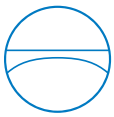


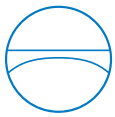
Fakultät für Bauingenieur- und Vermessungswesen
Chair for Computation in Engineering
Prof. Dr. rer. nat. Ernst Rank
Fachbereich Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

Verbesserung der Performance eines kontinuierlichen Fußgängersimulators durch Einsatz von Cell-Lists

Oliver Fuchs

Bachelor Thesis
im Studienfach Umweltingenieurwesen

Author:	Oliver Fuchs
Matrikelnummer:	[redacted]
Betreuer:	Prof. Dr.-Ing. André Borrmann
	Dipl.-Inf. Angelika Kneidl
Ausgabedatum:	01. Juni 2011
Abgabedatum:	01. September 2011



Beteiligte Organisationen

Chair for Computation in Engineering
Fachbereich Computergestützte Modellierung und Simulation
Fakultät für Bauingenieur- und Vermessungswesen
Technische Universität München
Arcisstraße 21
D-80333 München

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Bachelor-Thesis selbstständig erstellt habe. Externes Gedankengut ist als solches gekennzeichnet.

München, 2. September 2011

Oliver Fuchs

Oliver Fuchs
[redacted]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Computersimulationen haben in den letzten Jahren stark an Bedeutung dazugewonnen. Heute werden Ergebnisse aus Simulationen in fast allen Bereichen der Wissenschaft und Technik verwendet. Als Vorreiter auf dem Gebiet galten Luft- und Raumfahrt, die Automobilindustrie und die Teilchenphysik. Doch schnell wurden auch andere Fachbereiche auf den Nutzen dieser Technik aufmerksam.
Heute hilft die Simulations-Technik zum Beispiel, Prozesse im Körper besser verstehen zu können, technische Bauelemente aller Art zu optimieren, Materialflüsse und -bewegungen vorherzusagen, etc. besonders die rasche Entwicklung in der Computertechnik trägt dazu bei, dass sich der Bereich der Anwendungen noch weiter ausdehnen wird.

1.1 Simulation von Fußgängerbewegungen

Für die Simulation von Fußgänger-Bewegungen gibt es mehrere Gründe. Zum Beispiel benötigen immer weiter Wachsende Großstädte eine attraktive und leistungsfähige Fußgänger-Infrastruktur, die geplant und getestet werden muss. Als weiteres Beispiel ist die Sicherheit von Massenevr-anstaltungen zu nennen.
Die Planung und Organisation solcher Projekte überfordert die zuständigen Stellen leider häufig. Das hat in letzter Zeit immer wieder zu größeren Unglücksfällen mit Verletzten und Toten geführt. Den letzten Vorfall dieser Art haben sicher noch alle vor Augen. Im Sommer 2010 brach es auf der Love-Parade in Duisburg an einer unterschätzten Engstelle Panik aus. In der Folge kamen 21 Menschen ums Leben und viele wurden verletzt.
Eine Simulation durch Veranstalter oder Behörden im Vorfeld der Veranstaltung hätte die Problemstelle vielleicht identifiziert und das Unglück verhindern können.

Interesse an solchen Modellen zeigen aber nicht nur Staat und Verwaltung, sondern auch die Wissenschaft. Das Verhalten von Menschen in Gruppen unterscheidet sich grundlegend von dem einer Einzelperson. Die Forschung in diesem Bereich (die Biologie, Psychologie und Verhaltensforschung verbindet) wird mit Hilfe geeigneter Modelle unterstützt.
Aber auch wirtschaftliche Interessen spielen mitlerweile eine nicht unerhebliche Rolle. Große Kaufhäuser simulieren heute schon Kundenströme. Entsprechend der Ergebnisse werden Gebäude und die Platzierung von Verkaufsstätten optimiert.

Ähnlich wird auch beim Bau von Bahn- und Flughäfen verfahren. Gewinnsteigerungen durch Einsatz von Simulationen sind unumstritten.

Doch so einfach ist es nicht, menschliches Verhalten vorherzusagen. Die zentrale Frage, die sich nun stellt, ist: Wie lassen sich Fußgänger am besten simulieren?
Aus dieser Fragestellung haben sich etliche Modellansätze entwickelt, von denen viele ursprünglich aus der Physik stammen. Vergleiche dazu [1][Kap. 3] oder englischen Wikipedia-Eintrag zu „Cell lists“ (bei Interesse an weiteren Beispielen aus der Physik). Modelle zur Prognose von Atombewegungen oder Teilchenreaktionen wurden übernommen und erweitert.
Eines dieser Modelle hat sich als Grundlage weithin als verlässlich erwiesen: Das Modell für Kurzreichweiten-Potenziale aus der Gaskinetik. Analogien im Verhalten von Fußgängern und Gasteilchen sind dabei die Grundidee. Eine Anwendung dieses Modells auf Fußgänger werde ich in Kapitel 2 genauer vorstellen.

An der **TU München** am **„Fachgebiet für Computergestützte Modellierung und Simulation“(CMS)** wird momentan parallel an zwei Fußgänger-Simulatoren gearbeitet.
Der erste wird im Rahmen eine Promotion zusammen mit Siemens entwickelt. Das Ziel ist eine Evakuierungs-Simulation für den Außenbereich von Fußballstadien. Das zugrundeliegende Modell ist hierbei ein zellulärer Automat für die Bewegungen, überlagert mit einem Graph zur optimalen Wegfindung. Es handelt sich dabei um einen **diskreten** Simulator, was bedeutet, dass sich die Fußgänger nicht frei im Raum positionieren können, sondern an Zellen gebuden sind.
Der zweite Simulator, um den es auch in der folgenden Arbeit geht, ist ein internes Projekt, das bisher größtenteils von Studenten bearbeitet wurde (siehe [2]). Ziel ist die Simulation kleinräumiger Szenarien, aber auch der Gewinn von Vergleichswerten für den anderen Simulator. Ber auch der Test verschiedener Rechen-Modelle gehört zum Projektumfang.
Größter Unterschied zum anderen Simulator ist, dass es sich bei unserer Eigenentwicklung um einen **kontinuierlichen** Simulator handelt. Im Gegensatz zum diskreten Fall können sich die simulierten Objekte hier frei im Raum bewegen. Das bewirkt eine höhere Genauigkeit, die besonders in kleinskaligen Szenarien wichtig ist. Jedoch führt diese Genauigkeit auch zu einer Steigerung des Rechenaufwands.

1.2 Motivation und Thema der Arbeit

Grundlage dieser Arbeit ist **Der kontinuierliche Fußgänger-Simulator des Fachbereich CMS**, an dem auch ich im Moment arbeite. Wie schon erwähnt, dient dieser auch zum Testen von Modellen. Das momentan im Test befindliche Simulations-Modell, das im folgenden Text als **Helbing-Modell**[1] bezeichnet wird, verursacht aber noch einige Probleme.

[1] Eigentlich „Social force model of pedestrian dynamics“ von Helbing D., Molnár P. aus dem Jahre 1995. Siehe Abschnitt 2.1 oder [3] für Details

Eines dieser Probleme, das ich schon angesprochen habe, betrifft den Rechenaufwand, bzw. die benötigte Rechenzeit. In Zukunft soll der Simulator auch größere Fußgängermengen in Echtzeit - oder schneller - simulieren. Davon ist er aber heute noch weit entfernt.

Das liegt vor allem daran, dass in sehr kurzen Zeitintervallen (bis zu 20mal pro Sekunde) für jeden Fußgänger sehr viele komplizierte Potenzialberechnungen durchgeführt werden. Genauer gesagt muss jeder Fußgänger seine Interaktion mit allen anderen Fußgängern, egal wo und wie weit weg, überprüfen.
Das ist bei Kurzreichweiten-Potenzialen aber nicht sinnvoll und führt beim Berechnungs-Algorithmus zu einer Komplexität von $O(n^2)$, was sehr ungünstige Auswirkungen auf die Rechenzeit hat (siehe Abschnitt 5.2.1).

Eine Lösung des Problems, die ursprünglich auch wieder aus der Physik stammt, ist die Reduktion der Anzahl von Interaktionen. Ein Modell, das diesen Gedankengang nutzt, ist das **Linked-Cell-Modell**, auch **Cell-List**[2] genannt.
Es schränkt den Interaktionsbereich des einzelnen Objekts ein und reduziert damit den Rechenaufwand zur Laufzeit (siehe Kapitel 3). Im Optimalfall ergibt sich dann nur noch eine Komplexität von $O(n)$ und somit eine immense Rechenzeit-Reduzierung bei hohen Fußgänger-Zahlen.
Es gäbe aber auch andere Modelle, die diesen Zweck erfüllen. Ein einfacher Weg wäre die Überprüfung des Absolut-Abstandes zwischen zwei Fußgängern, jedoch kann z.B. die sichtblockierende Wirkung von Wänden dabei nicht berücksichtigt werden und zur Laufzeit wären wieder $n * (n - 1)$ Rechenoperationen notwendig.
Die Cell-List hat sich deshalb in der Vorauswahl gegen alle anderen Lösungswege durchgesetzt.

