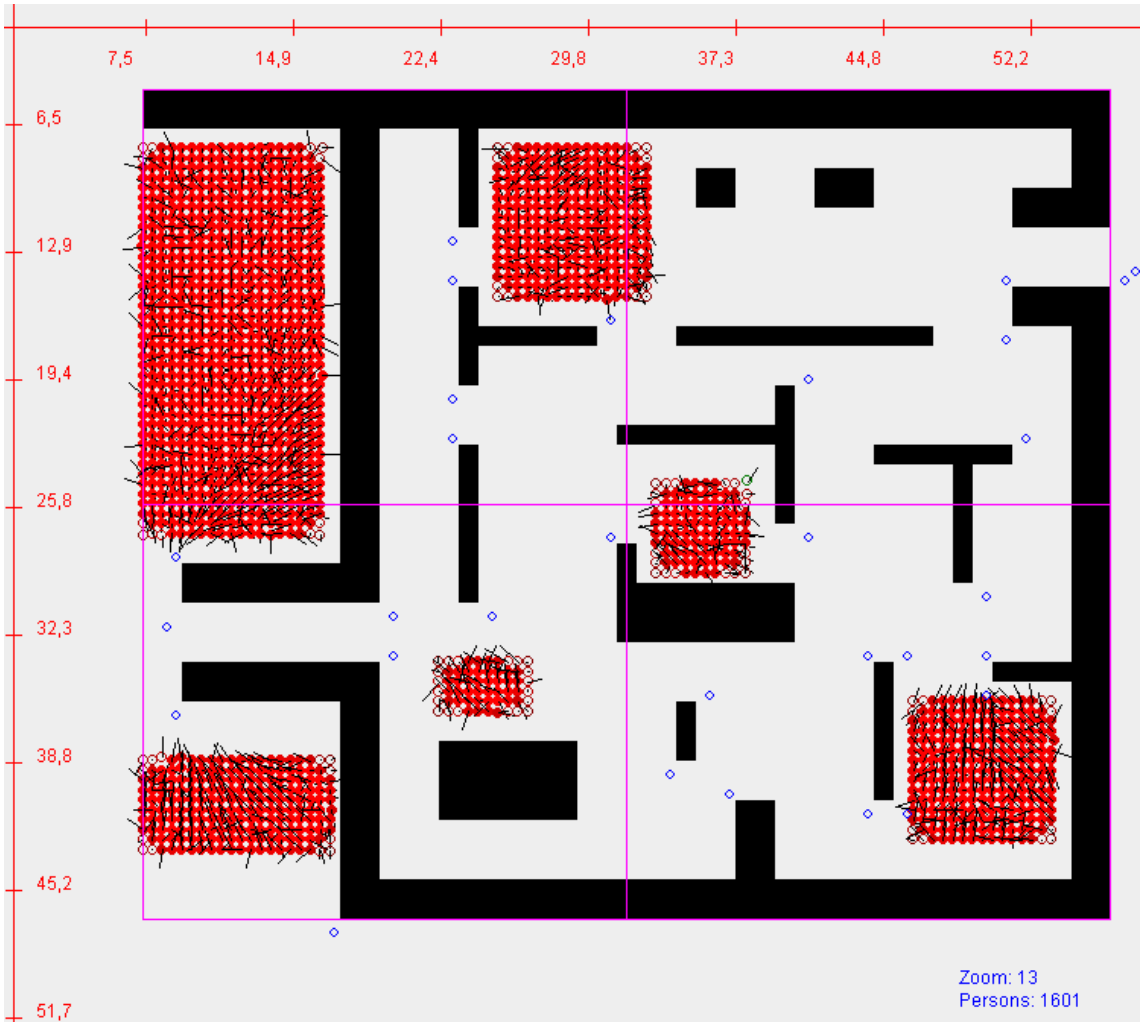


Abbildung 6.1.: Karte a

6.2.2. Karte b

Karte b ist ein mittleres Gebäude. Es besteht im Wesentlichen aus dem kleinen Gebäude von Karte a, welches auf ein grösseres Gebäude ergänzt wurde. Diese Karte wurde in zwei Hauptszenarien verwendet: 5088 Fussgänger (Abstand zwischen Personen: 1 Meter) und 14282 Fussgänger (Abstand zwischen Personen: 0.5 Meter). In Abbildung 6.2 (Seite 56) ist der Zielpunkt mit einem roten Pfeil markiert. Weiter wurde dieselbe Karte für Messungen verwendet, in denen laufend der Abstand der Fussgänger verkleinert (und damit die Fussgängeranzahl vergrössert) wurde, um eine Kurve der Laufzeit in Abhängigkeit zur Fussgängeranzahl zu erhalten. Die Karte hat eine Fläche von 100x100 Meter.

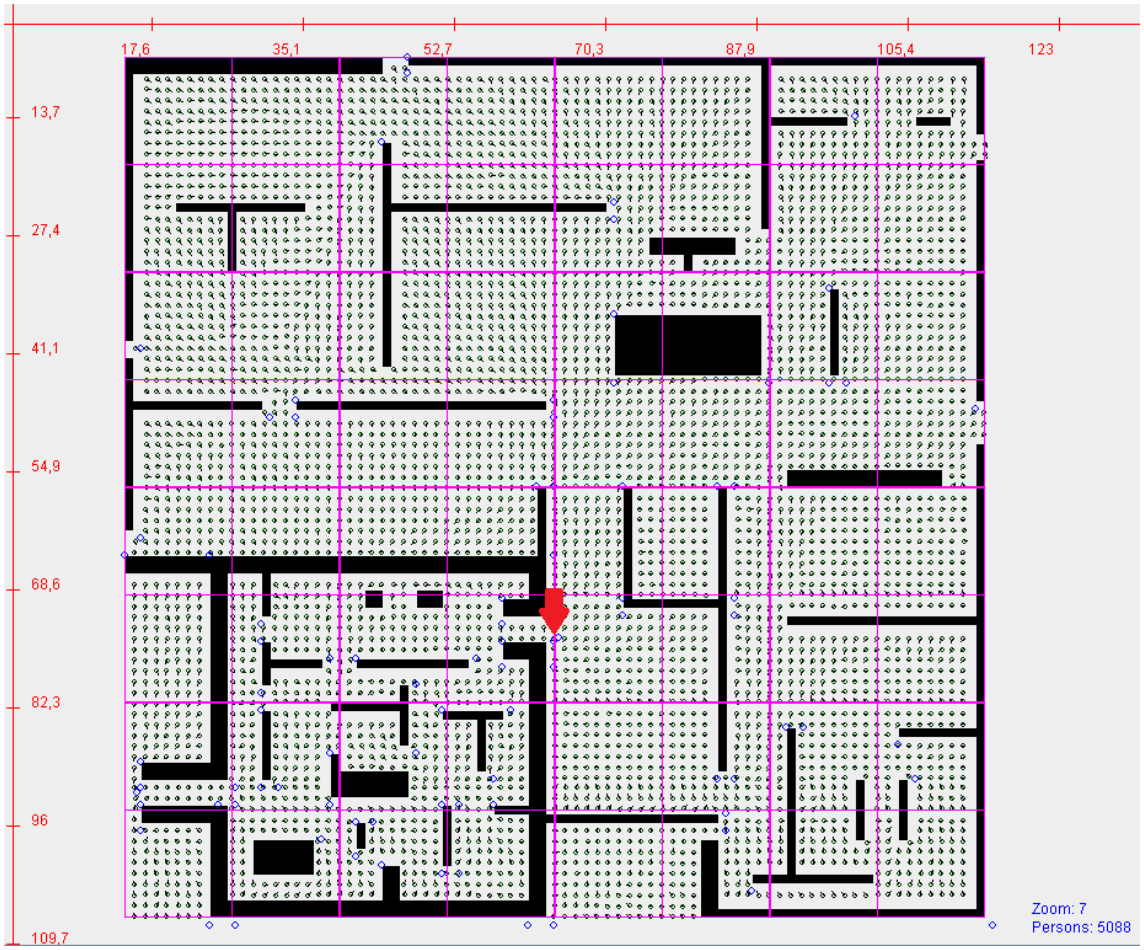


Abbildung 6.2.: Karte b, 5088 Fussgänger

6.2.3. Karte c

Karte c ist eine Karte mit nur wenigen und kleinen Hindernissen. Sie wurde verwendet um die Laufzeit in Abhängigkeit zur Fussgängerdichte zu messen. Dabei wurden 1936 Fussgänger mit jeweils unterschiedlichem Initialabstand erzeugt, wobei die Fussgängeranzahl konstant gehalten wurde. Abbildung 6.3 (Seite 57) zeigt eine dichtstehende Gruppe Fussgänger. Weiter wurde die Karte verwendet, um Messungen mit bis zu 40000 Fussgängern zu erheben. Die Karte hat eine Fläche von 250x250 Quadratmeter. Der Zielpunkt liegt in der Mitte des unteren linken Quadranten.

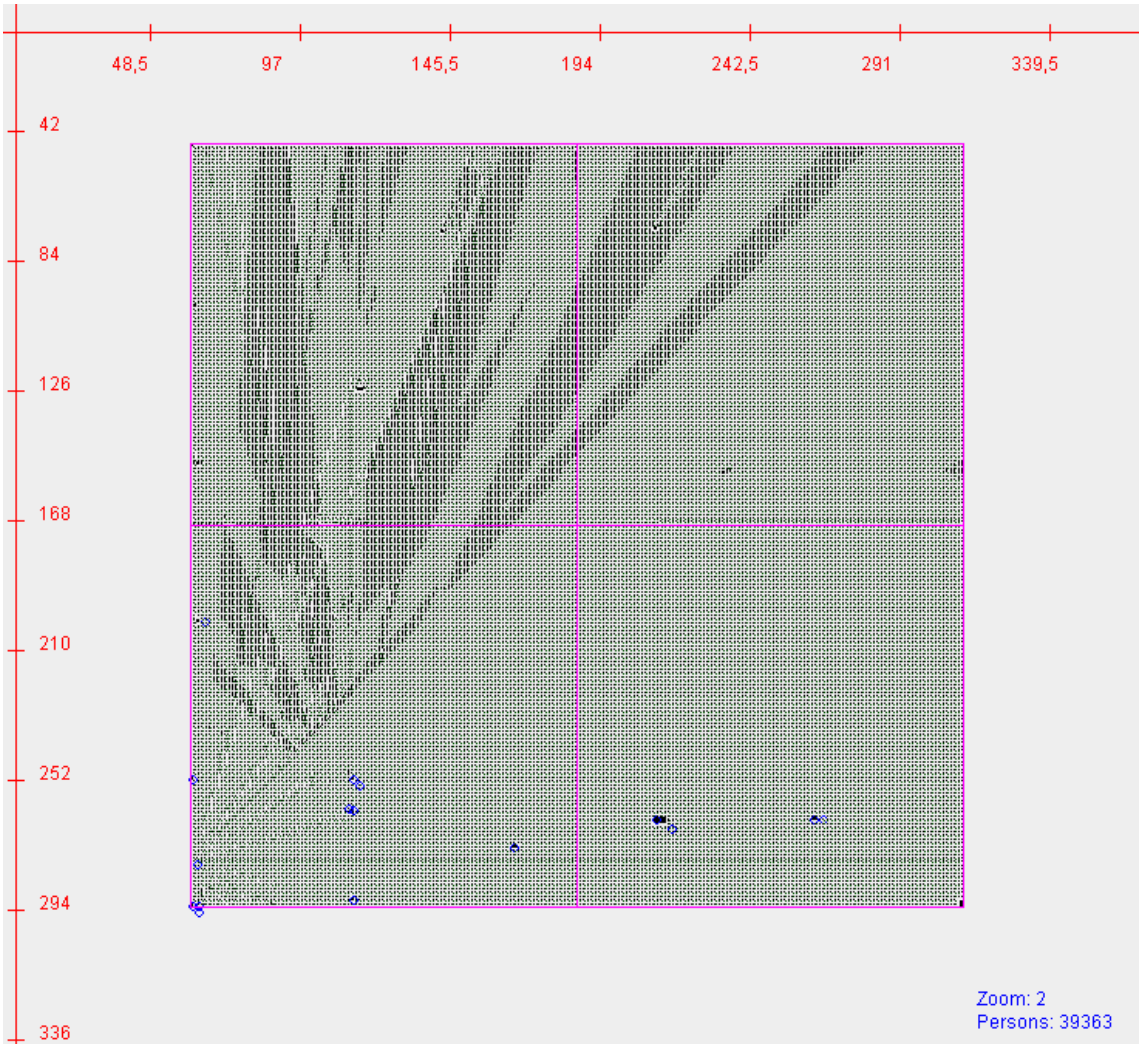


Abbildung 6.3.: Karte c, 39363 Fussgänger

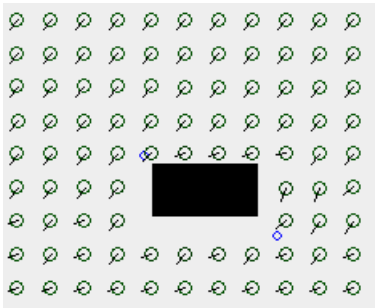


Abbildung 6.4.: Karte c, Ausschnitt aus Abbildung 6.3 mit Zoom-Faktor 16.7

6.2.4. Karte d

Karte d ist eine grosse Karte mit Mauern an den vier Rändern. Die Karte bietet Platz für mehr als eine Million Menschen. Die Karte hat eine Fläche von 1000x1000 Quadratmeter. Die Karte besitzt vier Zielpunkte. Fussgänger aus einem Quadranten streben auf den Mittelpunkt des in Uhrzeigersinn

nächsten Quadranten. Die Karte wurde verwendet um die Laufzeit mit vielen Fussgängern (325000) zu messen.

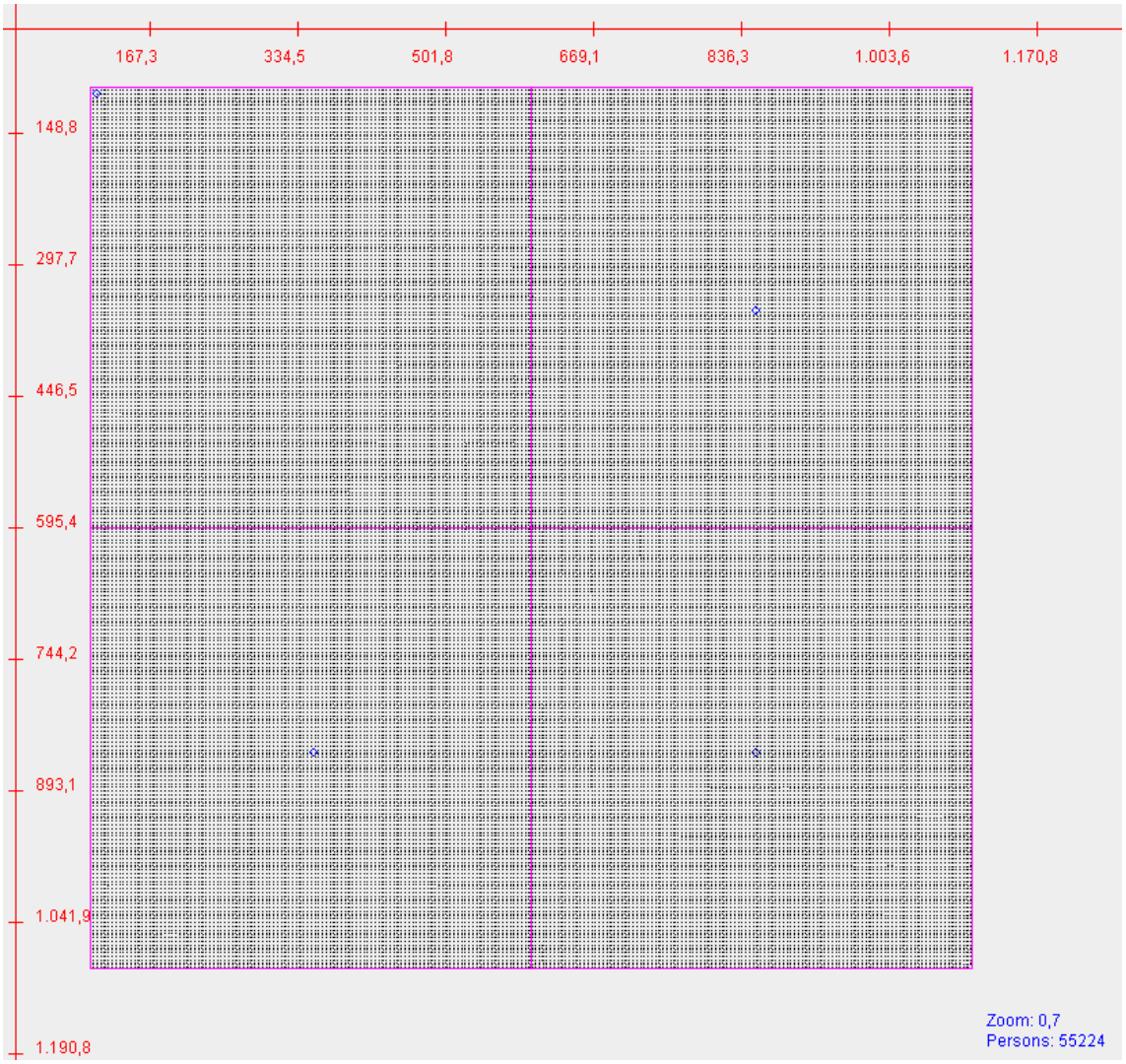


Abbildung 6.5.: Karte d, 55242 Fussgänger

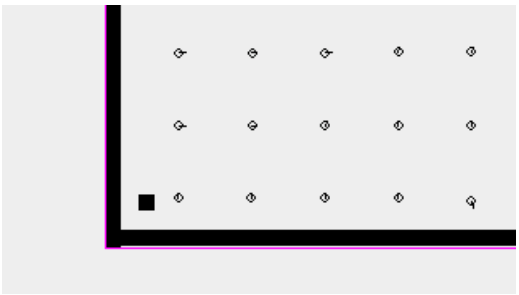


Abbildung 6.6.: Karte d, Ausschnitt aus Abbildung 6.5 mit Zoom-Faktor 10.7, linker unterer Ecken

6.2.5. Karte e

Bei Karte e handelt es sich um eine grosse Anlage mit der Fläche von 420x290 Quadratmetern. Im südlichen Teil befindet sich ein Gebäude der Grösse von ungefähr 210x200 Quadratmetern, das viele Mauern, Zimmer, Gänge und Kreuzungen enthält. Im oberen Teil ist die Fläche leer mit der Ausnahme eines 10x10 Quadratmeter grossen Objekts. Umzäunt wird die ganze Anlage von einer Mauer. Die Karte wurde in einem Szenario verwendet. Dieses soll eine Evakuation simulieren. Bei der Initialisierung der Simulation werden alle Fussgänger mit gleichen Abständen zueinander im Gebäude platziert. Sobald die Simulation gestartet wird, bewegen sich alle Fussgänger in Richtung des grossen Objekts im Norden. Im Extremfall kann eine Simulation mit 116474 Fussgängern berechnet werden, allerdings beträgt dann der Initialabstand zwischen den Fussgängern exklusive Körperumfang nur noch 0.3 Meter.

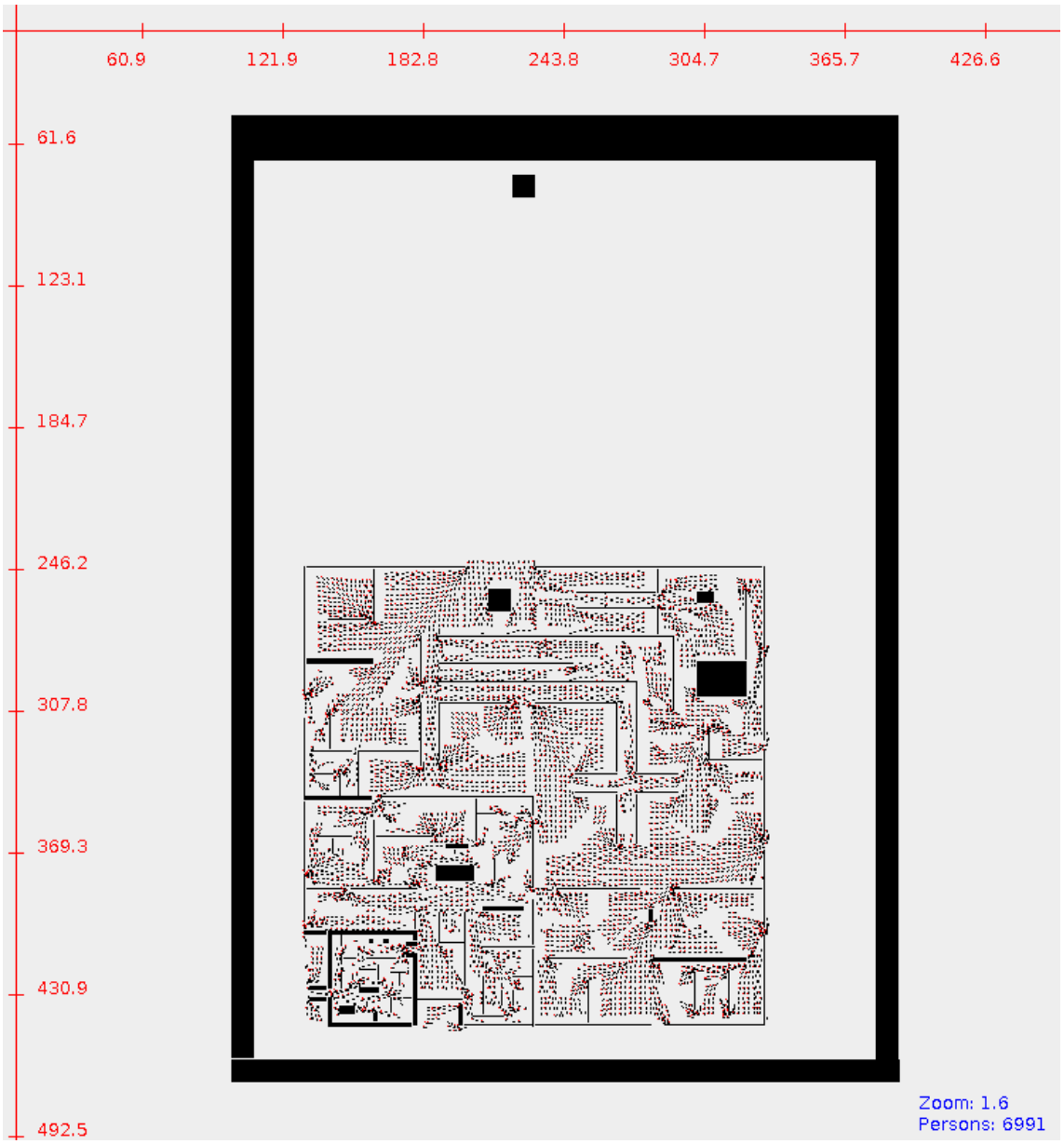


Abbildung 6.7.: Karte e, 6991 Fussgänger

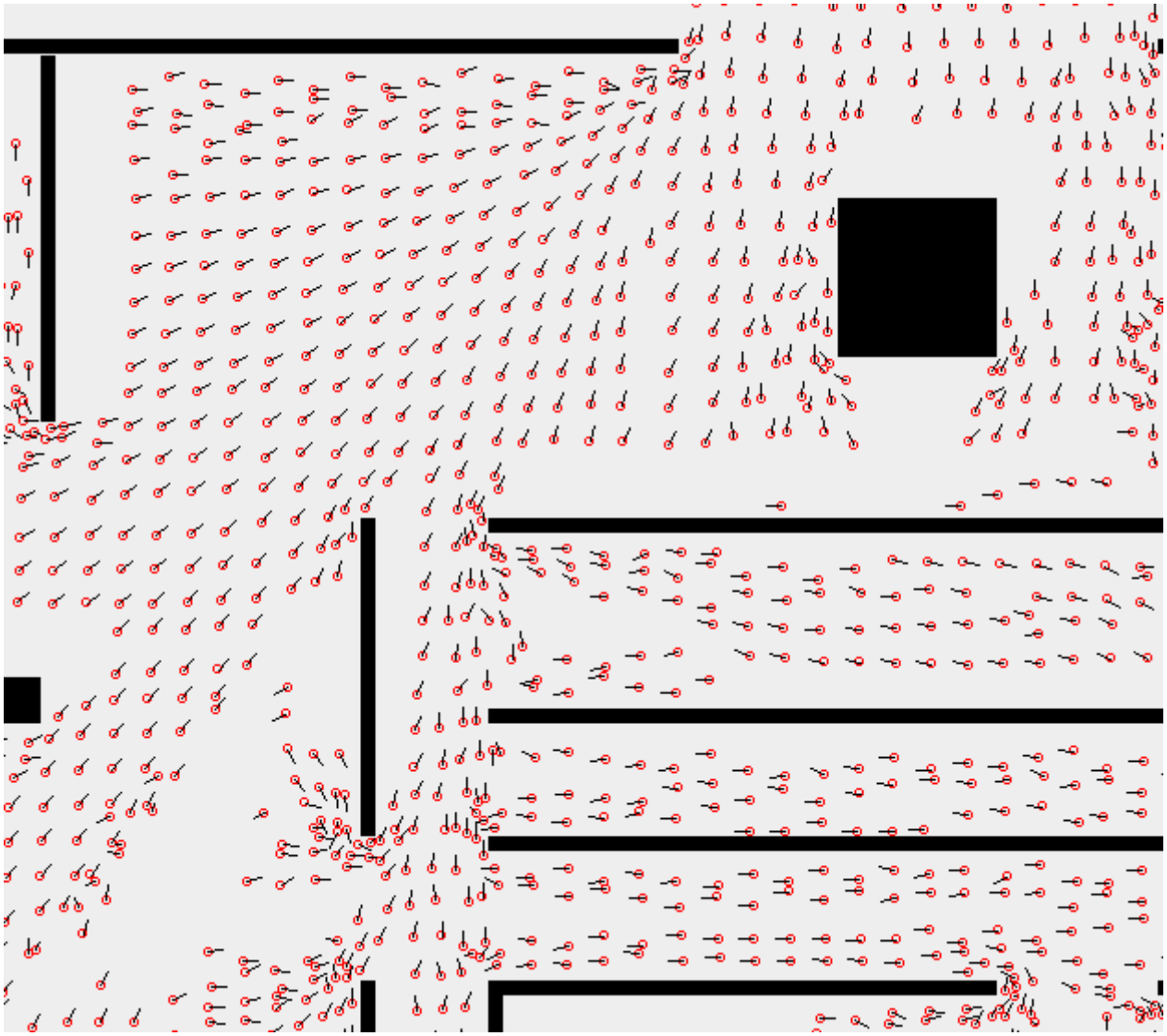


Abbildung 6.8.: Karte e, Ausschnitt aus Abbildung 6.7 mit Zoom-Faktor 8.6

6.3. Verwendete Simulationsparameter

Simuliert wurde mit einem Zeitschritt von 0.1 Sekunden, womit 1000 Simulationsschritte 100 Sekunden in Echtzeit sind. Alle anderen Simulationsparameter wurden wie folgt gesetzt:

- Standardsichtfeld: 180°
- Winkelauflösung (δ): 18°
- Standardsichtweite (S): 10 Meter
- Standardkörperradius (B): 0.25 Meter
- Standardwunschgeschwindigkeit: 1 m/s
- Relaxation Time τ: 0.4 Sekunden

Die Werte wurden ähnlich gewählt, wie sie auch in [7] in Simulationen verwendet wurden. Anstatt 1.3 m/s für die Standardwunschgeschwindigkeit wurde nur 1 m/s verwendet und die Relaxation Time wurde auf 0.4 Sekunden statt 0.5 Sekunden gesetzt. Anstatt dem niedrig gewählten 90° oder 150° Sichtfeld, wurden 180° verwendet. Der Körperradius r_i wurde anhand der Formel $r_i = \frac{m_i}{320}$ [7, S. 2] berechnet, wobei von einer Masse m_i einer 80 kg schweren Person ausgegangen wurde.

Die Standardwunschgeschwindigkeit entspricht der Geschwindigkeit mit der ein Fussgänger laufen möchte. Sie entspricht somit auch gerade der maximalen Geschwindigkeit, die ein Fussgänger haben kann. Sichtfeld, Sichtweite, Körperradius und Wunschgeschwindigkeit sind pro Fussgänger einstellbar. Bei allen Simulationen wurden jedoch für sämtliche Fussgänger die angegebenen Standardwerte verwendet. Der Zeitschritt kann erhöht werden, um dieselbe Zeit in Echtzeit in weniger Simulationsschritten und somit insgesamt schneller zu rechnen. Allerdings führt ein höherer Zeitschritt zu einer höheren Ungenauigkeit. Bei hohem Zeitschritt muss zudem die Mindestsichtweite der operativen Ebene beachtet und gegebenenfalles erhöht werden, da sonst Kollisionen nicht mehr zuverlässig erkannt werden können.

6.4. Hinweise zu den Messungen

Bei den Messungen handelt es sich jeweils um Einzelmessungen, das heisst, sie sind keine Mittelwerte mehrerer Messungen. Aus zeitlichen Gründen war es leider nicht möglich, jede Messung mehrere Male durchzuführen, um dann über mehrere Messungen zu mitteln. Insbesondere das Einrichten des Rechnerverbunds benötigte jeweils bereits viel Aufwand, der aus technischen Gründen jeweils pro Tag an dem gemessen wurde, wieder erneut betrieben werden musste. Beim i5 und i7 handelt es sich um die Arbeitslaptops, die auch zum Entwickeln der Software benötigt wurden und konnten deshalb nicht für viele oder lange Messungen verwendet werden, da ansonsten diese Geräte nicht für Arbeiten an der Software zur Verfügung standen. Einzelne Messungen wurden wiederholt, falls das gemessene Resultat nicht mit den Erwartungen übereinstimmte, um sicher zu gehen, dass das gemessene Resultat gültig ist und nicht durch unbekannte Umstände verfälscht wurde.