Meine Aufgabe ist es nun, zu testen, ob sich die Cell-List zur effektiven Reduktion der Rechenzeit des Helbing-Modells eignet, oder nicht. Dazu werde ich erst den Simulator und das Helbing-Modell kurz erklären, danach auf die Theorie und die Implementierung der Cell-List eingehen und zum Schluss einen aussagekräftigen Praxistest vorstellen.
Das Ergebnis dieser Arbeit wird darüber entscheiden, ob die Cell-List danach als funktionaler Bestandteil in das Projekt integriert wird, oder eine andere Lösung gesucht werden muss.

[2] Der Begriff Linked-Cell-Modell stammt daher, dass die Zellen ursprünglich als linked list (z.B. in der Programmiersprache c) verwaltet wurden. Ich werde im folgenden Text den Begriff Cell-List verwenden, der für meine Implementierung korrekter ist.

Kapitel 2

Kontinuierlicher Fußgänger - Simulator

Bei diesem Kapitel handelt es sich nicht direkt um einen Teil meiner Bachelor-Thesis, sondern um die Grundlage dafür. Ich werde hier einen kurzen Überblick über den Aufbau und die Funktion des Simulators geben, um den es in der folgenden Arbeit geht. Ohne diese Grundlage wäre der Rest der Arbeit unverständlich und zusammenhanglos.

2.1 Kontinuierliches Modell nach Helbing

Bevor ich auf die technischen Details eingehe, möchte ich erst einmal das zugrundeliegende Modell erklären.

Entwickelt wurde es in den '90er-Jahren des vergangenen Jahrhunderts von Dirk Helbing und Peter Molnár. Ich beziehe mich auf eine Version von 1995, die auch im Simulator implementiert ist. Die mathematischen Inhalte des folgenden Absatz sind eine Zusammenfassung von [3][361-367].
Grundlage für das Fußgängermodell waren, wie schon erwähnt, physikalische Teilchen-Modelle und viele Beobachtungen. Die Idee ist nun, dass Fußgänger, wie physikalische Teilchen, von Kraftfeldern im Raum beeinflusst werden. Die resultierenden Kräfte daraus heißen **social forces** ($\vec{f}(t)$) und lassen sich als ablenkende Beschleunigungskräfte interpretieren.

Die Fußgänger folgen damit den einfachen physikalischen Bewegungsgleichungen[1] 2.1 und 2.2

$$\vec{v}(t) = d\vec{s}(t)/dt \tag{2.1}$$

[1] $\vec{v}(t)$: Geschwindigkeit eines Fußgängers(FG), $\vec{s}(t)$: Position eines FG, t: Zeit, $\vec{a}(t)$: Beschleunigung eines FG

$$\vec{a}(t) = d\vec{v}(t)/dt \tag{2.2}$$

Im massenfreien System können wir nun Kräfte und Beschleunigungen gleichsetzen[2] (Vorsicht! Physikalisch nicht korrekt!)

$$\vec{f}(t) = \vec{a}(t) \tag{2.3}$$

Da verschiedene Kräfte[3] bzw. Beschleunigungen auf die Fußgänger einwirken, teilen wir den Vektor $\vec{f}(t)$ noch auf

$$\vec{f}(t) = \sum \vec{f}_i(t) \tag{2.4}$$

Gleichung 2.5 zeigt, aus welchen verschiedenen Einflüssen bzw. Einzelkräften sich $\vec{f}$(t) zusammensetzt[4]

$$\vec{f}(t) = \vec{f_{self}} + \vec{f_{obs}} + \sum \vec{f_{int,i}} + \sum \vec{f_{attr,j}} + R \tag{2.5}$$

Die richtungsändernde Kraft auf den einzelnen Fußgänger zur Zeit t setzt sich also aus **Eigenantrieb** (Zielorientierung), **Abstoßung** durch **Hindernisse** und **andere Fußgänger** und **Anziehung** durch **attraktive Objekte** (z.B. Schaufenster) zusammen. Die Fluktuation spielt in unserem Fall keine Rolle.
Die jeweiligen Kräfte berechnen sich aus entfernungsabhängigen (z.B. Abstand Fußgänger - Wand) **Kurzreichweiten-Potenzialen**, die entweder abstoßend oder anziehend wirken. Der Fußgänger bewegt sich letztendlich entlang der Gradiente dieser Potenziale, die man auch zu einem kompletten Feld überlagern könnte.
Wie diese Einzelpotenziale im genauen aussehen und sich berechnen, werde ich hier nicht beschreiben, da das zu weit führen würde. Es sei nur erwähnt, dass im hier verwendeten Simulator eine relativ einfache, aber trotzdem schon komplizierte, Berechnung mit glockenförmigen Potenzialen durchgeführt wird. Die Reichweite der Potenziale beträgt dabei teilweise nur $4[m]$.

Auf eine der Einflussgrößen möchte ich dennoch etwas genauer eingehen, da sie später noch wichtig sein wird. Dabei handelt es sich um die **Interaktion mit anderen Fußgängern** ($\sum \vec{f_{int,i}}$).
Bei der Berechnung dieser Größe wird zu jeder Zeit t die Interaktion jedes Fußgängers mit allen anderen Fußgängern berechnet. Dabei erfolgt für jeden anderen Fußgänger eine komplette Potenzialberechnung, ob er sich nun im Wirkungsradius befindet oder nicht. Das macht diesen Schritt zum aufwendigsten Rechenschritt des Modells und erzeugt gewisse Probleme. Siehe dafür auch Abschnitt 2.3.3.

[2] $\vec{f}(t)$: alle social forces vektoriell addiert
[3] $\vec{f_i}(t)$: einzelne social force (Einflussgröße)
[4] $\vec{f_{self}}$: Eigenantrieb des FG, $\vec{f_{obs}}$: Abstoßung durch Hindernisse, $\sum \vec{f_{int,i}}$: Interaktion mit anderen FG, $\sum \vec{f_{attr,j}}$: Anziehungen durch attraktive Dinge, R: Fluktuationsterm

Damit ist die Theorie, die hinter dem Helbing-Modell steckt, auch schon abgehandelt.
Implementiert wurde es nun so, dass zu jedem Zeitschritt t (Schleife) die verschiedenen Einflüsse einzeln berechnet werden (die Attraktivität ist jedoch noch nicht implementiert). Dafür sind jeweils Potenzialrechnungen notwendig.
Danach werden die Einflüsse zu einer Gesamtbeschleunigung aufsummiert. Mit Hilfe dieser wird die neue Geschwindigkeit (Vektor) des Fußgängers unter Zuhilfenahme der Zeit berechnet. Zum Schluss wird der Fußgänger mit dieser neuen Geschwindigkeit bewegt.

2.2 Aufbau und Funktion

2.2.1 Aufbau

Grafik 2.1 zeigt den groben Aufbau des Simulators.

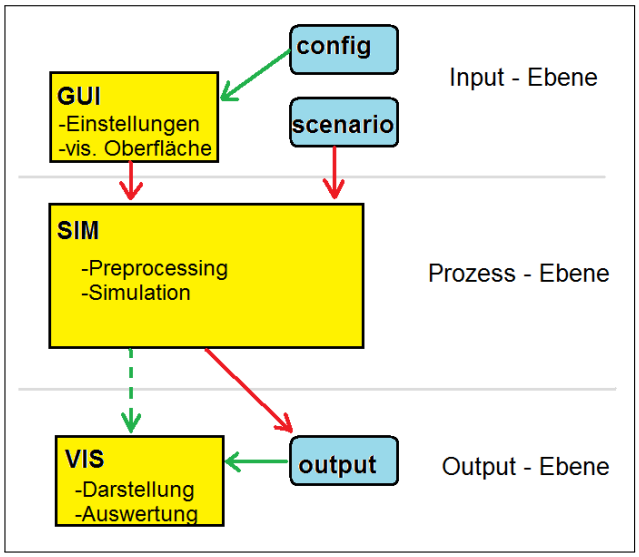


Abbildung 2.1: Ebenen des Simulators

(rote pfeile: notwendiger Datenfluss, grüne Pfeile: optional, gestrichelt: geplante Funktion)

Wie man sehen kann, ist der Simulator in drei logische Abschnitte unterteilt.
In der Input-Ebene werden alle Informationen von außen (Einstellungen, Szenario, config, etc.) gesammelt. Diese werden überprüft und in ein Paket zusammengefasst.
Dieses wird dann an die Prozess-Ebene weitergegeben, wo die eigentliche Simulation abläuft. Nachdem diese beendet ist, werden die gewonnenen Informationen (Simulationsdaten) an die Output-Ebene weitergegeben. Dort werden sie formatiert und dem Benutzer als lesbare Ausgabe (im Moment nur als*.txt-Datei) zur Verfügung gestellt.