Einige Teile der Kurven in den Messdiagrammen wurden automatisch interpoliert und stellen keinen effektiv erfassten Messwert dar. Die erfassten Messwerte sind in den Kurven auf den Diagrammen mit Symbolen speziell gekennzeichnet. Ausnahme davon ist Abbildung 6.10 (Seite 63). Um die Übersichtlichkeit aufgrund der vielen Messwerte zu wahren, wurde in dieser Abbildung auf die Symbole verzichtet.

Die Auslastung der Systeme wurde bei Windows-Systemen mittels des Windows Task Managers überprüft. Unter Linux wurde dafür die Applikation *htop* verwendet.

6.5. Phase 1 und Phase 2

In Phase 1 wurde der Schwerpunkt auf einen funktionierenden Prototypen gelegt. Anschliessend wurden Messungen mit jenem Prototypen getätigt, um erste Ergebnisse bezüglich Skalierbarkeit, Auslastung und Performanz zu erhalten. Ausgehend von diesen ersten Ergebnissen wurde nach Optimierungspotenzial sowohl im Ansatz als auch im Code gesucht. In Phase 2 wurden diese Optimierungen implementiert. In Phase 1 wurde ohne Subzonen-Vorfilterung und ohne Optimierungen des Codes gemessen, während in Phase 2 dann mit der Subzonen-Vorfilterung und den Code-Optimierungen gemessen wurde.

6.6. Messergebnisse (Phase 1)

6.6.1. Laufzeit in Abhängigkeit der Dichte

Messungen haben ergeben, dass aufgrund der adaptiven Sichtweite und des Algorithmuswechsels dicht stehende Fussgänger effizienter gerechnet werden können als nicht so dicht stehende Fussgänger. Bei

einem durchschnittlichen Personenabstand von einem Meter ist die Simulation am langsamsten. Ideal ist ein durchschnittlicher Personenabstand d für $((d \geq 4) \vee (d \leq 0.25))$ Meter. Gemessen wurde auf Karte c mit 1936 Fussgängern, die jeweils unterschiedlich dicht stehend erzeugt wurden und dann alle auf denselben Zielpunkt zustrebten. Abbildung 6.9 (Seite 62) zeigt die Laufzeiten über 1000 Simulationsschritte mit unterschiedlichen Dichten.

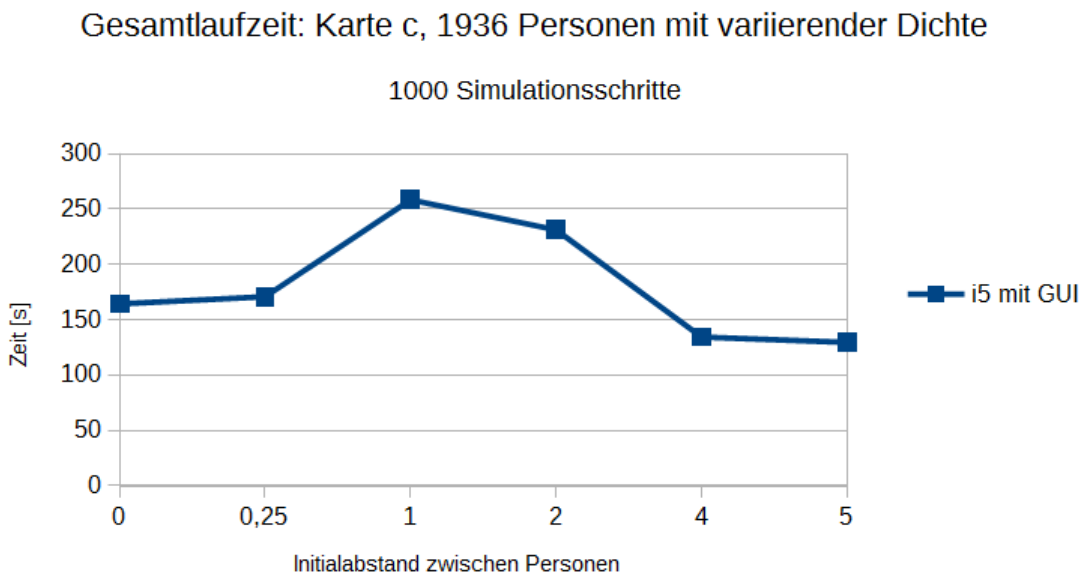


Abbildung 6.9.: Laufzeit in Abhängigkeit der Dichte

6.6.2. Laufzeit in Abhängigkeit der Fussgängeranzahl

Einzelrechner

Die Messungen der Laufzeit in Abhängigkeit der Fussgängeranzahl haben gezeigt, dass der Aufwand mittels Vorfilterung durch Sichtweite und aufgrund der Aufteilung in Zonen nicht so stark wächst wie zuerst angenommen. Die Messwerte sind in Abbildung 6.10 (Seite 63) ersichtlich. Ebenso haben die Messungen gezeigt, dass weniger Fussgänger auf dem i5 effizienter berechnet werden können als auf dem i7, wobei der i7 bei mehreren Fussgängern klar bessere Laufzeiten liefert.

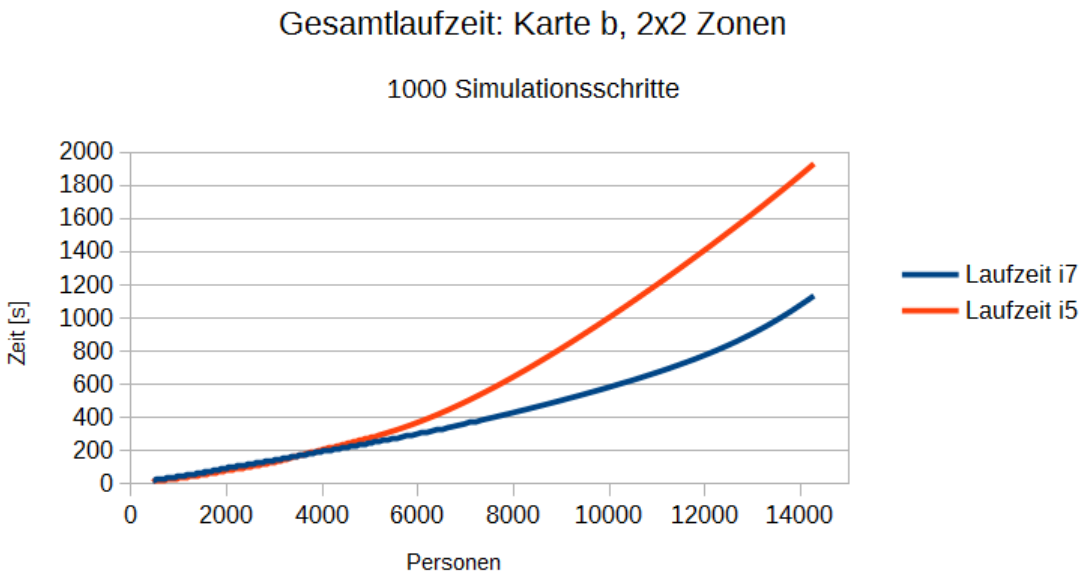


Abbildung 6.10.: Laufzeit in Abhängigkeit der Fussgängeranzahl

Rechnerverbund

Um eine Abschätzung machen zu können, wie viele Rechner man benötigt, um eine Simulation in einer bestimmten Zeit auszuführen, wurden weitere Messungen in Abhängigkeit der Fussgänger- und Rechneranzahl getätigt. Die Messungen erfolgten über 250 Simulationsschritte. Der Rechnerverbund bestand aus mehreren PCs vom Typ DSK. Die Messungen zeigen, dass man auf gegebener Karte viele Fussgänger (mehr als 10000) nicht in Echtzeit rechnen kann, selbst wenn man sehr viele Rechner zur Verfügung hat. Weiter zeigen die Messungen, dass man mit n-mal so vielen Rechnern nicht n-mal schneller simulieren kann. Beispielsweise bei 39363 Fussgängern erreicht man mit 5-mal so vielen Rechnern nur eine Verbesserung der Laufzeit um Faktor 3.5. Dies Begründet sich durch den erhöhten Datenaustauschaufwand übers Netzwerk. Es hat sich gezeigt, dass die CPUs der Rechner in einem Rechnerverbund kaum eine Auslastung von höher als 30% erreicht haben. Die Rechner können deshalb nicht die ganze Rechenleistung effizient ausnutzen. Abbildung 6.11 (Seite 64) stellt die Messungen im Rechnerverbund grafisch dar.

Mehrere Simulationen gleichzeitig

Da die Auslastung der Rechner eher klein ist, bot es sich an zu testen, ob man auf denselben Rechnern mehrere Simulationen gleichzeitig rechnen lassen kann. Die Ergebnisse zeigen, dass man zwei Simulationen gleichzeitig auf denselben Rechnern schneller rechnen kann als hintereinander. So benötigten zwei Simulationen von 39363 Fussgängern auf Karte c mit 16 Zonen pro Rechner 221 Sekunden (wobei eine Simulation der beiden Simulationen nur 184 Sekunden benötigte). Die beiden Simlulationen hintereinander - also sequentiell - gerechnet benötigten 304 Sekunden. Somit ist man um Faktor 1.37 schneller, wenn zwei Simulationen gleichzeitig gerechnet werden.

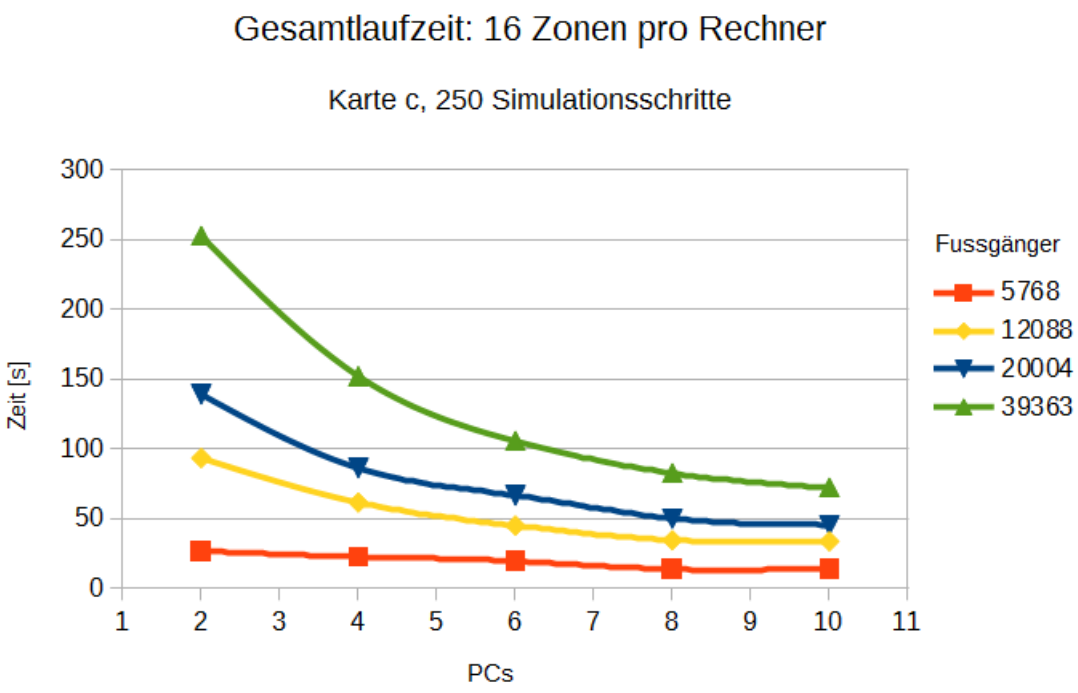


Abbildung 6.11.: Laufzeit in Abhängigkeit der Fussgängeranzahl

6.6.3. Laufzeit in Abhängigkeit der Zonenanzahl

Einzelrechner

Auf einzelnen Rechnern hat es sich gezeigt, dass eine möglichst hohe Zonenanzahl bei vielen Fussgängern die besten Ergebnisse liefern. Bei geringerer Fussgängeranzahl kann es jedoch vorkommen, dass eine Erhöhung der Zonenanzahl zu einer längeren Gesamtlaufzeit führt und somit kontraproduktiv ist. Die maximal mögliche Zonenanzahl ist gegeben durch die Grösse einer Karte. Denn eine Zone darf nicht kleiner als die maximale Sichtweite der Fussgänger sein, da ansonsten ein Fussgänger am Rand der Zone bis in die übernächste Zone sehen könnte.
Abbildung 6.12 (Seite 65) zeigt die Gesamtlaufzeit einer Simulation auf Karte b mit 14282 Personen mit steigender Zonenanzahl. Auf dem i7 kann bei einer Erhöhung der Zonenanzahl von 2x2 auf 9x9 die Gesamtlaufzeit um Faktor 2.4 gesenkt werden. Abbildung 6.12 zeigt weiter einen Vergleich mit der Gesamtlaufzeit gerechnet auf dem i7 und dem i5 (über Netzwerk) sowohl mit als auch ohne laufende GUI. Abbildung 6.13 (Seite 65) zeigt den Effekt, dass eine Erhöhung der Zonenanzahl bei geringerer Fussgängeranzahl eine Verlängerung der Gesamtlaufzeit zur Folge haben kann.

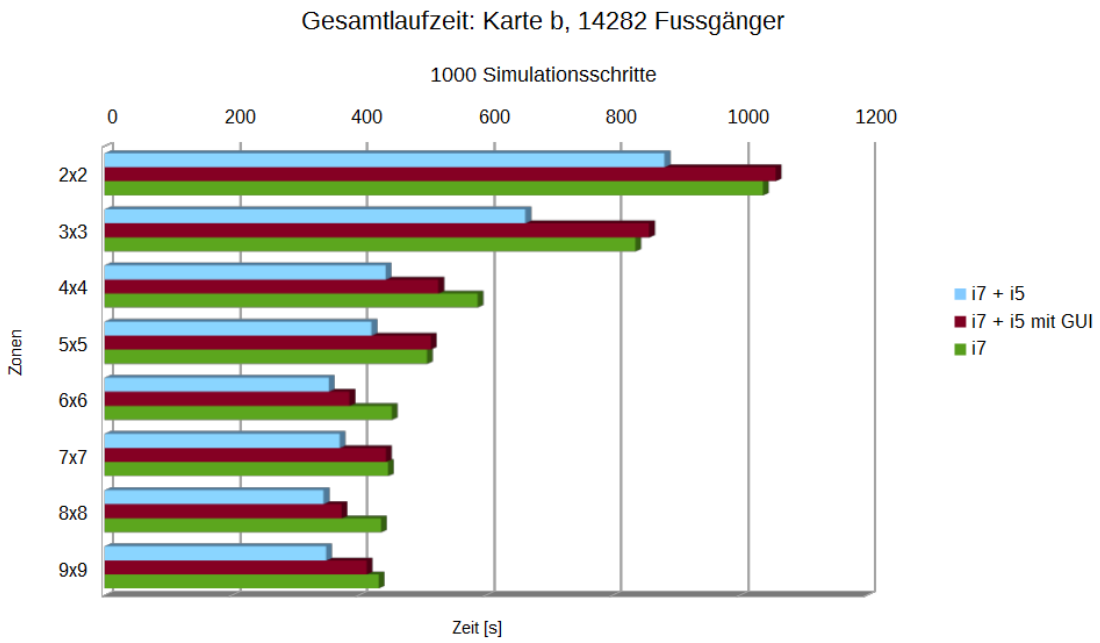


Abbildung 6.12.: Laufzeit in Abhängigkeit der Zonenanzahl, Karte b, 14282 Fussgänger

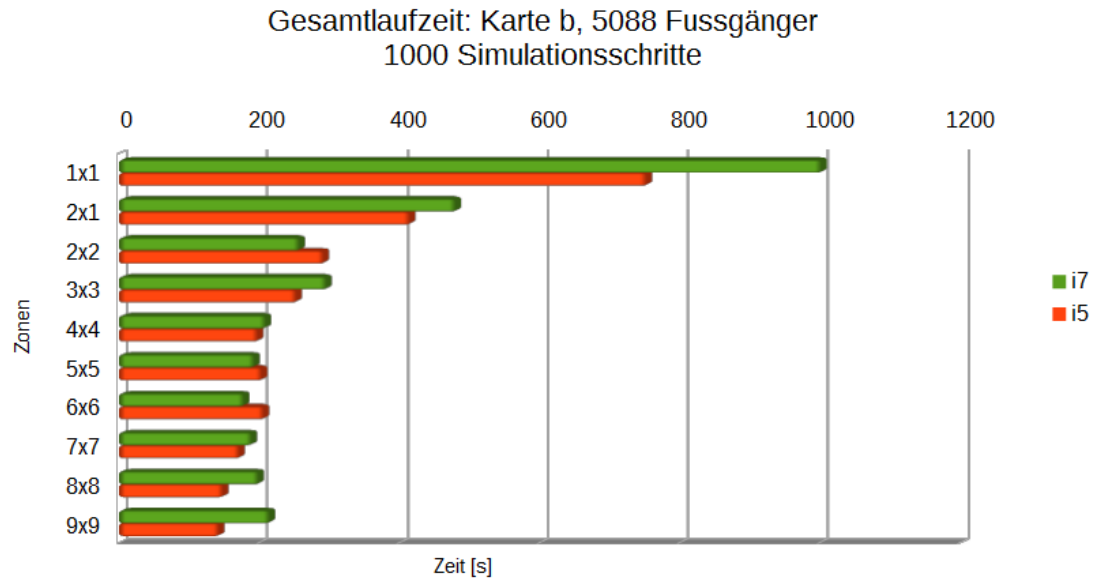


Abbildung 6.13.: Laufzeit in Abhängigkeit der Zonenanzahl, Karte b, 5088 Fussgänger

Rechnerverbund

In einem Rechnerverbund mit mehreren Rechnern ist eine möglichst hohe Zonenanzahl kontraproduktiv. Im Rechnerverbund mit Rechnern vom Typ DSK hat es sich gezeigt, dass 16 Zonen pro Rechner optimal sind. Abbildung 6.14 (Seite 66) stellt die Resultate dieser Messung grafisch dar, dabei wurde mit 20004 Fussgängern auf Karte c gerechnet.

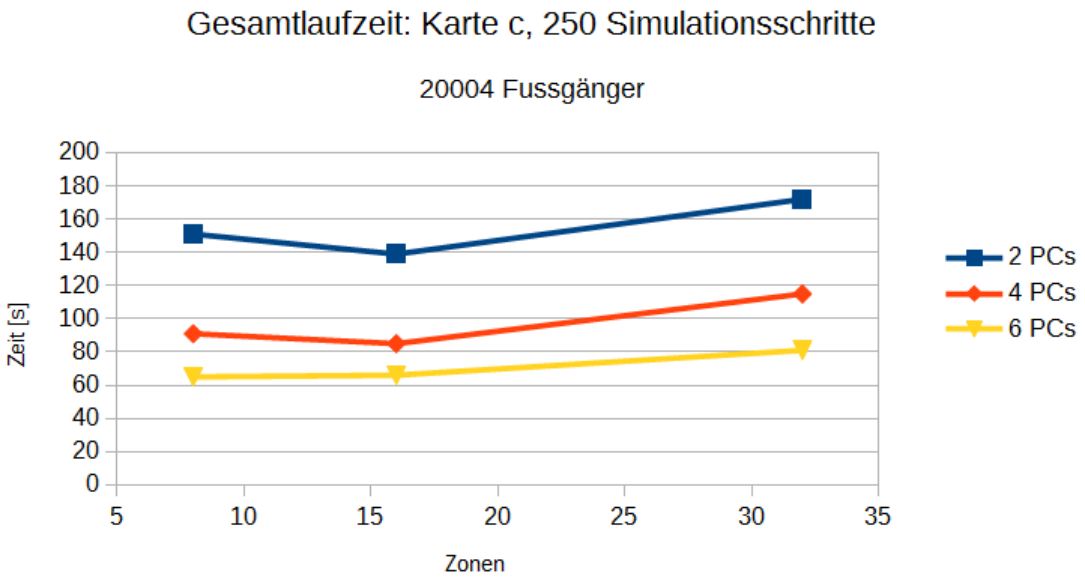


Abbildung 6.14.: Laufzeit in Abhängigkeit der Zonenanzahl, optimale Zonenanzahl für Rechnerverbund

6.6.4. Einfluss der GUI auf die Gesamtlaufzeit

Die GUI ermöglicht es, das Verhalten der Fussgänger grafisch zu verfolgen und ist insbesondere zur Fehlersuche ein nützliches Instrument. Mit aktivierter GUI ist die Gesamtlaufzeit einer Simulation jedoch höher als ohne GUI. Es hat sich gezeigt, dass die GUI zwar zu längeren Gesamtlaufzeiten führt, jedoch keinen grossen Einfluss auf die Skalierung hat, wenn verteilt gerechnet wird. Abbildung 6.16 (Seite 67) stellt Messungen auf der Karte a mit 1601 Fussgängern jeweils mit und ohne GUI grafisch dar.

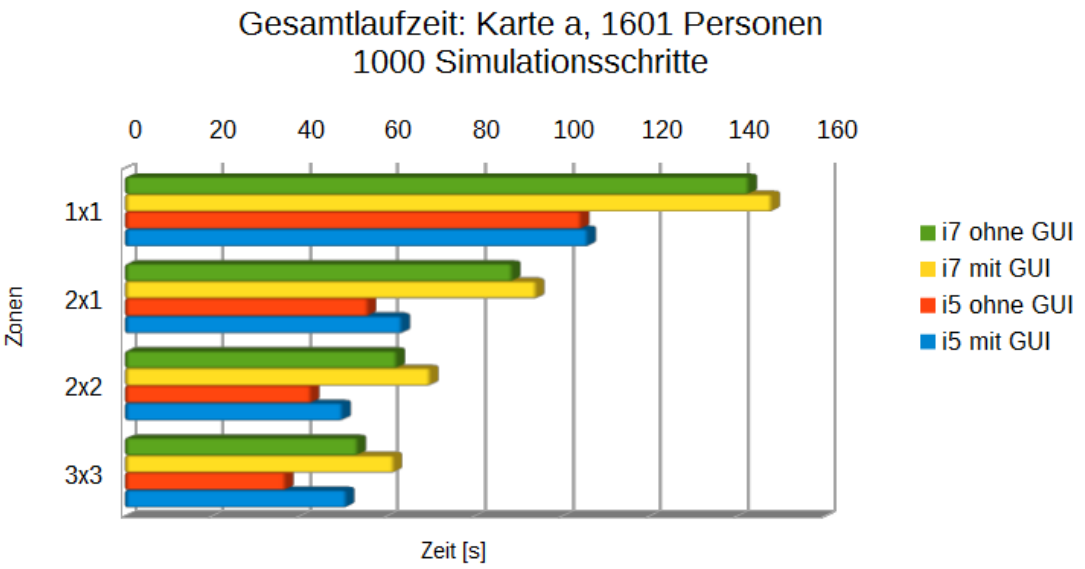


Abbildung 6.15.: Laufzeiten mit und ohne GUI

6.6.5. Overhead der Simulation

Um festzustellen, wie viel Zeit für den Datenaustausch benötigt wird oder Zonen-Aktoren darauf warten, bis sie den nächsten Schritt berechnen dürfen, wurde nebst der Gesamtlaufzeit in einer Messung zusätzlich noch der Overhead erfasst. Es hat sich gezeigt, dass der Overhead mit der Anzahl Rechner steigt. Obwohl zwar die Gesamtlaufzeit in der Messung bei einer Erhöhung auf 10 Rechner gesenkt wurde, stieg gleichzeitig der prozentuale Anteil des Overheads. Im Falle von 10 Rechnern rechneten in $\frac{1}{5}$ der Zeit Zonen-Aktoren nicht, sondern mussten auf Daten oder auf ihre Ausführung warten.

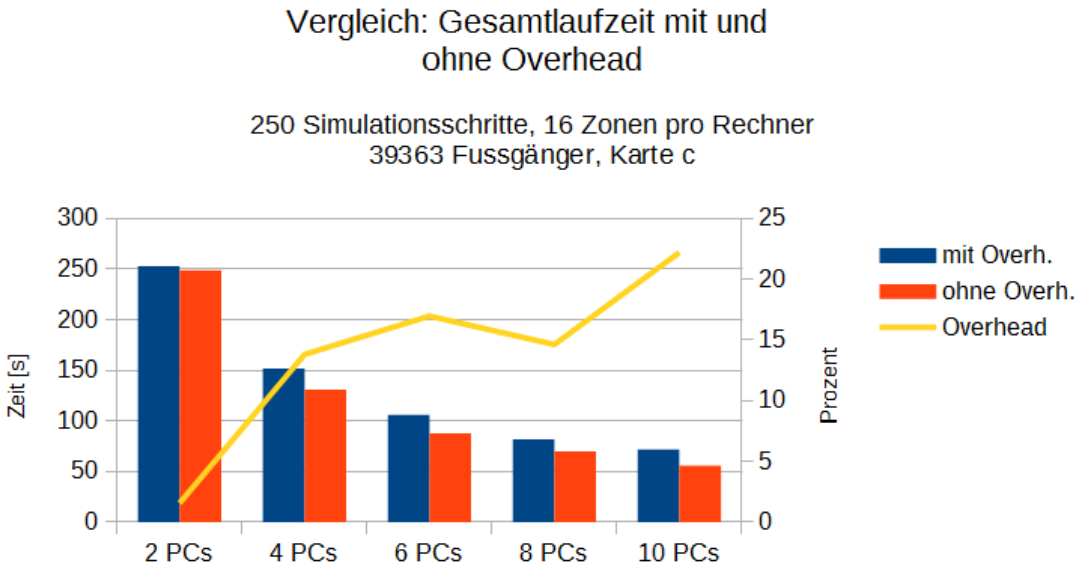


Abbildung 6.16.: Anteil des Overheads in Abhängigkeit der Anzahl Rechner

6.6.6. Weitere Messungen

Nebst den oben erwähnten Messungen wurden noch einzelne sporadische Messungen durchgeführt, die an dieser Stelle aufgelistet sind:

Karte	Zonen	Fussgänger	Rechner	Simulationsschritte	GUI	Gesamtlaufzeit (in Sekunden)
d	12x12	197133	i7 + 8x DSK	1000	Nein	1658
d	20x20	324896	i7 + 20x DSK	1000	Nein	1093
c	20x20	39427	1x DSK	1000	Nein	1577
c	10x10	39427	i7 + 4x DSK	1000	Ja	831

Die Veränderung der Laufzeit mit unterschiedlichen Zeitschritten Δt wurden ebenso gemessen. Diese Messungen fanden auf Karte a mit 1601 Personen statt:

Zeitschritt (in Sekunden)	Gesamtlaufzeit (in Sekunden)
0.1	132
0.3	36
0.5	28

6.7. Messergebnisse (Phase 2)

Alle Messungen in Phase 2 wurden auf Rechnern des Typs DSK durchgeführt.

6.7.1. Vergleich mit unoptimierter Version aus Phase 1

Ein Vergleich mit der unoptimierten Version zeigt deutlich, wie viel die getätigten Optimierungen ausmachen. Für den Vergleich wurde erneut bestimmt, welche Zonenanzahl die besten Ergebnisse liefert. Anschliessend wurden neue Messungen auf Karte c ausgeführt. Abbildung 6.17 (Seite 68) zeigt die Gesamtlaufzeit mit unterschiedlicher Anzahl Zonen. Um sicher zu gehen, dass vier Zonen pro Rechner optimal sind, und nicht etwa konstant $4 \cdot 4 = 16$ Zonen, wurden die weiteren Messungen jeweils einmal mit vier Zonen pro Rechner und einmal mit 16 Zonen im gesamten Rechnerverbund durchgeführt. Die Resultate dieser Messungen sind in Abbildung 6.18 (Seite 69) zu sehen. Das Kürzel (siehe Legende Abbildung 6.18) *opt. const* bedeutet, dass die entsprechende Messung mit konstant $4 \cdot 4$ Zonen durchgeführt wurde. Das Kürzel *opt. 4* bedeutet, dass die Messung mit vier Zonen pro Rechner durchgeführt wurde. Ist kein Kürzel angegeben, so handelt es sich um eine alte Messung mit der unoptimierten Version. Die Auslastung der einzelnen Computer ist in Phase 2 deutlich besser als in Phase 1. Diese lag meistens zwischen 60% – 80%.