Im Package-Explorer von Eclipse sieht das folgendermaßen aus:

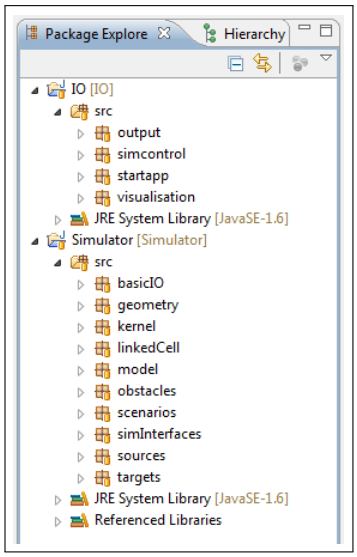


Abbildung 2.2: Einzelne Pakete des Simulators

Die Input- und Output-Ebene sind im Projekt **IO** zusammengefasst. Das Projekt **Simulator** stellt die Prozess-Ebene dar. Wichtigste Klasse ist die **Klasse Simulator**, die im operativen **Paket kernel** liegt. Die beiden Projekte sind über Schnittstellen miteinander verbunden. Eine genaue Beschreibung des Quellcodes würde an dieser Stelle zu lange dauern und ist auch garnicht das Thema dieser Arbeit. Ich verweise dafür auf die DVD, die den kompletten und auskommentierten Code enthält.

2.2.2 Ablauf einer Simulation

Was nun aber interessant und auch für spätere Kapitel wichtig ist, ist der Ablauf einer Simulation. Zu diesem Zweck werde ich einen Simulationsvorgang der Reihe nach durchgehen und auf wichtige Funktionen genauer eingehen.

GUI

Startet man das Simulationsprogramm, sieht man als erstes die **GUI** aus Grafik 2.3. Hier werden alle Grundeinstellungen vorgenommen (=Input-Ebene)

Die einzelnen Punkte bedeuten dabei:

- **File:** Laden und Speichern von config-Datein möglich
- **Scenario-Geometry:** Pfad der *.XML-Datei, in der die Szenario-Geometrie in einem festgelegten Format abgelegt ist
- **Output-Folder:** Pfad, wo die Output-Dateien gespeichert werden sollen

- **Timestep:** Zeit, die zwischen zwei Simulationsschritten vergeht (50[ms] bisher Standardwert)
- **Evacuees:** Anzahl der zu simulierenden Fußgänger
- **Max Evac Time:** Zeit, die maximal zur Evakuierung aller Fußgänger gebraucht werden darf
- **Visualisation:** noch ohne Wirkung
- **Simulation Structure:** keine anderen Auswahlmöglichkeiten. Dient zur Erweiterung mit anderen Modellen
- **Log:** zeigt alle wichtigen Events während der Simulation an
- **Start/Stop Simulation:** selbsterklärend

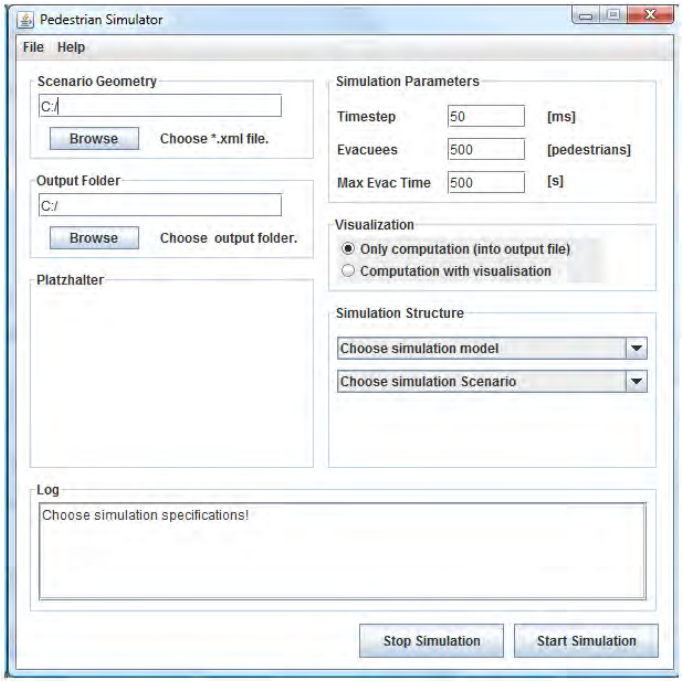


Abbildung 2.3: GUI im Grundzustand

Hat man alle Einstellungen vorgenommen, beginnt die Simulation mit einem Druck auf **Start Simulation**.

Simulations-Vorgang

Die Simulation läuft nun in zwei Schritten ab. Als erstes wird im Preprocessing alles für den Beginn der Simulation vorbereitet. Alle notwendigen Objekte werden generiert.
Dann startet die Simulationsschleife, die wie folgt abläuft:

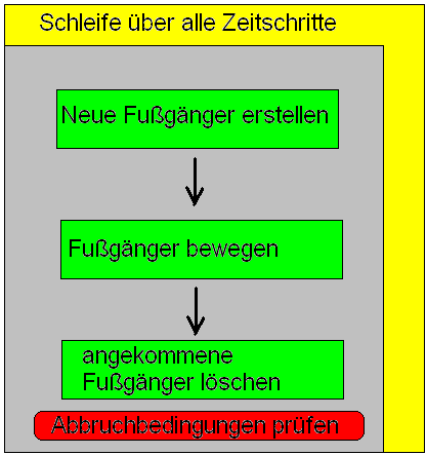


Abbildung 2.4: Ablauf einer Simulation

Eine äußere Schleife arbeitet alle Zeitschritte (bis Max Evac Time erreicht ist) ab, falls vorher keine andere Abbruchbedingung wirksam wird. Die anderen Bedingungen sind entweder ein Systemfehler oder das erfolgreiche Evakuieren aller Fußgänger (Evacuees).
Innerhalb eines Zeitschrittes werden unter Anderem alle Berechnungen für das Helbing-Modell durchgeführt. Es werden aber auch Unter-Schleifen über Quellen, Ziele und alle Fußgänger durchlaufen.

Da der Schritt **Fußgänger bewegen** der ist, in dem später auch die Cell-List aktiv sein wird, betrachten wir diesen Schritt nochmal besonders:

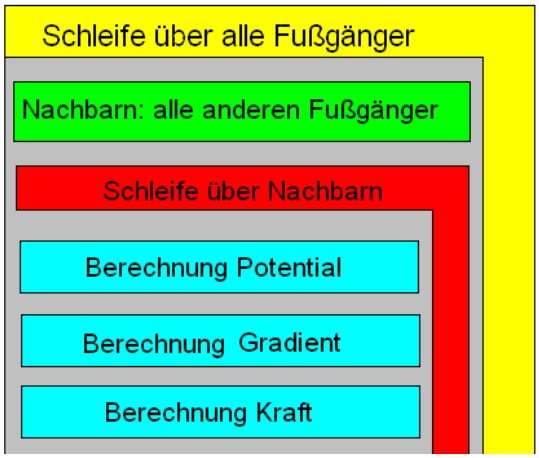


Abbildung 2.5: Schleife über alle Fußgänger

Als erstes werden die Interaktionspartner (Nachbarn) des aktuellen Fußgängers ausgewählt. Das sind im Moment noch alle anderen Fußgänger. Danach wird die Interaktion mit allen Nachbarn berechnet (Helbing-Modell). Zum Schluss wird der Fußgänger bewegt.

Ist schließlich eine Abbruchbedingung erreicht worden, stoppt der Simulator und die Output-Ebene übernimmt die weiteren Schritte.

Output

Am Ende der Simulation gibt der Simulator einen unformatierten Output-String zurück. Dieser enthält Positionsangaben aller Fußgänger zu allen Zeitschritten in Koordinaten. Dieser String wird nun formatiert und mit einigen anderen Informationen (Geometrie, Startwerte, etc.) in eine *.txt-Datei geschrieben.
Diese relativ große Datei kann nun für Auswertungen verwendet werden. Am einfachsten geht das mit Hilfe eines Post-Visualierungs-Tools (hier SinoView von Siemens). Das kann dann z.B. folgendermaßen aussehen:

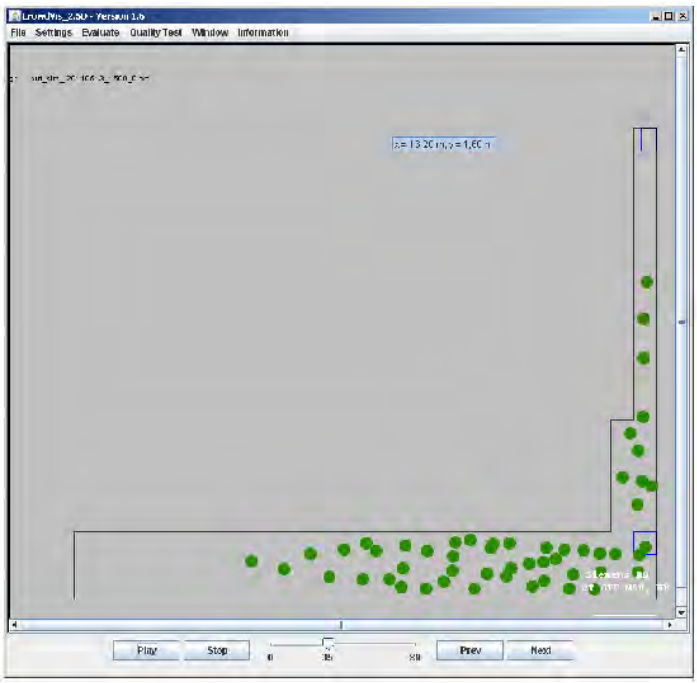


Abbildung 2.6: Auswertung mit SinoView

2.3 Probleme

Es gibt im verwendeten Modell noch einige Probleme. Drei davon möchte ich etwas genauer betrachten und mögliche Lösungsansätze kurz darstellen.