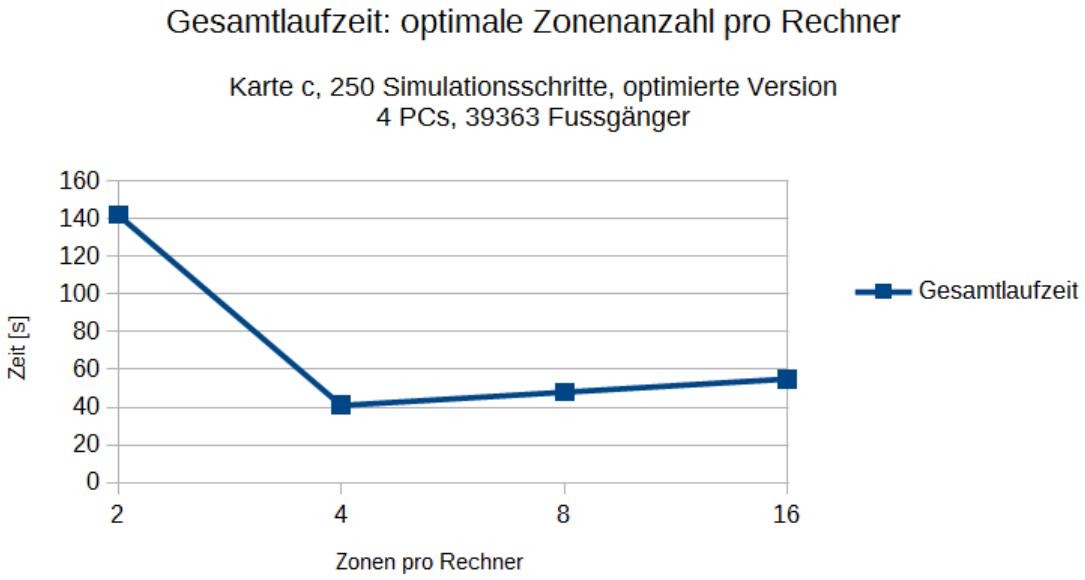


Abbildung 6.17.: Optimale Zonenanzahl auf Karte c

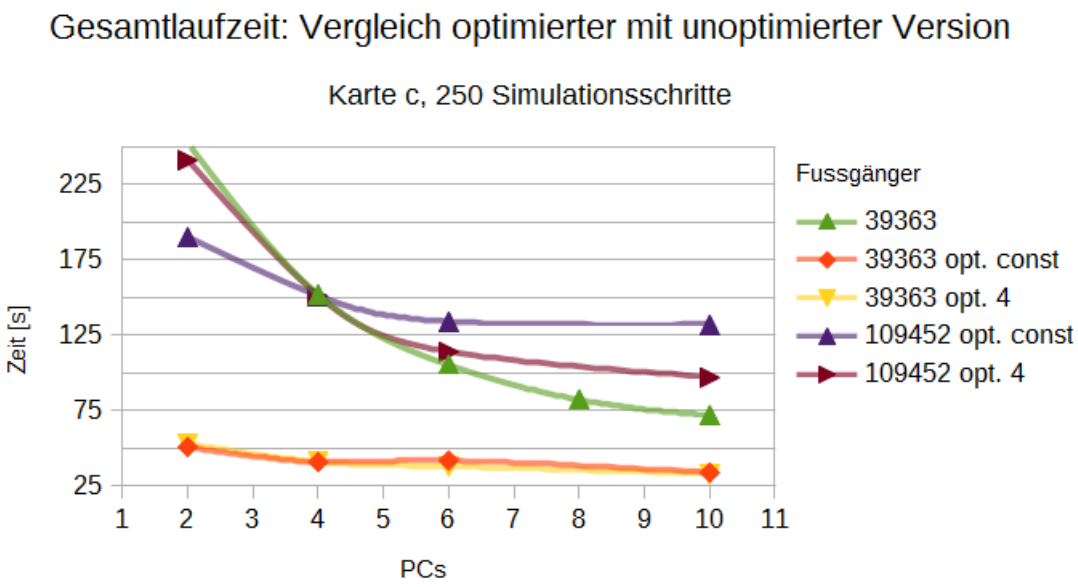


Abbildung 6.18.: Vergleich unoptimierte mit optimierter Version auf Karte c

6.7.2. Messungen mit optimierter Version mit hoher Fussgängeranzahl

Mit der optimierten Version war es nun möglich, Messungen mit sehr vielen Fussgängern durchzuführen. Diese Messungen fanden auf Karte d statt. Zuerst wurde wieder die optimale Zonenanzahl für Karte d ermittelt. Diese Werte sind in Abbildung 6.19 (Seite 70) abgebildet. Es stellte sich heraus, dass auf Karte d die optimale Zonenanzahl acht Zonen pro Rechner ist. Es wurden Messungen von ca. 130'000 Fussgängern bis zu ca. 1 Mio. Fussgängern durchgeführt. Eine Million Fussgänger lässt sich auf Karte d mit der optimierten Version in ca. 12-mal Echtzeit simulieren. Alle Ergebnisse zu den Messungen auf Karte d sind in Abbildung 6.20 (Seite 71) ersichtlich.

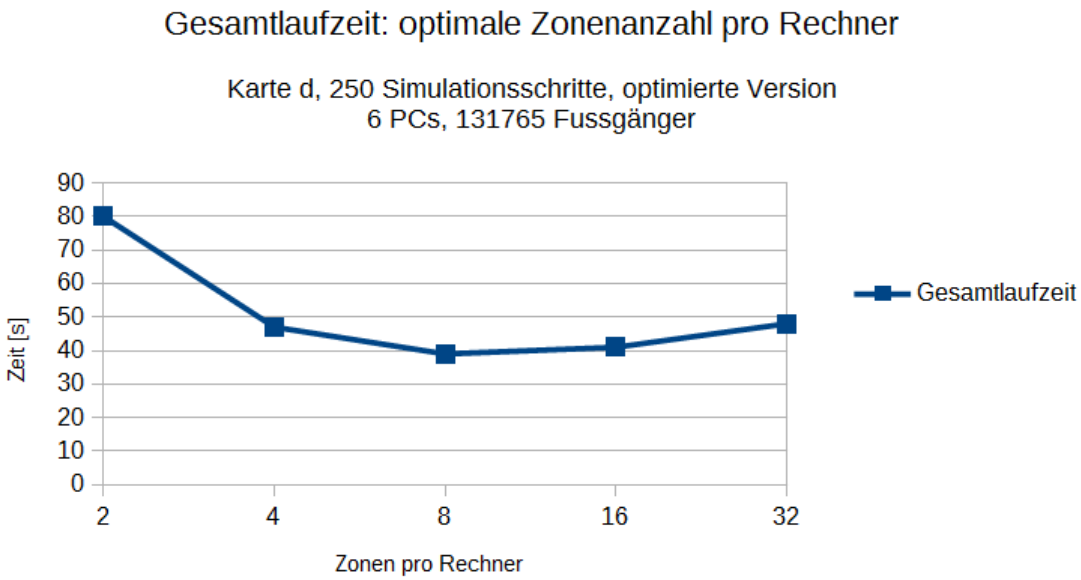


Abbildung 6.19.: Optimale Zonenanzahl auf Karte d

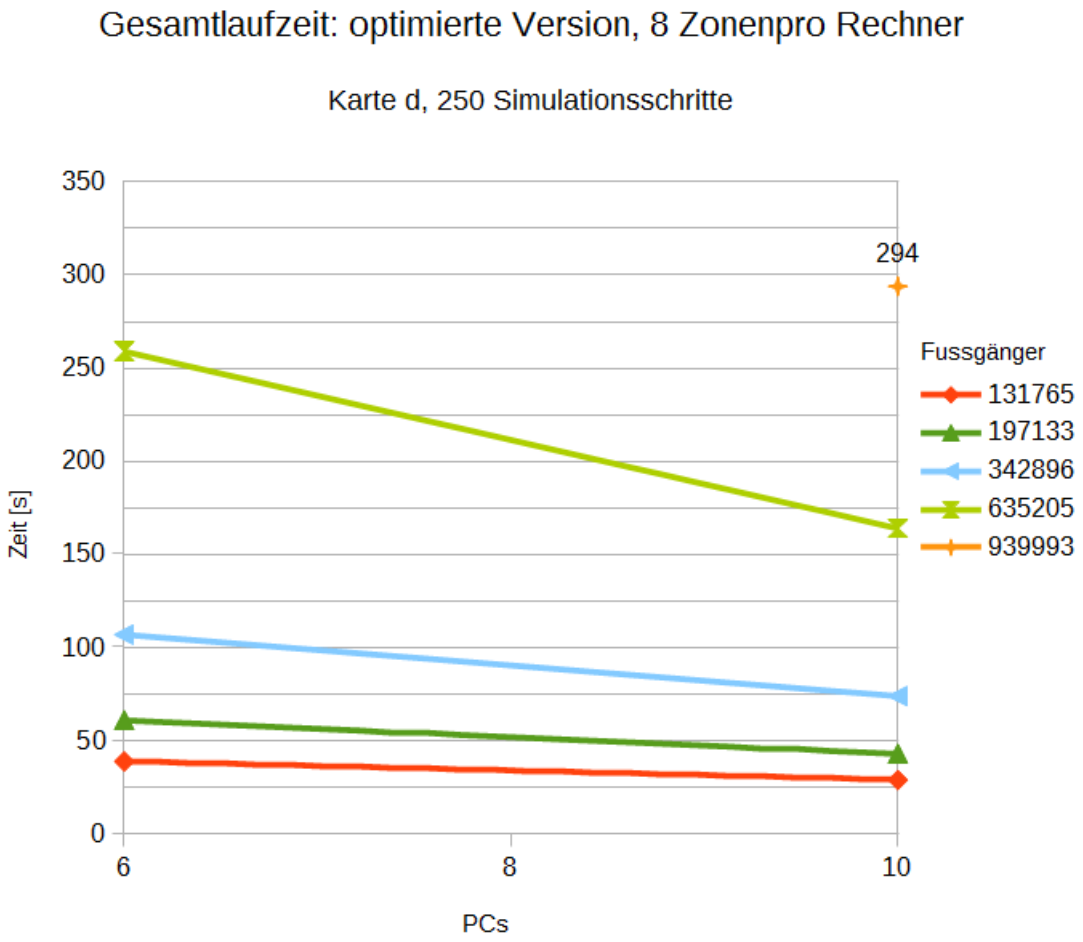


Abbildung 6.20.: Messungen mit optimierter Version auf Karte d

6.7.3. Messung mit optimierter Version mit hoher Gebäudekomplexität

Mit vielen Fussgängern wurde bisher auf Karten gerechnet, die nur wenig Mauern enthielten. Durch Karte e gibt es die Möglichkeit, viele Fussgänger in einer Anlage zu simulieren, die einem grossen Gebäude ähnelt. Bei den Ergebnissen, die in Abbildung 6.21 (Seite 72) grafisch festgehalten sind, zeigt sich ein etwas anderes Verhalten als bei Karten mit wenigen statischen Hindernissen.
Auch mit wenigen Fussgängern (6991) werden bereits 52 Sekunden benötigt, um 25 Sekunden zu simulieren. Die erhöhte Laufzeit ist während der Simulation besonders während der Simulation des ersten Schrittes bemerkbar. Dieser dauert auf Karte e erheblich länger als beispielsweise auf Karte d, da für jeden Fussgänger im ersten Schritt eine Routenberechnung durchgeführt werden muss. Dies dauert auf Karte e länger, da der Weggraph viel grösser ist. Weil bei dieser Messung nur 25 Sekunden simuliert wurden, erhält der erste Simulationsschritt ein relativ grosses Gewicht. Würde eine Messung über eine längere Zeitdauer erfolgen, würde die Dauer des ersten Simulationsschrittes weniger Anteil an der Gesamtlaufzeit erhalten.
Die Simulation wurde auf 10 Rechnern durchgeführt mit einer Zonenanzahl von 8 Zonen pro Rechner.

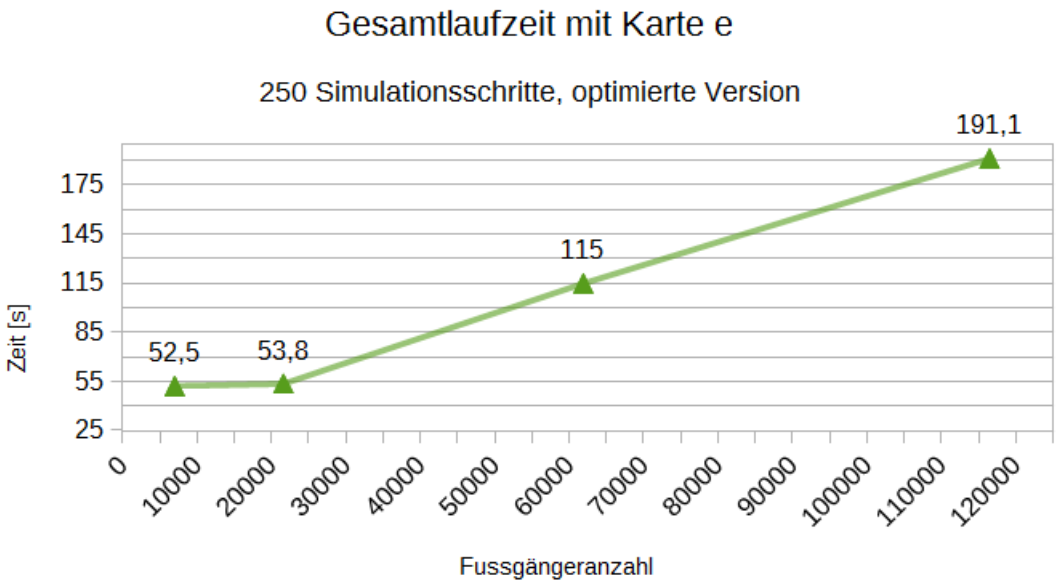


Abbildung 6.21.: Messung der Laufzeit auf Karte e mit steigender Fussgängeranzahl

7. Diskussion

7.1. Messergebnisse

Die Messergebnisse zeigen, dass kein linearer Speedup erreicht wurde. Es wurde jedoch erwartet, dass der Speedup nicht linear ist. Dies kann einerseits durch das Gesetz von Amdahl begründet werden. Ein gewisser Anteil der Applikation wird sequentiell abgearbeitet und muss auch sequentiell erfolgen. Andererseits liegt es auch daran, dass in jedem Schritt ein Datenaustausch und somit Kommunikationsoverhead in nahezu jedem Schritt anfällt. Es wurde ebenfalls erwartet, dass der Kommunikationsoverhead, da jede Zone mit jeder Nachbarzone kommunizieren muss, steigt, je mehr Zonen vorhanden sind. Dies ist z.B. in Abbildung 6.17 (Seite 68) gut ersichtlich. Eine Erhöhung der Zonenanzahl von 2 auf 4 brachte einen vergleichsweise grossen Speedup, eine weitere Erhöhung der Zonenanzahl führte jedoch wieder zu einer längeren Gesamtlaufzeit. Ebenso wurde erwartet, dass man nicht beliebig viele Rechner in einem Rechnerverbund zusammenfassen kann, um beliebig schnell simulieren zu können. Abbildung 6.18 (Seite 69) verifiziert diese Vermutung. In der Abbildung ist gut ersichtlich, dass der Speedup bei einer bestimmten Rechneranzahl abflacht und mehr Rechner kaum zu kürzeren Gesamtlaufzeiten einer Simulation führt. Um die optimale Zonenanzahl und Rechneranzahl zu finden, sollte eine gewünschte Simulation mit einer geringen Anzahl Simulationsschritte mit verschiedener Anzahl Zonen und Anzahl Rechner durchgeführt werden (bspw. 100 Simulationsschritte). Anhand dieser Ergebnisse können dann die optimalen Parameter bestimmt werden. Es ist leider nicht gelungen, eine Abschätzung als Formel zu finden, um diese Parameter näherungsweise zu berechnen. Dies, da es zu viele Variablen wie Karte, Anzahl Hindernisse, Anzahl Fussgänger, Geometrie, Dichte der Fussgänger, Zielpunkte der Fussgänger, Verteilung der Fussgänger, Verteilung der Hindernisse und so weiter gibt. Es muss daher an bestehenden Messungen auf ähnlichen Karten orientiert werden.

Die Messungen zeigen, dass es durchaus machbar ist, in einem Rechnerverbund mit ungefähr 10 Rechnern eine Million Fussgänger in vernünftiger Zeit zu simulieren. Weiter zeigen sie, dass es sich lohnt - möchte man eine einzelne Simulation möglichst schnell rechnen - die Simulation in einem Rechnerverbund mit mehreren Rechnern zu rechnen. Eine Erhöhung der Rechneranzahl von bspw. 2 auf 10 kann durchaus die Gesamtlaufzeit halbieren (siehe z.B. Kurve mit 109452 Fussgängern in Abbildung 6.18 auf Seite 69). Der nicht lineare Speedup bedeutet jedoch auch, dass, möchte man mehrere Simulation insgesamt so schnell wie möglich rechnen, sich die Verteilung auf mehrere Rechner nicht lohnt. Möchte man beispielsweise fünf Simulationen rechnen, wobei eine auf zwei Rechnern $100s$ benötigt, so würden diese fünf Simulationen hintereinander auf 10 Rechnern (Annahme: Gesamtlaufzeit halbiert sich) $5 * (100/2) = 5 * 50 = 250s$ dauern. Auf 10 Rechnern können jedoch nebeneinander fünf Simulationen auf je zwei Rechnern gerechnet und damit alle fünf Simulationen in insgesamt $100s$ simuliert werden.