2.3.1 Sichtweiten-Problem

Beschreibung:

Die Potenziale des Helbing-Modells sind Kurzreichweiten-Potenziale. Diese sind auf eine Reichweite unter $10[m]$ eingeschränkt. Das führt dazu, dass ein Fußgänger in unserem Fall nur etwa $4[m]$ weit sieht.
Das Problem ist nun, dass besonders größere Pulks von Fußgängern auch auf größere Entfernung unser Bewegungsverhalten beeinflussen. Diese werden im Moment aber völlig ignoriert.

Mögliche Lösung:

Solche Pulks könnten nun z.B. intern als Hindernisse modelliert werden. Diese würden mit einem manuell festgesetzten sehr hohen Potenzial versehn werden, das auch auf Entfernung noch wirksam ist. Damit würde die Sichtweite wenigstens in solchen speziellen Situationen erhöht werden.

2.3.2 V-Problem

Beschreibung:

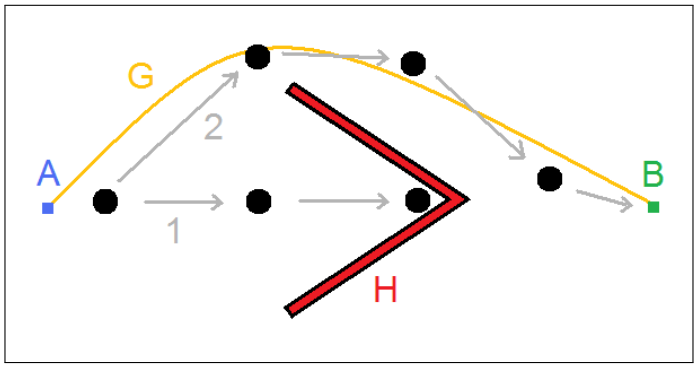


Abbildung 2.7: Das V-Problem

Grafik 2.7 illustriert das V-Problem sehr gut. Die Fußgänger wollen hier von Punkt A nach Punkt B. Das V-förmige Hindernis H befindet aber im Weg. So wie das Modell im Moment funktioniert, wird immer der kürzeste und direkte Weg zum Ziel gewählt (Weg 1). Damit bleiben die Fußgänger im Trichter hängen.

Mögliche Lösung:

Ein Graph, der als zusätzlicher Layer über dem aktuellen Modell liegt, könnte die Wegfindung optimieren. Der Graph bemerkt das Hindernis, berechnet einen Weg außen herum (Weg 2) und gibt dem Fußgänger z.B. Zwischenziele entlang der Linie G. Mit diesen kann der Fußgänger wieder dem Modell folgend den neuen kürzesten Weg wählen und kommt trotzdem um das Hindernis herum.

2.3.3 Rechenzeit-Problem

Beschreibung:

Aufgrund von komplexen und teilweise überflüssigen Berechnungen und vielen Iterationen benötigt das Modell, wie schon beschrieben, sehr große Rechenkapazitäten. Auf normalen Rechnern führt das zu astronomischen Rechenzeiten (siehe Abschnitt 1.2).

Mögliche Lösung:

Dieses Problem lässt sich von verschiedenen Seiten angehen. Eine einfache Möglichkeit ist es, díe Zahl der Iterationen pro Sekunde zu reduzieren. Das hilft aber nur im kleinskaligen Bereich.
Eine weitere Möglichkeit wäre das Tabellieren der quasilinearen Potanziale im Preprocessing, so dass man sich die Potenzialrechnung zur Laufzeit spart.

Ich wähle jedoch zuerst einen anderen Weg. Mein Ziel ist die Reduzierung der Interaktionen zwischen den Fußgängern und damit auch die Anzahl der Kraftberechnungen.
Eine Möglichkeit, das zu realisieren, stellt die Implementierung einer Cell-List dar. Wie und ob das funktioniert, werde ich in den folgenden Kapiteln genauer erleutern.

Kapitel 3

Cell-List: Theorie

In diesem Kapitel wird kurz auf die Theorie hinter der Cell-List eingegangen und wie das ganze am besten programmiertechnisch umzusetzten ist.

Die Idee hinter diesem Modell ist, wie bereits erwähnt, eine Einschränkung des „Interaktionsradius“ oder „Sichtradius“ der beteiligten Objekte. Diese Einschränkung muss bei allen Modellen, die größere Objektmengen in einer Umgebung mit Kurz-Reichweiten-Potenzialen darstellen, irgendwann vorgenommen werden.
Tut man das nicht, nimmt die Leistungsfähigkeit mit steigender Objektzahl rapide ab. Doch die Einschränkung birgt auch Risiken. Bei falsch gewählter Sichtweite kann das Ergebnis erheblich verfälscht werden. Es gilt daher, sich vorher gut zu überlegen, wo und wie man auf die Berechnung Einfluss nimmt.

Was die Geschichte des Cell-List-Modells angeht, lässt sich nicht viel herausfinden. Überlegungen dieser Art waren von Anfang an Inhalt von physikalischen Simulationen. Einen Erfinder oder eine allgemeine Theorie gibt es im eigentlichen Sinne nicht.
Die theoretischen Überlegungen und die Implementierung ist immer abhängig von den Eigenheiten des zu simulierenden Mediums und des Simulators.

Die folgende Theorie ist daher meine eigene Interpretation des allgemein bekannten Linked-Cell-Modells. Sie ist, abgesehen von gewissen Grundgedanken, nur im Rahmen des hier betrachteten Simulators anwendbar und korrekt.
Externe Gültigkeit müsste jeweils im Experiment nachgewiesen werden.

3.1 Der effektive Radius

Der **effektive Radius** (R_{eff}) ist die zentrale Größe in diesem Modell. Er beschreibt kreisförmig um ein Objekt herum dessen Interaktions-Reichweite.

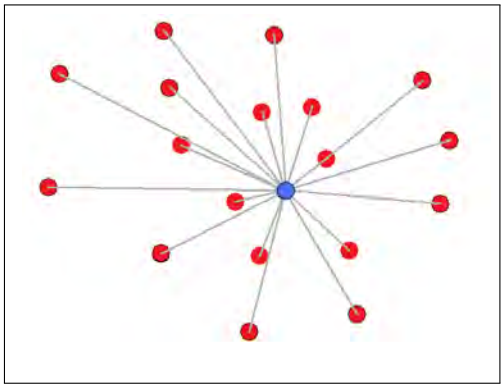


Abbildung 3.1: Raum mit unbeschränkter Interaktions-Reichweite

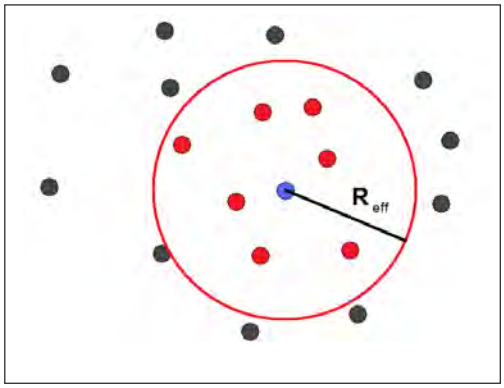


Abbildung 3.2: Beschränkte Interaktions-Reichweite

In Abbildung 3.1 ist die Ausgangs-Situation illustriert. Das blaue Objekt wird durch alle roten Objekte beeinflusst, egal wie groß die Entfernung zu diesen ist. Zur Laufzeit führt das zu folgender Komplexitäts-Rechnung (vgl [4] zur Durchführung der folgenden Komplexitätsrechnungen):

Sei n die Anzahl aller Objekte im Modell.

Zu jedem Zeitschritt muss nun jedes der n Objekte mit jedem anderen, also mit $(n-1)$ Objekten, seine Interaktion durchführen. In einem Zeitschritt ergibt sich so eine Anzahl von A_{Int} Interaktionen:

$$A_{Int,ohneCL} = n(n-1) = n^2 - n \tag{3.1}$$

Da in unserem Modell jedes Objekt auch eine Eigen-Interaktion (self driving force) besitzt, kommen sogar nochmal n Interaktionen dazu. Damit sind wir genau bei n^2, was bei dieser Betrachtung zur Komplexität des Rechen-Algorithmus von $O(n^2)$ führt.

Betrachten wir nun im Vergleich dazu Abbildung 3.2. Hier wird die Interaktions-Reichweite des blauen Objekts auf den **Effektiv-Radius** R_{eff} eingeschränkt. Nun ist die Zahl der roten Objekte, die jetzt **effektive Nachbarn** heißen, innerhalb des Kreises limitiert. Betrachten wir dazu wieder die Komplexitäts-Rechnung:

Sei n wieder die Anzahl aller Objekte im Modell.