7.2. Realismus des Modells

Das implementierte Modell bietet nur einen geringen Grad an Realismus. Ohne eingebaute zufällige Richtungsänderung in bestimmten Situationen verklumpen sich entgegengesetzte Fussgängerströme und es kommt zu einem 'Deadlock' in dem sich keine Person mehr bewegen möchte. Dem wurde in dieser Arbeit entgegengewirkt, indem in solchen Situationen zufälliges Gedrängel entsteht. Weitere unrealistische Situationen kann man bei engen Durchgängen beobachten. Möchten z.B. zwei Fussgänger zur selben Zeit durch eine enge Türe, behindern sie sich gegenseitig und können sich nicht einigen, wer nun den Vortritt haben soll. Man merkt deutlich, dass den Fussgängern eine gewisse Intelligenz fehlt, um mit solchen Situationen umgehen zu können. Eine andere Einbusse an Realismus findet bei abgedrängten Fussgängern statt (siehe Abschnitt 5.4.2 auf Seite 42). In solchen Fällen wird schlicht der erste sichtbare

Wegpunkt im Weggraph genommen. Dieser erste sichtbare Wegpunkt ist nicht optimal und kann sogar in entgegengesetzter Richtung liegen. Es werden daher teilweise grosse Umwege eingeschlagen.

7.3. Akka und das Aktorenmodell

Durch Akka lässt sich eine Applikation gut auf einem relativ hohen Abstraktionsniveau entwickeln. Dabei muss man sich nicht um Details kümmern, wann wie viele Threads gestartet werden müssen, wie die Arbeit auf diese Verteilt werden, wie die Kommunikation zwischen ihnen stattfindet etc. Die grundlegenden Probleme der parallelen Entwicklung bleiben aber natürlich bestehen. Die ganze Kommunikation zwischen Aktoren wird grundsätzlich asynchron implementiert. Dies erfordert beim Entwickeln der Applikation ein Umdenken, da in konventionellen Programmiersprachen meist mit synchronen Methodenaufrufen gearbeitet wird. Durch die asynchrone Kommunikation steigt auch die Gefahr von Race Conditions, da viele verschiedene Messages jeweils in unterschiedlicher Reihenfolge einem Aktor zugestellt werden können.
Es gibt aber auch negative Seiten am Akka-Framework. Durch die grosse Flexibilität, die das Framework bietet, ist es möglich sehr viele Konfigurationsparameter zu definieren. Diese sind auch dokumentiert, jedoch so zahlreich, dass man schnell den Überblick verlieren kann. Des Weiteren ist das Debugging einer Akka-Applikation eher schwierig. Da sich Aktoren defaultmässig bei einem Fehler neu starten, können Fehler eventuell übersehen werden. Wenn Messages den Empfänger nicht erreichen, wird dies durch eine Fehlermeldung *dead-letters* angezeigt. Dabei ist es aber schwierig herauszufinden, was der Inhalt der Message war und aus welchem Grund die Message nicht ordnungsgemäss zugestellt werden konnte. Dies ist insbesondere bei Remote-Aktoren problematisch, da hier noch Fehler bei der Übertragung im Netzwerk auftreten können.

8. Ausblick

8.1. Parallelisierung auf anderen Systemen

In dieser Arbeit wurde untersucht, wie gut sich die Simulation auf mehrere Rechner verteilen lässt. Der Prototyp ist daher nicht auf den Einzelrechnerfall mit Shared Memory optimiert. Es müsste untersucht werden wie gut der Zonenansatz optimiert auf Shared-Memory-Systemen skaliert. Eine Portierung der operativen Ebene auf GPUs müsste ebenso untersucht werden, sowohl lokal als auch auf mehrere Rechner verteilt. Die Aktoren könnten als Kommunikations- und Synchronisationsmittel beibehalten und die operative Ebene auf den GPUs gerechnet werden.

8.2. Effizientere Algorithmen für die operative Ebene

Der alternative Algorithmus, der bei kritischer Dichte zum Einsatz kommt, ist schneller als der Algorithmus basierend auf der Strahlabtastung. Der alternative Algorithmus kann jedoch keine Mauern erkennen. Er müsste entsprechend erweitert werden, um solche statische Hindernisse zu erkennen. Anschliessend wäre ein Vergleich mit der Strahlabtastung möglich, um zu bestimmen, ob der alternative Algorithmus auch mit Hinderniserkennung noch schneller ist.

Weiter könnte man nach noch effizienteren Algorithmen für die operative Ebene suchen.

8.3. Erweiterung der taktischen Ebene

8.3.1. Stauumgehung

Die taktische Ebene besitzt momentan noch nicht die Fähigkeit der Stauumgehung. Fussgänger verfolgen jeweils die kürzeste Route, wobei die Auslastung der Route (z.B. Stau durch Fussgänger) nicht gewichtet wird. Die taktische Ebene sollte erweitert werden, sodass Fussgänger ihre Route ändern, falls es mehrere Wege zu einem Ziel gibt und einer der Wege durch Stau zeitlich nicht mehr der schnellste Weg ist. Dies würde erfordern, dass Zonenaktoren Daten über die Auslastung von Wegen (Kanten im Weggraph) periodisch untereinander austauschen.

8.3.2. Wegweiser

Das Modell könnte um Wegweiser (Wegschilder) für die taktische Ebene erweitert werden. Dadurch wäre es möglich zu simulieren, wie sich Ortsunkenntliche verhalten und ob sie mit den Wegweisern den Weg finden können, oder ob weitere Wegweiser angebracht werden müssten. Ebenso könnte man simulieren, dass Fussgänger mit einer Wahrscheinlichkeit p einen Wegweiser übersehen. Findet ein Fussgänger den Weg nicht mehr nachdem er einen Wegweiser übersehen hat, müssten möglicherweise weitere Wegweiser angebracht werden. Unter Umständen wäre es möglich, mit genügend Wegweisern auf einen Weggraph zu verzichten und wäre dadurch nicht mehr auf die – je nach Karte – sehr rechenintensiven Routenberechnungen angewiesen.

8.3.3. Optimierung des Weggraphen

Der Weggraph ist im Prototypen, der im Zuge dieser Arbeit entstand, nicht optimal. Er enthält nur unidirektionale Kanten und hat alleine deswegen mehr Kanten als unter Umständen nötig wären (Es sei denn es gibt Wege, die tatsächlich nur in eine Richtung begangen werden dürfen). Ausserdem überlagern sich viele Kanten. Einige dieser sich überlagernden Kanten könnten wegoptimiert werden.

8.3.4. Bessere Handhabung des Abdrängungsproblem

Das Abdrängungsproblem (siehe Abschnitt 5.4.2 auf Seite 42) muss besser gelöst werden. Es darf nicht einfach der erste sichtbare Wegpunkt gewählt werden, stattdessen sollte der beste sichtbare Wegpunkt verwendet werden. Dies würde jedoch erfordern, den Weggraph ganz zu durchsuchen, was sehr ineffizient sein dürfte. Hier muss nach einer effizienten Lösung gesucht werden.

8.4. Verteilung der Zonen auf Workers

Der Algorithmus zur Verteilung der Zonen auf Workers kann noch optimiert werden. Die Zuteilung könnte während der Laufzeit gewechselt werden, sobald festgestellt wird, dass ein Worker sehr viel mehr Rechenaufwand bewältigen muss. Eine einfache Möglichkeit wäre, dass die Zonen nach jedem Simulationsschritt die Zeit mitschicken, die für die Berechnung für den Simulationsschritt benötigt wurde. Der Master kann nun die Zeit eines Schrittes pro Worker summieren und vergleichen. Sobald sich der Rechenaufwand eines Workers deutlich unterscheidet, wird die Simulation unterbrochen und eine rechenintensive Zone einem anderen Worker zugewiesen. Danach kann die Simulation fortgesetzt werden.

8.5. Zonenmodell

Das Zonenmodell könnte erweitert werden, so dass es möglich ist, verschieden grosse Zonen zu verwalten. Ebenso könnten möglicherweise bei geringer Rechenauslastung in Zonen, diese Zonen zu einer grossen Zone zusammengefasst werden.

8.6. Realisierung mit Akka

Einige Optimierungsmöglichkeiten bestehen auch im Umgang mit Akka. Einige sind nachfolgend dokumentiert.

8.6.1. Futures

Aktoren sollten grundsätzlich nicht blockierend implementiert werden. Dies bedeutet, dass die Abarbeitung einer Nachricht nicht zu lange dauern sollte. Der Grund dafür ist, dass der Aktor während der Verarbeitung einer Nachricht nicht mehr ansprechbar ist. Akka verfügt über ein Algorithmus, der überprüft, welche Aktoren noch erreichbar sind. Antworten Aktoren für eine bestimmte Zeit nicht, werden sie als unerreichbar gekennzeichnet. Nachrichten, die an solche Aktoren geschickt werden sollen, werden nicht zugestellt.
Um dies zu verhindern, können innerhalb von Aktoren sogenannte Futures eingesetzt werden. Futures erlauben die nebenläufige Durchführung eines Tasks. Ein Future ist dabei eine Datenstruktur, welche es erlaubt, auf das Ergebnis eines solchen Tasks zuzugreifen. Dieser Zugriff kann asynchron oder synchron erfolgen. In Aktoren können Futures verwendet werden, um das Resultat eines Futures an einen Aktor zu schicken. Mit Hilfe dieses Konstruktes können Berechnungen innerhalb von Aktoren parallel

und nicht blockierend durchgeführt werden. Resultate dieser Berechnungen können als Resultat eines Futures einem anderen Aktor zugeschickt werden.

8.6.2. TCP vs. UDP

Bei den Messergebnissen wurden die Messages jeweils mittels TCP-Protokoll ausgetauscht. Akka erlaubt die Konfiguration des Netzwerk-Transportprotokolls und somit könnte auch UDP verwendet werden. Dabei müsste man aber in der Applikation sicherstellen, dass es nicht zu Verlust von Nachrichten kommen kann. Dies könnte zum Beispiel durch erneutes Senden der Nachricht nach einer bestimmten Zeit erfolgen. Eine Umstellung auf UDP könnte zur Folge haben, dass die Übertragungsgeschwindigkeit steigt und somit die Zeit für den Datenaustausch sinkt. Dadurch würde die Applikation besser skalieren.

8.6.3. Serialisierung

Akka erlaubt verschiedene Implementationen der Serialisierung von Nachrichten. Die standardmässige Java-Serialisierung ist sehr einfach zu benutzen. Allerdings ist diese nicht auf Geschwindigkeit oder kleine Datenmengen optimiert. Es gibt andere Implementationen, welche diese Eigenschaften aufweisen und mit Akka verwendet werden können. Dazu muss allerdings zusätzlicher Aufwand betrieben werden, um die Nachrichten und ihre Attribute zu spezifizieren.
Ein Beispiel einer alternativen Serialisierung ist *protobuf*[1]. Durch die Verwendung einer effizienteren Serialisierung könnte der Kommunikationsaufwand durch kleinere Nachrichten und schnellere Serialisierung verkleinert werden, was wiederum die Skalierbarkeit der Applikation positiv beeinflussen würde.

8.6.4. Konfiguration

Akka erlaubt eine umfassende Konfiguration von fast jeder Komponente. In dieser Arbeit wurden aber meist die Standardeinstellungen verwendet. Es müsste ausprobiert werden, wie sich das System verhält, wenn gewisse Parameter angepasst würden. Zum Beispiel könnte die maximale Anzahl Threads, die erzeugt werden dürfen, angepasst werden. Ebenfalls könnte der Dispatcher geändert werden. Anstatt eines *Fork-Join-Executor* könnte der *Thread-Pool-Executor* verwendet werden. Weitere Parameter für Message-Queue-Grössen und verschiedene Timeouts, Intervalle und Beschränkungen wären ebenfalls zu prüfen.

[1] `https://code.google.com/p/protobuf/`

9. Verzeichnisse

Literaturverzeichnis

[1] J. Dadová. Cellular automata approach for crowd simulation. http://www.sccg.sk/~dadova/phd/rigorozka_dadova_final.pdf [Stand: 23.05.2014], 2012.

[2] D. Helbing. A fluid-dynamic model for the movement of pedestrians. https://www.complex-systems.com/pdf/06-5-1.pdf [Stand: 23.05.2014], 1992.

[3] D. Helbing and A. Johansson. Pedestrian, crowd and evacuation dynamics. http://www.ethlife.ethz.ch/archive_articles/100727_Massenpanik_Helbing_sch/Pedestrian_Crowd_and_Evacuation_Dynamics_Helbing.pdf [Stand: 02.06.2014], 2010.

[4] D. Helbing and P. Molnár. Social force model for pedestrian dynamics. http://www.cs.uu.nl/docs/vakken/mcrs/papers/10.pdf [Stand: 23.05.2014], 1995.

[5] S. P. Hoogendoorn, P. Bovy, and W. Daamen. Microscopic pedestrian wayfinding and dynamics modelling. In M. Schreckenberg and S. Sharma, editors, *Pedestrian and Evacuation Dynamics*, pages 123–155. Springer, 2002.

[6] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming*. Morgan Kaufmann Publishers, 2012.

[7] M. Moussaïd, D. Helbing, and G. Theraulaz. How simple rules determine pedestrian behavior and crowd disasters. http://www.pnas.org/content/early/2011/04/08/1016507108.full.pdf+html?with-ds=yes [Stand: 23.05.2014], 2011.

[8] A. Patel. Implementation notes from amit's thoughts on pathfinding. http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html [Stand: 02.06.2014].

[9] T. Rauber and G. Rünger. *Multicore: Parallele Programmierung*. Springer, 2008.

[10] A. Steiner. Modellierung und simulation von verkehrssystemen. Präsentationsunterlagen im Fach Transportsysteme 1.

[11] A. U. K. Wagoum. Route choice modelling and runtime optimisation for simulation of building evacuation. http://d-nb.info/1037086201/34 [Stand: 21.05.2014], 2012.

Abbildungsverzeichnis

Glossar

In diesem Abschnitt werden Abkürzungen und Begriffe kurz erklärt.