Nun muss jedes Objekt n zu jedem Zeitschritt nur noch mit einer festen Zahl k von Objekten interagieren[1]. Das führt wieder zu A_{Int} Interaktionen.

$$A_{Int,mitCL} = nk \tag{3.2}$$

Auch hier müssen wir wieder die n Selbst-Interaktionen hinzuaddieren. Es kommt auch noch eine Konstante p, die das Preprocessing und damit den zusätzlichen Aufwand zum Aufbau des R_{eff} abdeckt, hinzu.
Damit wird Gleichung 3.2 zu Gleichung 3.3:

$$A_{Int,mitCL} = nk + n + p = n(k+1) + p \tag{3.3}$$

Nun gehen wir davon aus, dass in einem realen Szenario mit vielen hundert Objekten das Verhältnis von k zu n sehr klein ist ($k << n$). Da es sich bei k und p um Konstanten handelt, sind sie für $lim_{n\to\infty}$ zu vernachlässigen. Damit sinkt die Komplexität des Rechen-Algorithmus für auf $O(n)$. Das theoretische Ziel ist damit erreicht.

---

[1] Dieses k ist in unserem Modell keine Funktion von n. Bei steigendem n steigt zwar anfangs bei Volumengleichheit auch k. Dieser Anstieg hört jedoch auf, wenn ein Grenzwert erreicht ist (Interaktions-Radius voll mit Fußgängern). Ab da kann n weiter steigen, ohne k zu beeinflussen. Ist k eine Funktion von n, stimmt folgende Rechnung nicht!

3.2 Das Gittermodell

Nun muss das vorher beschriebene Modell in programmierbare Form gebracht werden.

Die Diskretisierung des Raumes als Gitter bietet die passende Lösung. Ein Gitter in zwei Raumrichtungen ist programmiertechnisch einfach als zweidimensionales Feld umzusetzen. Die Zellen erhalten dabei jeweils einen Zeilen- und Spaltenindex.
Dieser später auf das Simulations-Szenario aufgelegte Gitter-Layer wird im Preprocessing aus den Ausmaßen des zu simulierenden Raumes berechnet. Abbildung 3.3 zeigt eine Illustration der fertigen Struktur.

In einer Gitterzelle können sich natürlich auch, je nach Größe, mehrere Objekte befinden.

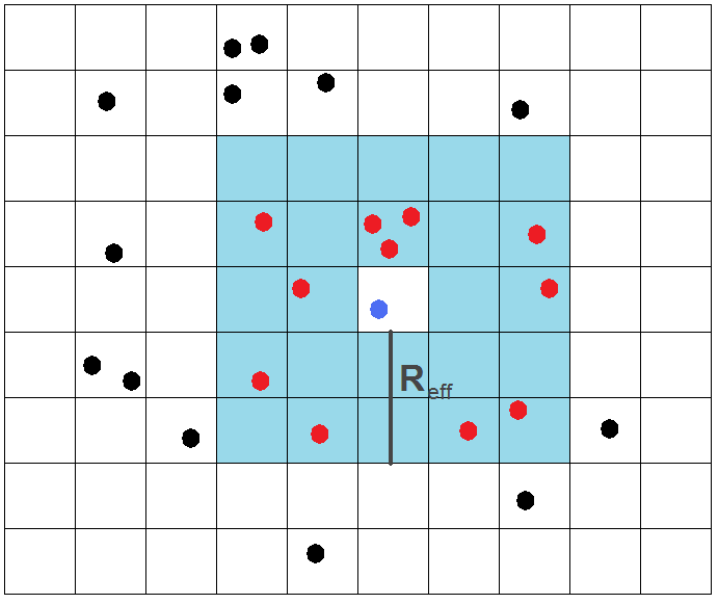


Abbildung 3.3: Raum in Gitter-Struktur mit effektivem Radius

Das hier dargestellte Gitter besteht aus quadratischen Zellen (Höhe = Breite). Diese Form ist die einfachste und gleichzeitig effektivste. Die Seitenlänge der Gitterzellen ist eine der Variablen, die je nach Simulations-Szenario vor dem Preprocessing festgelegt werden muss.

Zusätzlich sieht man im Bild, wie der effektive Radius in das Gittermodell integriert wurde. Von der Zelle aus, in der sich das Referenz-Objekt befindet, wird jeweils in x-Richtung und y-Richtung eine ebenfalls vorher festgelegte Zahl von Zellen erst addiert und dann subtrahiert.Daraus ergibt sich eine Bounding-Box. Alle Objekte innerhalb dieser Box befinden sich im **effektiven Gebiet**(A_{eff}). So heißt die Zone, die vorher der effektive Radius umschlossen hat, jetzt.

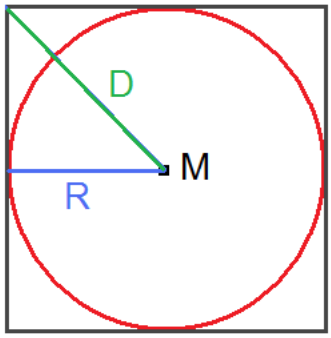


Abbildung 3.4: Größendifferenz zwischen R_{eff} und A_{eff}

Die runde Form und damit auch der konstante Abstand zum Rand gehen dadurch verloren.

Abbildung 3.4 zeigt den Unterschied zwischen Kreis- und Rechteckform. Dabei wird vereinfachend davon ausgegangen, dass sich das Referenz-Objekt auch weiterhin im Zentrum der Interaktions-Zone befindet. Rechnerisch muss A_{eff} nun so bestimmt werden, dass der blaue Radius R dem R_{eff} von vorher entspricht.
Die Tatsache, dass das Objekt damit um bis zu $\sqrt{2}$ entlang der Diagonale D weiter „sehen“ kann, spielt bei der Simulation keine Rolle. Es besteht natürlich trotzdem die Möglichkeit im Postprocessing die Form des Gebietes wieder annähernd kreisförmig zu gestalten. Davon wird hier aber abgesehen.

Befindet man sich nun in einem objektorientierten Programmier-Umfeld, was ich voraussetze, lassen sich die einzelnen Zellen als Objekte modellieren. Diese werden in dem vorher generierten Feld untergebracht und übernehmen die Verwaltung der Simulations-Objekte (Fußgänger, etc.).

Die eigentliche Aufgabe der Cell-List ist es nun, für jedes Simulations-Objekt bzw. dessen Zelle das **effektive Gebiet** zu bestimmen. Damit lassen sich in der Folge die **effektiven Nachbar-Zellen** und über diese die **effektiven Nachbarn** ermitteln.

Kapitel 4

Cell-List: Implementierung

Im folgenden Kapitel wird die Umsetzung der Theorie in ein Programm behandelt. Die Programmierarbeiten sind natürlich alle, wie beim Simulator auch, in der Programmiersprache Java erfolgt. Als Hilfsmittel habe ich dabei [5] benutzt.
Die wichtigsten Funktionen und Methoden werden kurz erklärt. Zur Veranschaulichung von Quellcode werde ich Pseudocode benutzen. Für die komplette Implementierung verweise ich auf den ausführlich auskommentierten Code der Cell-List in Anhang A oder den kompletten Code auf der DVD.

4.1 Idee und Vorgehen

Die Implementierung der Cell-List erfolgt nun auf Grundlage der im letzten Abschnitt gezeigten Theorie. Dass es sich bei dieser Implementierung erstmal nur um ein Test-Modell handelt, wird im folgenden Kapitel deutlich. Die endgültige Struktur und wichtigen Funktionalitäten sind zwar bereits vorhanden, aber evtl. nicht in der Form, wie man sie in einem fertigen Programm erwarten würde. An der Aussagekraft der Testergebnisse ändert das aber nichts.

Bei der Implementierung des Modells entschied ich mich für eine Dreiteilung. Inhalt und Aufgabe dieser drei Klassen ist:

- Die Klasse **LCModel** dient als Oberklasse bzw. Zugriffsklasse. Über ein einzelnes Objekt werden so alle externen Funktionsaufrufe für die Cell-List zentral gesteuert. Die Anzahl beschränkt sich zur Laufzeit logischerweise auf ein Objekt dieser Klasse. Sollten zukünftig mehrere Cell-Lists (z.B. für Hindernisse) aktiv sein, sind sie alle unter diesem einen LCModel organisiert.
- Die Klasse **LCList** verwaltet das Gitter, genauer gesagt das Feld, in dem die einzelnen Zellen-Objekte liegen. Sie enthält mit der Methode zur Identifizierung der effektiven Nachbarn die wichtigste Methode des ganzen Modells. Wie schon erwähnt, könnten rein theoretisch mehrere Objekte dieser Klasse vorliegen.

- Die Klasse **LCElement** repräsentiert die einzelnen Zellen einer Cell-List. Jede dieser Zellen enthält eine Liste der Simulations-Objekte, die sich gerade darin aufhalten. Dazu kommt eine Liste an effektiven Nachbar-Zellen. Es gibt zur Laufzeit so viele Objekte dieser Klasse, wie es Gitter-Zellen gibt.

Diese Dreiteilung wäre nur zum Testen der Cell-List eigentlich nicht notwendig gewesen. Da ein positives Ergebnis aber sehr wahrscheinlich ist, habe ich mich im Hinblick auf zukünftige Weiterentwicklungen für dieses Vorgehen entschieden. Das widerspricht zwar der Tatsache, dass ich alles so einfach wie möglich halten wollte, aber in diesem Fall hat sich am Progrmmieraufwand nichts geändert.