Abk	Abkürzung
API	Application Programming Interface. Eine Programmierschnittstelle, die es anderen Programmen ermöglicht eine angebotene Funktionalität zu benutzen.
Barrier	Eine Barriere, bei der Threads warten müssen, bis alle anderen Threads ebenfalls die Barriere erreicht haben. Eine Barrier ist somit ein wichtiges Instrument in der Synchronisation von parallelen Programmen.
Buffer	Eine Datenstruktur beziehungsweise Speicher, der mehrere Elemente enthalten kann.
Cloud	Ein erweiterbares Netzwerk von Computern dessen Rechenleistung, Speicher oder eine spezifische Applikation üblicherweise als Dienst für Kunden über das Internet angeboten werden.
Deseralisieren , Deserialisierung	Siehe Serialisieren.
Fork/Join	Ein Problem wird aufgeteilt (Fork) in mehrere Teile, die am Schluss wieder zusammengeführt (Join) werden. Für Threads bedeutet dies, dass eine Menge an Threads erzeugt wird, deren Teilresultate zu einem Schlussresultat zusammengeführt werden.
Heap	Eine spezielle Region im Speicher. Sie wird unter anderem von Java für für Objekte benutzt.
KTI	Kommission für Technologie und Innovation.
Laufzeitkomplexität	Grössenordnung eines Algorithmus oder Problemes. Die Laufzeitkomplexität gibt an, wieviele Schritte (asymptotisch) benötigt werden, um ein Problem der Grösse N zu lösen.
Master/Slave	Ein Pattern bei dem ein Master mehrere Slaves kontrolliert. Der Master stellt somit die Kontrollinstanz dar. Die Slaves folgen den Instruktionen des Masters.
Message Queue	Eine Queue ist eine Datenstruktur in der Informatik, die es ermöglicht, Elemente hinzuzufügen und in der selben Reihenfolge wieder zu entfernen. Eine Message Queue ist eine Queue, die für die Abarbeitung von Nachrichten verwendet wird.
Overhead	Zusatzdaten, die nicht zu den eigentlichen Nutzdaten zählen. Vielfach wird damit auch die Zeit bezeichnet, die dazu benötigt wird, diese Zusatzinformationen zu speichern oder zu verschicken.
Pattern	Vorgehensweise beziehungsweise Lösung für bekannte und häufige Probleme.
Performance	Performanz. Leistungsfähigkeit eines Programmes oder Rechners.
Polyline	Eine aus zusammenhängenden Liniensegmenten bestehende Linie wobei jeweils der Endpunkt eines Liniensegments der Startpunkt des nächsten Liniensegements ist.
Race Condition	Von einer Race Condition spricht man, wenn das Verhalten eines Programmes von der Reihenfolge der Ausführung (z.B. durch mehrere Threads) abhängig ist.

Serialisieren , Serialisieung	Ein (Java-)Objekt im Speicher in eine Form bringen, in der es über das Netzwerk übertragen werden oder auf einer Festplatte gespeichert werden kann. Das Gegenstück zur Serialisierung ist die Deserialisierung. Die Deserialisierung rekonstruiert das Objekt im Speicher.
Stack-Overflow	Ein Stack-Overflow tritt auf, wenn ein Programm zu viel Speicher auf dem Stack benötigt. Der Stack ist dabei eine Region im Hauptspeicher.
Thread	Eine Ausführungseinheit innerhalb eines Prozesses. Dabei kann ein Prozess mehrere Threads besitzen, welche parallel ausgeführt werden können.
Threadpool	Ein Pool, d.h. eine Menge an im Voraus erstellten Threads auf die zurückgegriffen werden kann.

A. Anhang

A.1. Offizielle Aufgabenstellung

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Parallelisierung eines Modells zur Simulation von Personenströmen
BA14_tham_2

BetreuerInnen:	Markus Thaler, tham Manuel Renold, reno Albert Steiner, sine
Fachgebiete:	Mobilitat und Verkehr (MVK) Software (SOW)
Studiengang:	IT
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Interne Partner:	Institut für Datenanalyse und Prozessdesign (IDP)
Industriepartner:	Savannah Simulations AG (8704 Herrliberg)
Gruppengrösse:	2

Kurzbeschreibung:
Die zunehmende Mobilität der Bevölkerung macht es notwendig, Fussgängerströme optimal durch öffentliche Anlagen zu führen. Für die Planung solcher Anlagen wir aktuell am Institut für Datenanalyse und Prozessdesign im Rahmen eines KTI-Projektes zusammen mit der Firma Savannah Simulations AG ein Software-Modell entwickelt, mit dem das Verhalten von Fussgängern in solchen Anlagen simuliert werden kann.

Das Modell besteht aus mehreren Teilmodellen:

(i) Das Bewegungsmodell beschreibt das lokale Verhalten der Fussgänger, also wie sie mit anderen Fussgängern und der Anlagegeometrie interagieren,

(ii) das Routenwahl-Modell definiert, welche Route die Fussgänger für ihren Weg zwischen Start- und Zielpunkt durch eine Anlage wählen.

Die beiden Teilmodelle sind gekoppelt, wobei u.a. das Routenwahl-Modell Vorgaben für das Bewegungsmodell liefert. Weiter unterscheiden sich die beiden Modelle in Bezug auf Häufigkeit der Ausführung: die Routenwahl muss etwa alle 5 bis 10 Sekunden neu berechnet werden, die Berechnung der Bewegung der Fussgänger erfordert hingegen eine hohe zeitliche Auflösung, da sie über Differentialgleichungen beschrieben wird.

Ziel dieser Arbeit ist es, das Software-Modell so zu parallelisieren, dass das Modell möglichst gut mit der Anzahl Fussgänger und der Anlagengrösse- und komplexität skaliert. Dazu soll ein Daten- und Verarbeitungsmodell für die parallele Verarbeitung des Simulationsmodells entworfen und auf verschiedenen Plattformen getestet werden. Als Plattformen kommen beispielsweise in Frage: Shared Memory Systeme zusammen mit GPU's und verteilte Systeme (OpenMPI).

Voraussetzungen:
Sehr gute Softwarekenntnisse, Spass an konzeptueller Arbeit und Modellierung von Anwendungen in Software.

Die Arbeit ist reserviert für:
Martin Blöchlinger (bloechma)
Roman Müntener (muentero)

Montag 2. Juni 2014 17:22

A.2. Wegweisung: GUI

Die GUI zeichnet Personen, Graphen, Zonen und Zonengrenzen sowie Zielpunkte, Mauern und Weiteres ein. Dies kann auf den ersten Blick etwas verwirrend sein, weshalb an dieser Stelle kurz auf dies eingegangen wird. Dunkelgraue Linien stellen den Weggraph dar. Jede Linie ist ein möglicher Pfad, den ein Fussgänger nehmen kann. Fussgänger werden als ausgefüllte oder ungefüllte Kreise gezeichnet. Dabei werden sie, je dichter sie stehen, grün bis rot eingefärbt. Rot gezeichnete Fussgänger stehen sehr dicht während grün gezeichnete Fussgänger nicht sehr dicht stehen. Zusätzlich besitzt jeder Fussgänger eine feine schwarze Linie, welche die Bewegungsrichtung des Fussgängers anzeigt. Wird ein Fussgänger ausgefüllt gezeichnet (d.h. nicht nur Kreisumriss), wird für dessen Berechnung der alternative Algorithmus verwendet, da er die kritische Dichte erreicht hat. Mauern werden schwarz gezeichnet. Die Zonengrenzen werden in magenta und der Sichtbarkeitsberreich wird grün gestrichelt gezeichnet. Nächste Wegpunkte von Fussgängern werden mittels kleinen blauen Kreisen dargestellt. Links und oben sind mit roten Linien und rotem Text die Masse (in Meter) angegeben.

A.3. Client

Der Client kann mit einem Parameter gestartet werden. Mit diesem Parameter kann eine Datei angegeben werden, die pro Zeile einen Befehl enthält. Diese Befehle werden ausgeführt sobald der Client gestartet wurde. Dies kann beispielsweise Verwendet werden, um beim Starten der GUI sofort eine Simulation zu starten.

Start der GUI: `java -jar client.jar -m=macro.txt`
Die Datei macro.txt beinhaltet dabei:

```
connect
gui enable
start
```

A.3.1. Befehle

Die Befehle der folgenden Tabelle können im unteren Teil des Client-Fensters eingegeben werden.

Variablen:
Die Argumente x,y können ganzzahlige Parameter sein.

Befehl	Argument	Bedeutung
clear		löscht die Ausgabe in der Konsole
connect		erstellt die Verbindung zum Master
gui	enable	aktiviert die GUI
gui	disable	deaktiviert die GUI
help		zeigt die Hilfe an (Auflistung der Befehle)
position	x,y	der Mittelpunkt der Visualisierung wird auf die Koordinaten (x,y) gesetzt
scale	x	die Visualisierung wird mit dem Faktor x Skaliert
shutdown		beendet den Master und alle Zonenaktoren
shutdown	zones	beendet alle Zonenaktoren
start		startet eine Simulation
stop		beendet eine Simulation

A.3.2. Navigation

Im Visualisierungsbereich des GUI kann man sich durch verschiedene Mausgesten bewegen:

Mausrad Hinein-/Hinauszoomen

Doppelter Linksklick Hineinzoomen

Doppelter Rechtsklick Hinauszoomen

Linksklick und Maus bewegen Ausschnitt in gewünschte Richtung bewegen

A.4. Prototypen

Im Laufe dieser Arbeit entstanden mehrere Prototypen unter anderem um mögliche alternative Algorithmen für die operative Ebene zu testen.

A.4.1. Kräfte

Der Kräfte-Prototyp benutzt einen Ansatz, bei dem sich Kräfte auf die Richtung eines Fussgängers auswirken. Dabei müssen eine Vielzahl von Kräften mit verschiedenen Faktoren gewichtet aufsummiert und auf die Richtung und Geschwindigkeit des Fussgängers angewandt werden. In Abbildung A.1 (Seite 87) sind diese Kräfte eingezeichnet. Die einzelnen Kräfte und die dafür in der Abbildung A.1 verwendeten Farben sind:

Zielfolgungskraft (Schwarz)

Die Zielfolgungskraft ist eine anziehende Kraft, die einen Fussgänger in Richtung seines nächsten Wegpunktes oder Ziels zieht. Die Zielfolgungskraft nimmt mit der Distanz zu. Diese Zunahme sorgt dafür, dass Fussgänger, die von anderen Fussgängern mitgerissen werden, wieder stärker versuchen auf ihr eigentliches Ziel zuzulaufen.

Richtungsfolgungskraft (Blau)

Ein Fussgänger versucht der Richtung von anderen Fussgängern in seiner Umgebung zu folgen. Dadurch entsteht eine minimale Gruppendynamik. Die Richtungen aller anderen Fussgänger wird als Kraft aufsummiert. Die Richtungsfolgungskraft nimmt mit der Distanz ab. Aufgrund dieser Richtungsfolgungskraft kann es vorkommen, dass ein Fussgänger von einer Gruppe anderer Fussgänger, die einen anderen Zielpunkt haben, mitgerissen wird. Im Extremfall kann ein Fussgänger sogar von einer Gruppe so eingeschlossen werden, dass er nicht mehr wegkommt. Für dieses Problem konnte jedoch noch keine Lösung gefunden werden.

Ausweichkraft (Magenta)

Ein Fussgänger versucht der Laufrichtung eines anderen Fussgängers auszuweichen. Dazu wird die Richtung des anderen Fussgängers um 90°Richtung Fussgänger rotiert und dann als Kraft angewandt. Die Ausweichkraft nimmt mit der Distanz ab.

Kollisionsverhinderungskraft (Grün)

Fussgänger versuchen von anderen Fussgängern einen Minimalabstand zu halten. Dazu wird eine abstossende Kraft in der Richtung der anderen Fussgängern zum Fussgänger angewandt. Die Kollisionsverhinderungskraft nimmt mit der Distanz ab. Möchte man statischen Hindernissen ausweichen, muss auch von diesen Hindernissen eine Kollisionsverhinderungskraft berechnet und angewandt werden.

Performance

Der Kräfte-Prototyp besitzt keine Zonen und keine Hindernisse. Ein Vergleich mit dem effektiv verwendeten Prototypen ist daher schwierig unter anderem auch deshalb, da es sich wirklich nur um einen Prototypen für die operative Ebene handelte und somit auch keine taktische Ebene vorhanden war. Ebenso wurde ein Zeitschritt von 1 Sekunde verwendet. Mit 3200 Fussgängern konnten 6 Simulationsschritte pro Sekunde berechnet werden.

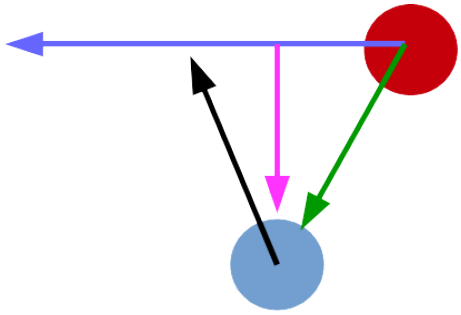


Abbildung A.1.: Kräfte im Kräfte-Prototypen

A.4.2. Akka Zonen-Prototyp

Um sich mit dem Akka-Framework vertraut zu machen und das Zonenmodell vereinfacht ausprobieren zu können, ist ein Akka-Zonen-Prototyp entstanden. Das Fenster ist fix in vier Zonen eingeteilt. Die Zonen und Fussgänger sind als Aktoren implementiert. Zudem koordiniert ein Simulation-Controller (ebenfalls ein Aktor) den generellen Ablauf der Simulation. Die Fussgänger werden an einem zufälligen Ort mit zufälliger Richtung und Geschwindigkeit erzeugt. Nachdem die Simulation gestartet wurde, berechnet jeder Fussgänger seine nächste Position. Er bewegt sich immer geradeaus mit der gleichen Geschwindigkeit. Tritt ein Fussgänger in eine andere Zone, kümmert sich der Zonen-Controller mit dem Simulation-Controller um den Austausch. Verlässt ein Fussgänger das Fenster, wird er gestoppt. Ein neuer Fussgänger wird wieder zufällig erzeugt.
Abbildung A.2 zeigt den Prototyp mit vier Zonen. Die Fussgänger sind als Kreise dargestellt. Je nach Zone werden sie in einer anderen Farbe gezeichnet.