Der Arbeitsplan, nach dem ich bei der Implementierung vorging, sieht folgendermaßen aus:

1. Erstellen des Pakets **linkedCell**
2. Erstellen der drei Klassen **LCModel**, **LCList** und **LCElement**
3. Erstellen der wichtigsten Methoden und Funktionen der drei neuen Klassen
4. Identifizieren der Schnittstellen zum existiernden Simulator
5. Einfügen von neuen Variablen und Methoden in die existierende Klasse **Simulator**
6. Ausprogrammieren der Funktionen
7. Erstellen spezieller Methoden für den folgenden Test

4.2 Programm-Ablauf

Nun wollen wir sehen, was mit aktivierter Cell-List zur Laufzeit geschieht. Obwohl dieser Abschnitt eher an das Ende des Kapitels passen wüde, schiebe ich ihn an dieser Stelle ein. Das erleichtert später bei der Erklärung des Quellcodes die Zuordnung der einzelnen Code-Elemente. Da der Ablauf ohne Cell-List schon in Abschnitt 2.2) vorgestellt wurde, werde ich hier nur auf die Unterschiede eingehen.
Den Anfang macht das Preprocessing:

- Bis zur Generierung des Simulator-Objekts bleibt alles unverändert
- Mit dem Simulator-Objekt wird nun zusätzlich der Konstruktor von LCModel aufgerufen. Der wiederum berechnet und erstellt ein Gitter-Objekt der Klasse LCList, das sofort mit Zellen-Objekten der Klasse LCElement gefüllt wird
- Während des Aufbaus der Cell-List werden nun sofort für jede Zelle des Gitters die effektiven Nachbar-Zellen bestimmt und als Liste von Verweisen in den jeweiligen Zellen abgelegt. Dieser Weg ist nicht unbedingt effektiv, besonders für große Szenarien. In der Testphase habe ich ihn dennoch gewählt, um möglichst viel Rechenaufwand von der Laufzeit ins Preprocessing zu verlagern. Sollte die Cell-List fest ins Projekt integriert werden, wird dieser Schritt auf jeden Fall nochmal überarbeitet werden

Das Preprocessing ist nun abgeschlossen und die eigentliche Simulation beginnt. Folgende Vorgänge geschehen innerhalb jedes Zeitschritts:

- Als erstes werden Fußgänger generiert. Diese melden sich nun aber sofort bei der Erstellung in der Quelle an der Cell-List an
- Die Berechnung der Bewegung bleibt bis auf einen Unterschied gleich. Die Bestimmung der Interaktions-Partner ändert sich. Ohne Cell-List wird einfach die komplette Fußgängerliste zur Interaktion übergeben. Mit Cell-List wird dagegen aus den Fußgängern in den effektiven Nachbarzellen eine neue Liste an effektiven Nachbarn zusammengestellt und damit weitergerechnet
- Nach der Bewegung wird für jeden Fußgänger überprüft, ob seine Position noch innerhalb der zugewiesenen Zelle ist. Falls nicht, erfolgt eine Neuzuweisung
- Hat ein Fußgänger sein endgültiges Ziel erreicht, wird er aus der Cell-List wieder entfernt

Der Programmablauf sollte nun klar sein. Falls doch noch Unklarheiten bestehen, verweise ich nochmals auf den kompletten Quellcode auf der DVD.

4.3 Schnittstellen zum Simulator

Zusätzlich zu den neuen Klassen sind natürlich auch kleine Veränderungen am existierenden Simulator notwendig, um die Cell-List zu integrieren. Dazu gehört das Einfügen neuer Variablen, die Anpassungen existierender Methoden und der Aufruf der neuen Funktionen. Die wichtigsten genannten erkläre ich im Folgenden:

4.3.1 Variablen

```
private String outDir;
private String specOut = "";
private boolean messung1 = false;
private boolean messung2 = false;

// Implementierung des Linked-Cell-Models. Alle Einstellungen sind an dieser
// Stelle im Code vorzunehmen!
private LCModel linkedCell = null;
private boolean triggerLC = true;// Schalter für LCM (true=aktiviert)
private final double cellSize = 1.0;// Seitenlänge der Zellen
private final int effectiveRad = 1;// [Zellen] Wirkradius des LCM
```

Abbildung 4.1: Neue Variablen der Klasse Simulator

Die wichtigsten neuen Variablen befinden sich in der Simulator-Klasse.
Der erste Block hat keinen funktionalen Einfluss auf die Cell-List. Es handelt sich dabei um Hilfs-Variablen für die folgende Testphase.

- **String *outDir*** enthält den Pfad des Output-Verzeichnisses. Die neue Funktion zum Messen der Simulationszeit benötigt diesen
- **String *specOut*** enthält das Ergebnis der Simulationszeit-Messung
- Die beiden Wahrheitswerte ***messung1*** und ***messung2*** dienen zur Kontrolle der Messfunktion

Der zweite Block enthält die Variablen der Cell-List.

- **LCModel *linkedCell*** ist die Cell-List selbst
- Der **Wahrheitswert *triggerLC*** dient zur Deaktivierung und Aktivierung. *true* bedeutet dabei, dass die Cell-List genutzt wird
- **double *cellSize*** ist eine Gleitkommazahl (sinnvollerweise $> 0,5[m]$), die die Seitenlänge der einzelnen quadratischen Zellen in [m] enthält
- **int *effectiveRad*** ist eine Ganzzahl (positiv). Sie legt fest, um wie viele Zellen sich das effektive Gebiet in alle Richtungen, bezogen auf die Referenz-Zelle, ausdehnt. Der Wert 0 beschränkt die Betrachtung auf die Referenz-Zelle. 1 bedeuet, dass ein Gebiet von $3x3$ Zellen entsteht usw.

```
private int[] gridPos = { -1, -1 }; // Position im LCM (X,Y)
```

Abbildung 4.2: Neue Variable der Klasse Abstract Pedetrian

Eine weitere neue Variable bekommt der Fußgänger. Sie dient zur Identifizierung eines Zellenwechsels in der Cell-List. Es handelt sich dabei um ein ganzzahliges Feld, in dem die Position des Fußgängers im Cell-List-Gitter gespeichert ist.

4.3.2 Funktionen

Erweiterte Funktionen:

List<IPedestrian> getListAllNeis() (Klasse Simulator):
Funktion zur Bestimmung der Interaktions-Partner im Simulator. Wird in jedem Zeitschritt für jeden Fußgänger aufgerufen.

Pseudo-Code

Falls die Cell-List aktiviert ist
    Rückgabe einer von der Cell-List berechneten Liste der effektiven Nachbarn
Sonst
    Rückgabe der Liste aller im Modell befindlichen Fußgänger

void updatePedestrians (Klasse Simulator):
Methode, die zuerst die neue Position und Geschwindigkeit aller Fußgänger bestimmt. Danach folgt eine Ziel-Erreicht?-Abfrage und angekommene Fußgänger werden entfernt. Bei den anderen wird nun auch in der Cell-List die Position aktualisiert. Auf dieser Grundlage wird zum Schluss eine neue Fußgängerliste für den nächsten Zeitschritt erstellt. Auch die Zeitmessung für den folgenden Versuch befindet sich in dieser Methode.
Wird am Ende jedes Zeitschritts aufgerufen.

Pseudo-Code

Für alle Fußgänger auf der Fußgängerliste
    Berechne Position und Geschwindigkeit von Fußgänger
    Falls Fußgänger Messpunkt1 erreicht hat
        Schreibe Zeit in Ausgabe
        Beende Messung1
    Falls Fußgänger Messpunkt2 erreicht hat
        Schreibe Zeit in Ausgabe
        Beende Messung2
    Falls die Cell-List aktiviert ist
        Falls Fußgänger finales Ziel erreicht hat
            Entferne Fußgänger aus Cell-List
        Sonst
            Aktualisiere Position von Fußgänger in der Cell-List
            Füge Fußgänger der Liste für den nächsten Zeitschritt hinzu
    Sonst
        Falls Fußgänger finales Ziel erreicht hat
            do nothing
        Sonst
            Füge Fußgänger der Liste für den nächsten Zeitschritt hinzu

void generatePedestrian (Klasse PointSource):
Methode, die innerhalb einer Quelle neue Fußgänger erstellt. Diese werden sofort in der Cell-List angemeldet.
Der Aufruf erfolgt innerhalb des Quellen-Updates zu Beginn jedes Zeitschritts, falls die Quelle freigegeben ist.

Pseudo-Code

```
Für alle freien Generierungspunkte innerhalb einer Quelle
    Erzeuge Fußgänger
    Füge Fußgänger der Fußgängerliste hinzu
    Falls die Cell-List aktiviert ist
        Melde Fußgänger bei der Cell-List an
```

Neue Funktion:

void specialOutput (Klasse Simulator):
Erzeugt zu Beginn des Simulationsvorgangs eine *.txt-Datei, die mit Ausgaben direkt aus dem Simulator beschrieben werden kann. Dient dazu, auf einfache Art und Weise einzelne Testwerte zu gewinnen, ohne die eigentliche Ausgabe verändern zu müssen.
Auf eine Code-Darstellung verzichte ich an dieser Stelle, da es sich um eine Standardmethode handelt.