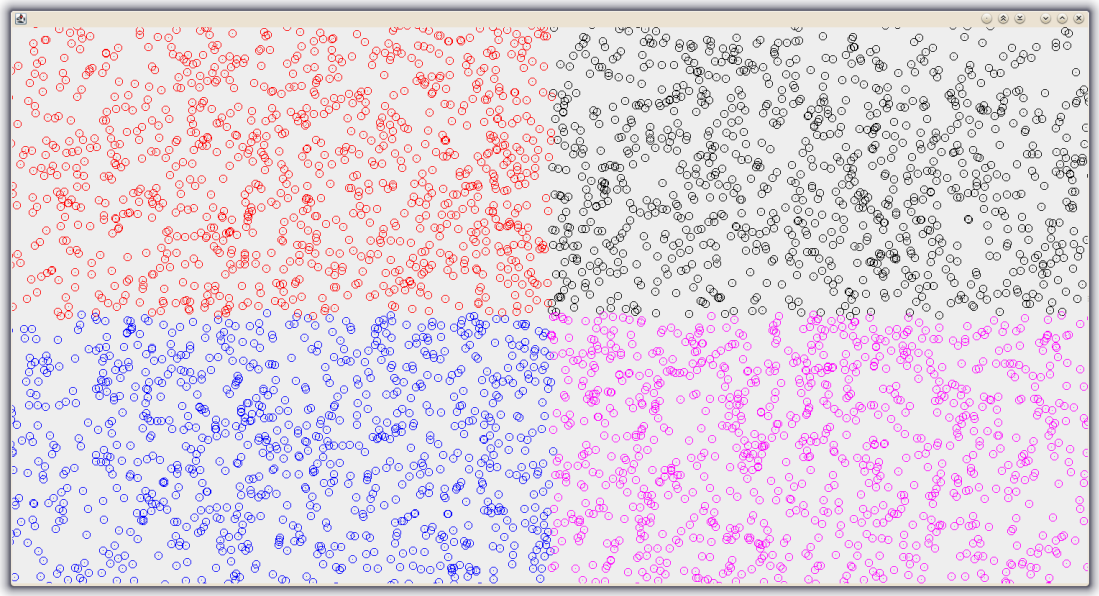


Abbildung A.2.: Fussgänger im Akka-Zonen-Prototyp

A.4.3. Visualisierung der operativen Ebene

Die operative Ebene ist komplex. Die Implementation ist umfangreich und die Konsequenzen eines Fehlers wirken sich auf die ganze Simulation aus. Wenn allerdings kleine Fehler in der Implementation bestehen, fällt dies unter Umständen bei der Simulation einer grossen Anzahl von Fussgängern nicht auf. Zur Fehlerbehebung sind visuelle Informationen sehr hilfreich, auch weil der Algorithmus die Wahrnehmung eines Fussgängers darstellt. Durch die Visualisierung einzelner Komponenten des Algorithmus kann das Verhalten viel einfacher nachvollzogen werden.
Deshalb ist eine Applikation entstanden, die Unabhängig von der eigentlichen Simulation die operative Ebene anzeigt. Einzig die Klassen für Geometrie-Objekte und die Klassen, welche die operative Ebene implementieren, werden referenziert.
Im Programmcode können beliebige rechteckige Objekte und Fussgänger platziert werden. Zudem kann ein Fussgänger, dessen operative Ebene visualisiert wird, hinzugefügt werden und dessen Start- und Zielpunkt festgelegt werden.
Nach dem Start der Applikation kann man durch einen Klick mit der linken Maustaste genau einen Simulationsschritt weiter berechnen. Dabei werden die Strahlen angezeigt, welche vom Fussgänger aus geschossen werden. Zu jedem Strahl wird ausserdem die Distanz bis zu einer Kollision, der Winkel und die berechneten Kosten angezeigt.
Durch das Drücken beider Maustasten gleichzeitig wird die Simulation mit einem Unterbruch von 40 ms zwischen den Schritten gestartet.
Abbildung A.3 zeigt einen Ausschnitt aus der gestarteten Applikation. Der betrachtete Fussgänger (rot eingezeichnet) bewegt sich auf das Ziel zu (schwarzer kleiner Kreis). Der direkteste Weg ist als gelbe Linie gezeichnet. Die Abtaststrahlen sind als grüne Linien eingezeichnet. Der Strahl, bei dem die Kosten am geringsten sind, ist blau eingefärbt. Der Sichtbereich des Fussgängers ist als grauer Kreis um den Fussgänger gezeichnet. Die schwarzen Rechtecke stellen mauern oder sonstige statische Hindernisse dar.

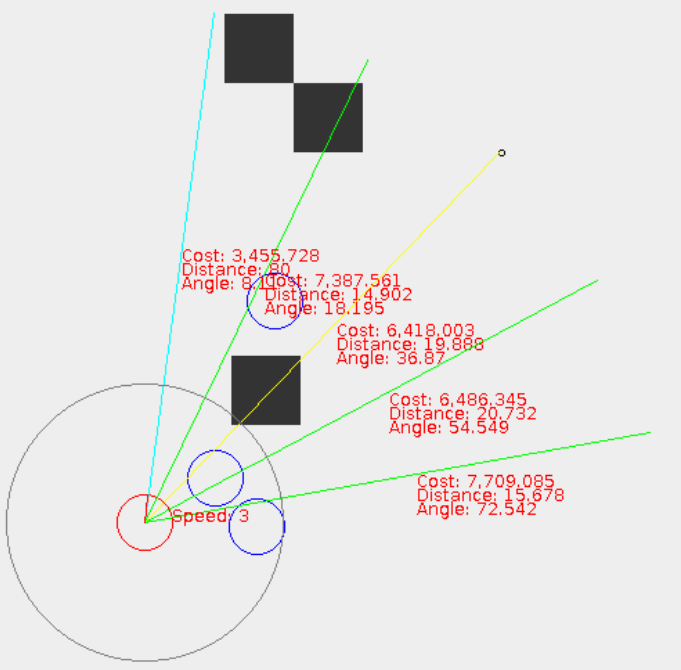


Abbildung A.3.: Applikation zum Testen der operativen Ebene

A.5. Konfiguration der Applikationen

Die einzelnen Applikationen Master, Worker und Client können jeweils durch die Datei *application.conf* konfiguriert werden. Diese Datei wird bereits für die Konfiguration von Akka benötigt, darum bot es sich an, die applikationsspezifischen Parameter ebenfalls in dieser Datei zu verwalten. Dies wird auch ermöglicht durch eine API des Akka-Frameworks, welche ein einfaches Auslesen dieser Datei ermöglicht.

A.5.1. Struktur der Konfigurationsdatei

Die Konfigurationsdatei haben eine JSON-ähnliche Struktur. Ein Parameter wird durch einen Namen definiert. Danach folgt bei einem Einzelwert ein Gleichheitszeichen gefolgt vom Wert. Wenn mehrere Werte definiert werden sollen folgt eine geschweifte Klammer. Auf die Parameter innerhalb dieses Blockes kann alternativ dazu mithilfe der Punkt-Schreibweise zugegriffen werden. Diese Blöcke werden nachfolgend als Namespaces bezeichnet.
Das folgende Beispiel zeigt zwei Äquivalente Konfigurationen:

```
# Konfiguration mit geschweiften Klammern
param_name {
    block_1 {
            param_1 = "zeichenkette"
            param_2 = 0
    }
}

# Äquivalente Konfiguration mit der Punkt-Schreibweise
param_name.block_1.param_1 = "zeichenkette"
param_name.block_1.param_2 = 0
```

Die Applikationen verwenden verschiedene Namespaces:

Master crowdsim.master

Worker crowdsim.worker

Client crowdsim.client

Innerhalb dieser Namespaces werden alle Parameter der jeweiligen Applikation definiert. Die Konfiguration befindet sich ebenfalls in diesem Namespace: also beispielsweise *crowdsim.master.akka*.

A.5.2. Master Applikationsparameter

Die folgenden Parameter können im Namespace *crowdsim.master* definiert werden:

zone.verticalZoneCount Anzahl Unterteilungen der Welt in vertikaler Richtung
Wertebereich: Ganzzahl > 0
Standardwert: 1

zone.horizontalZoneCount Anzahl Unterteilungen der Welt in horizontaler Richtung
Wertebereich: Ganzzahl > 0
Standardwert: 1

masterUrl URL des Masters
Beispiel: akka.tcp://crowdsim@127.0.0.1:2555/user/master
Werte: Protokoll, IP-Adresse und Port konfigurierbar

awaitWorkerCount Der Master wartet, bis diese Anzahl Worker sich angemeldet haben, bevor eine Simulation gestartet wurde.
Wertebereich: Ganzzahl ≥ 0
Standardwert: 0

masterAsWorker Definiert, ob Master ebenfalls Zonen zur Simulation übernimmt
Werte: true oder false
Standardwert: false

scenario.scenarioClass Java-Klasse, welche das Simulationsszenario implementiert
Werte: Eine Klasse, die das IScenario Interface implementiert
Standardwert: crowdsim.master.scenario.smallbuilding.SmallBuildingScenario

statsMaxStep Anzahl Simulationsschritte, die durchgeführt werden sollen, bevor die Simulation angehalten und die Statistik ausgegeben wird.
Wertebereich: Ganzzahl > 0
Standardwert: 999

scenario.spawnMargin Abstand der Fussgänger zueinander exklusive Körperumfang in Metern, wenn das Szenario die Fussgänger nicht manuell platziert.
Wertebereich: Dezimalzahlen > 0
Standardwert: 0.5

A.5.3. Worker Applikationsparameter

Die folgenden Parameter können im Namespace *crowdsim.worker* definiert werden:

heartbeatDelay Intervall der Nachrichten in Sekunden, die der Worker dem Master schickt, um sich anzumelden.
Wertebereich: Ganzzahl > 0
Standardwert: 1

A.5.4. Client Applikationsparameter

Die folgenden Parameter können im Namespace *crowdsim.client* definiert werden:

clientTimeout Zeit in Sekunden bis sich der Master melden muss, wenn sich ein Client anmeldet.
Wertebereich: Ganzzahl > 0
Standardwert: 10

gui.panSensitivity Sensibilität der GUI beim Verschieben des Bildschirmausschnitts
Wertebereich: Dezimalzahl > 0
Standardwert: 1

gui.scrollSensitivity Sensibilität der Zoom-Funktion mittels Mausrad
Wertebereich: Dezimalzahl
Standardwert: 1

gui.doubleClickScalefactorChange Skalierungsfaktor, der beim Doppelklick auf die Karte erhöht oder vermindert wird.
Wertebereich: Ganzzahl
Standardwert: 5

gui.refreshRate Aktualisierungsrate der GUI (alle n Millisekunden)
Wertebereich: Ganzzahl
Standardwert: 20

gui.windowWidth Fensterbreite der GUI in Pixel
Wertebereich: Ganzzahl
Standardwert: 800

gui.windowHeight Fensterhöhe der GUI in Pixel
Wertebereich: Ganzzahl
Standardwert: 900

Die folgenden Parameter bestimmen, was auf der GUI angezeigt werden soll. Dabei befinden sich die Parameter im Namespace *crowdsim.client.gui.drawing*

drawZones Definiert, ob Zonengrenzen eingezeichnet werden sollen.
Werte: true oder false
Standardwert: true

drawTransitionArea Definiert, ob die Sichtbarkeitsbereiche der Zonen eingezeichnet werden sollen.
Werte: true oder false
Standardwert: false

drawPersonCount Definiert, ob die Zusatzinformationen Personenanzahl und Skalierungsfaktor angezeigt werden sollen
Werte: true oder false
Standardwert: true

drawFocusPoint Definiert, ob ein Orientierungspunkt eingezeichnet werden soll, der die Koordinaten des Mittelpunktes des aktuellen Bildschirmausschnitts anzeigt.
Werte: true oder false
Standardwert: false

drawGraph Definiert, ob der Weggraph eingezeichnet werden soll.
Werte: true oder false
Standardwert: false

drawObstacleGraphOffsetPoints Definiert, ob die verschobenen Eckpunkte der statischen Objekte eingezeichnet werden sollen. Diese werden benötigt, um den Weggraph zu erstellen.
Werte: true oder false
Standardwert: false

drawPersonDirection Definiert, ob Richtungsvektoren der Fussgänger eingezeichnet werden sollen.
Werte: true oder false
Standardwert: true

drawPersonTargets Definiert, ob die Zielpunkte eingezeichnet werden sollen, auf welche die Fussgänger zusteuern.
Werte: true oder false
Standardwert: true

drawPersonDensityRadius Definiert, ob der Radius um einen Fussgänger eingezeichnet werden soll, der benutzt wird, um die Personendichte in seiner Umgebung zu berechnen.
Werte: true oder false
Standardwert: false

drawPersons Definiert, ob die Fussgänger gezeichnet werden sollen.
Werte: true oder false
Standardwert: true

useFancy Definiert, ob eine alternative Darstellung gezeichnet werden soll. Bei dieser Darstellung werden nicht die einzelnen Fussgänger angezeigt, sondern es wird ein Dichteprofil angezeigt.
Werte: true oder false
Standardwert: false

densityColors Definiert, ob Fussgänger, die den Algorithmus für dichtstehende Fussgänger verwenden, mit roter Farbe gefüllt werden.
Werte: true oder false
Standardwert: false

drawRuler Definiert, ob ein vertikaler oder horizontaler Massstab eingezeichnet werden soll.
Werte: true oder false
Standardwert: false

A.5.5. Akka-Konfiguration

Folgend sind die für die Kommunikation via Netzwerk benötigten Einstellungen gegeben (Auschnitt aus der Konfigurationsdatei für die Applikation). Die hohen Werte für Heartbeat-Intervall, Payload Size und Thresholds sind nötig, da ansonsten Aktoren vorzeitig als unerreichbar deklariert werden.

```
akka {
    remote {
            enabled-transports = ["akka.remote.netty.tcp"]
            netty.tcp {
                hostname = "127.0.0.1"
                port = 2567
                maximum-payload-bytes = 5000000
                maximum-frame-size = 5000000
            }

            transport-failure-detector{
                acceptable-heartbeat-pause = 100000s
                heartbeat-interval = 40 s
                threshold = 7000.0
            }

            watch-failure-detector {
                heartbeat-interval = 100 s
                threshold = 1000.0
                acceptable-heartbeat-pause = 100000 s
            }
    }
}
```

A.6. Starten der Applikationen

A.6.1. Prototypen

Da alles Java-Programme sind, können die Applikationen über die Kommandozeile entsprechend gestartet werden:

`java -jar x.jar`

Dabei ist x der Programmname:

operativelayerprototype Prototyp zum Testen der operativen Ebene

zoneprototype Akka-Zonen-Prototyp

forcesprototype Kräfte-Prototyp

routenwahlprototyp Prototyp für die Routenwahl

Üblicherweise lassen sich Java-Programme auch per Doppelklick auf die JAR-Datei starten.

A.6.2. Hauptapplikationen

Die Hauptapplikationen befinden sich je in einem Unterordner im Ordner *crowdsim*. Das Starten der Applikationen sollte in der Reihenfolge Master, Worker und schliesslich Client erfolgen.
Die Applikationen sind so konfiguriert, dass auf dem lokalen System eine Simulation mit der Karte b simuliert wird. Dabei werden die Ports 2555 (Master), 2561 (Client) und 2567 (Worker) verwendet. Dies kann in den jeweiligen Konfigurationsdateien *application.conf* geändert werden.

Master starten

Der folgende Befehl startet den Master mit einer Stack Size von 4 MB und einer Java Heap Size von 2 GB.

```
java -Xss4m -Xmx2g -jar master.jar
```

Damit der Master funktioniert, muss der Ordner mit den Karten (DXF-Dateien) im gleichen Ordner wie die Master-JAR-Datei liegen.

Worker starten

Der folgende Befehl startet den Worker.

```
java -jar worker.jar
```

Client starten

Der folgende Befehl startet den Client.

```
java -jar client.jar
```

Weitere Optionen und die möglichen Befehle für die GUI sind unter A.3 dokumentiert.

A.7. Ordnerstruktur (CD)

```
/
|- doc: Entählt den Bericht als PDF.
|- BASrc: Enthält den Quellcode (Java) des Prototypen.
|- bin: Enthält die ausführbaren Dateien der Programme
|- Messergebnisse: Rohdaten der Messresultate
    |- Konsolidierte Daten
    |- Rohdaten lokale Messungen
    |- Rohdaten Messungen Rechnerverbund
|- Prototypes: Quellcode Prototypen.
   |- AkkaAgents: Prototyp für Akka und Zonen.
   |- OperativeLayerTest: Prototyp für Test und Verifikation der operativen Ebene.
   |- Routenwahlprototyp: Prototyp für Routenwahl
   |- Forces: Kräfte-Prototyp
```