4.4 Quellcode der Cell-List

In diesem Abschnitt werden die wichtigsten Methoden der Cell-List erklärt. Auf die Bedeutung der Variablen werde ich hier nicht eingehen. Diese sollte aus dem Code klar zu erkennen sein.

4.4.1 LCModel

void buildLC():
Methode, die das zweidimensionale Feld der Cell-List berechnet und aufbaut. Der Aufruf erfolgt im Konstruktor des Simulators und damit logischerweise im Preprocessing.

Pseudo-Code

```
Berechnung der Spaltenanzahl der Cell-List (Ausdehnung_x/Zellengroesse aufgerundet)
Berechnung der Zeilenanzahl der Cell-List (Ausdehnung_y/Zellengroesse aufgerundet)
    /*Die Cell-List ist damit um bis zu einer Zellengröße größer als das zugrundeliegende Szenario*/
Konstruktor der Klasse LCList mit den berechneten Werten aufrufen
```

void updatePedPos(IPedestrian *ped*):
Die Methode dient zur Bestimmung der neuen Position des Fußgängers *ped* innerhalb der Cell-List. Dabei wird die im Fußgänger gespeicherte Gitterposition mit seiner geometrisch berechneten Gitterposition verglichen. Stimmen diese nicht überein, wird der Fußgänger in der Cell-List bewegt.
Sie wird nach dem Bewegungsvorgang in der oben schon beschriebenen Methode **void updatePedestrians** (Klasse Simulator) für jeden Fußgänger aufgerufen, der sein Ziel noch nicht erreicht hat.

Pseudo-Code

Falls errechnete Gitterposition von *ped* nicht mit gespeicherter übereinstimmt
    Entferne ped aus der Zelle mit der agespeicherten Position
    Füge *ped* der Zelle mit der errechneten Position hinzu
    Speichere die errechnete position in *ped*
Sonst
    do nothing

List<IPedestrian> getNeighbours(IPedestrian *ped*):
Diese Funktion stellt die Liste der effektiven Nachbarn für den Fußgänger *ped* zusammen. Das ist die zentrale und wichtigste Funktionalität der Cell-List
Im Simulator wird sie vor der Berechnung des social-force-Anteils der Fußgänger-Interaktionen aufgerufen.

Pseudo-Code

Hole Liste der Nachbarzellen von *ped*
Erzeuge Liste der effektiven Nachbarn
Für alle Zellen aus der Liste der Nachbarzellen
    Hole Liste der „Bewohner“
    Füge „Bewohner“ der Liste effektiver Nachbarn hinzu
Rückgabe der Liste effektiver Nachbarn

4.4.2 LCList

void setEffectiveNeighbours():
Diese Methode berechnet mit Hilfe des gegebenen *effektiven Radius* für jede Zelle die effektiven Nachbarzellen. Dabei werden alle Zellen nacheinander durchgegangen. Für jede Zelle berechnet sich nach dem bounding-box-Prinzip aus dem effektiven Radius eine Startzelle (links oben) und eine Stopzelle (rechts unten). Diese beiden schließen den effektiven Bereich ein, der dann als Liste von Zellen gespeichert wird.
Die Methode wird im Preprocessing nach der Erstellung des Cell-List-Gitters aufgerufen. Wie schon erwähnt, werde ich diesen Schritt wohl ersetzen, so dass nur noch für die Zellen, die auch wirklich betreten werden, eine Nachbarliste angelegt wird.

Pseudo-Code

```
Für alle Zeilen (y) der Cell-List
    Für alle Spalten (x) der Cell-List
        Betrachte Zelle[x|y]
        Berechne Startzelle aus Position von Zelle[x|y] − effektivem Radius
        Kontrolle der Startzelle auf zulässige Position
        Berechne Stopzelle aus Position von Zelle[x|y] + effektivem Radius
        Kontrolle der Stopzelle auf zulässige Position
        Für alle Zeilen (y_bb) der bounding-box
            Für alle Spalten (x_bb) der bounding-box
                Betrachte bounding-box-Zelle[x_bb|y_bb]
                Füge bounding-box-Zelle[x_bb|y_bb] als Nachbarn zu Zelle[x|y] hinzu
```

4.4.3 LCElement

void removePed(IPedestrian *ped*): Methode, die Fußgänger komplett aus der Cell-List löscht. Dazu wird erst die Zelle ermittelt, die der zu entfernende Fußgänger *ped* gerade „bewohnt“. Die „Bewohner“-Liste dieser Zelle wird dann nach *ped* durchsucht. Zum Schluss werden alle Fußgänger außer *ped* einer neuen Liste hinzugefügt. Diese ist die neue „Bewohner“-Liste ohne *ped*.
Aufgerufen wird die Methode während der Simulation, z.B. beim Zellenwechsel eines Fußgängers oder falls dieser sein Ziel erreicht hat.

Pseudo-Code

```
Erzeuge neue Liste
Für alle Fußgänger der „Bewohner“-Liste
    Falls „Bewohner“ nicht als ped identifiziert wird
        Füge „Bewohner“ der neuen Liste hinzu
Setze neue Liste als „Bewohner“-Liste
```

Kapitel 5

Praktischer Versuch

Die Theorie legt einen Erfolg der „Cell-List“ zur Reduzierung der Rechenzeit nahe. In welcher Größenordnung sich diese Reduzierung auswirkt, muss in Tests evaluiert werden.

Bei diesen Versuchen wurde die Rechenzeit des Computers für ein geeignetes Test-Szenario bei variabler Fußgängeranzahl bestimmt. Trägt man nun jeweils die Rechenzeit gegen die Anzahl der berechneten Fußgänger auf, lässt sich eine Komplexitätskurve abschätzen.

Im folgenden Kapitel wird eine aussagekräftige Versuchsanordnung beschrieben und das erzielte Ergebnis vorgestellt.

5.1 Aufbau

5.1.1 Scenario

Das Scenario für diesen Versuch ist ein langer Gang ohne Hindernisse. Die Fußgänger starten links an der Quelle und bewegen sich zum Ziel nach rechts. Der Gang ist insgesamt 83[m] lang und 6[m] breit.

Die lange Seite teilt sich in drei Zonen auf:

- Eine 30[m] lange **Start-Zone** vom Beginn des Gangs zum ersten Messpunkt (M1). Diese Zone ist notwendig, da zu Messbeginn (M1) alle Fußgänger generiert sein müssen, um ein sinnvolles Messergebnis zu erzielen (siehe 5.1.3). Die Quelle kann aber im Moment immer nur 6 Fußgänger gleichzeitig generieren. Danach muss sie warten, bis der Quellbereich wieder frei ist, bevor sie fortfahren kann. Das führt dazu, dass die „älteren“ Fußgänger schon eine gewisse Strecke (etwa 2,5[m] pro 6er-Gruppe) zurückgelegt haben. Bei der von mir gewählten Maximalanzahl von 60 Fußgängern (siehe Kapitel 5.1.2) garantiert ein Vorlauf von 30[m], dass die Messung korrekt ausgeführt werden kann

- Eine 50m lange **Messzone** von Messpunkt M1 zu Messpunkt M2. Der Wert hat sich in mehreren vorangegangenen Versuchen als sinnvoll erwiesen
- Eine 3m lange **Zielzone** von M2 bis zum Ende des Gangs. Diese Zone ist für den Versuchsablauf nicht relevant

Die Breite ist so gewählt, dass die Fußgänger genug Platz zur Entwicklung einer stabilen Bewegungsformation haben.

Abb. 5.1 stammt aus einem Visualisierungs-Tool von Siemens („CrowdVis“). Es stellt das Szenario kurz vor Simulationsbeginn dar. Die Bemaßung ist extra hinzugefügt.

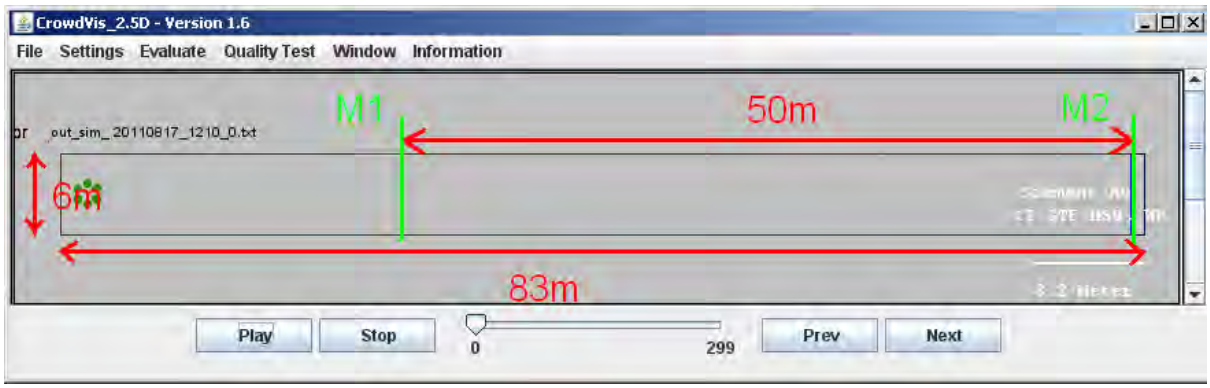


Abbildung 5.1: Geometrie des Test-Szenarios

5.1.2 Einstellungen

Voreinstellungen im GUI-Fenster (siehe Kapitel 2.2.2):

- Szenario-Geometrie: Der in Abschnitt 5.1.1 gezeigte Gang
- Zeitschritt: 50[ms] (Standardwert)
- Anzahl Fußgänger: **Versuchsvariable**. Wert zwischen 6 und 60 in 6er-Schritten. Ergibt 10 unterschiedliche Messungen
- Maximale Zeit: 120[s] frei gewählt. Für 83[m] braucht ein Fußgänger bei Maximal-Geschwindigkeit 1,3[m/s] etwa 64[s]. In 120[s] sollten selbst langsame oder verirrte Fußgänger die Strecke bewältigen
- Die restlichen Einstellungen sind entweder unveränderlich oder nicht relevant für den Versuch

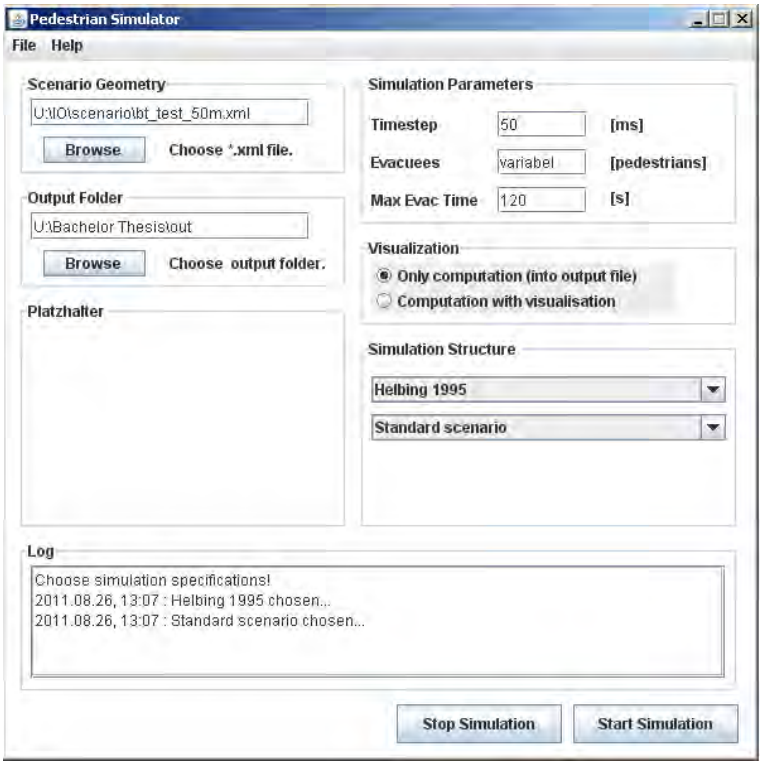


Abbildung 5.2: GUI-Fenster mit Versuchs-Einstellungen

Voreinstellungen bei Versuchen mit aktivierter Cell-List (siehe Kapitel 4.3):

- Cell-List mit Hilfe von Trigger aktiviert
- Zellen-Seitenlänge: 1.0[m] (Breite = Höhe). Klein gewählt, um den „Sichtradius“ der Fußgänger klein zu halten. Damit wird auch bei vergleichsweise geringen Fußgängerzahlen ein großer Unterschied zwischen der Gesamtanzahl (n) aller Fußgänger und der sichtbaren Anzahl (k) eines einzelnen Fußgängers geschaffen. Dieser Unterschied ist Voraussetzung für die Wirksamkeit der Cell-List.
- Effektiver Radius: 2[Zellen]. Die „Sichtweite“ beträgt damit je nach Position der beteiligten Fußgänger max. 4,20[m]. Das entspricht weder der Realität, noch der Reichweite der Potenziale, ist aber, wie im Unterpunkt davor beschrieben, bei dieser Versuchsanordnung sinnvoll

```
// Implementierung des Linked-Cell-Models. Alle Einstellungen sind an dieser
// Stelle im Code vorzunehmen!
private LCModel linkedCell = null;
private boolean triggerLC = true;// Schalter für LCM (true=aktiviert)
private final double cellSize = 1.0;// Seitenlänge der Zellen
private final int effectiveRad = 2;// [Zellen] Wirkradius des LCM
```

Abbildung 5.3: Source-Code aus Simulator-Klasse mit Versuchs-Einstellungen

5.1.3 Ablauf

Um verwertbare Ergebnisse erzielen zu können, muss für eine Messserie, bestehend aus fünf Messreihen (die wiederum aus jeweils 10 Einzelmessungen bestehen) immer derselbe Computer eingesetzt werden. Will man die absoluten Zahlenwerte mehrerer solcher Messserien vergleichen, müssen diese folglich auch auf dem selben Computer entstanden sein. In der hier betrachteten Versuchsreihe wurde jeweils eine Messserie mit und ohne Cell-List durchgeführt, leider jedoch nicht am selben Computer.

Zusätzlich ist zu beachten, dass sich während der Simulation keine Hintergrundprozesse aktivieren dürfen, da zusätzliche Rechenlast das Ergbnis der Messung verfälscht. Um den korrekten Ablauf jeder Einzelmessung garantieren zu können, muss nach Ende des Simulationvorgangs die Output-Visualisierung kontrolliert werden. Einzelne, auf diese Weise als fehlerhaft identifizierte Simulationen, können dann wiederholt werden.

Das Ergebnis des Einzel-Experiments, also die Rechenzeit zwischen M1 und M2 (siehe Abb. 5.1), berechnet sich aus dem jeweiligen Simulator-Output-File. In diesem File werden zwei Uhrzeiten sekundengenau gespeichert. Die erste Uhrzeit ist die Zeit, zu der ein Fußgänger M1 erreicht. Die zweite Zeit ist folglich diejenige, zu der ein Fußgänger M2 erreicht. Die Rechenzeit für 50[m] Strecke bestimmt sich aus der Differenz der beiden Werte. Die extra dafür eingefügten Output-Funktionen sind so gestaltet, dass sie die Rechenzeit im messbaren Bereich nicht beeinflussen.

5.2 Auswertung

5.2.1 Ohne Cell-List

Die Mittelung der ersten 5 Messreihen (siehe Zusatztabelle B.1 bis B.5 in Anhang B) liefert die Fußgänger-Rechenzeit-Abhängigkeit ohne Cell-List für 6 bis 60 Fußgänger. Gemittelt wurden dafür die fünf jeweils zueinander gehörenden und gleich gewichteten Zeit-Messwerte (ganzzahlig in Sekunden). Das Ergebnis ist wieder zur Ganzzahl gerundet. Es ergibt sich in tabellarischer Form folgendes Ergebnis:

Fußgänger [-]	Rechenzeit [s]
6	3
12	9
18	18
24	30
30	47
36	63
42	83
48	99
54	127
60	142

Tabelle 5.1: Mittelung der Messungen ohne Cell-List

Die grafische Auswertung in Excel (Microsoft Office) sieht folgendermaßen aus:

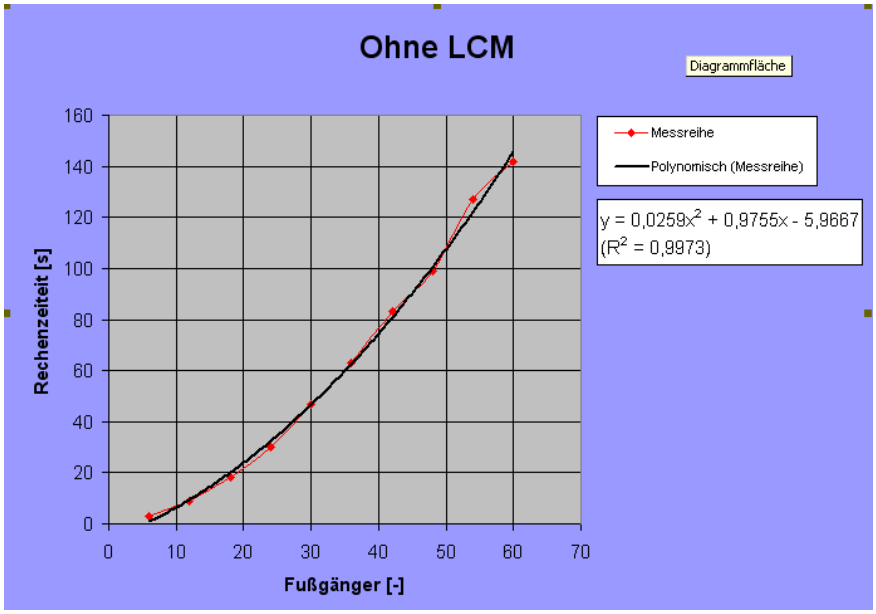


Abbildung 5.4: Grafische Auswertung ohne Cell-List

Wie man gut erkennen kann, verhält sich die Kurve der Messwerte (**Messreihe**) quadratisch. Man beachte, dass der Maßstab der beiden Achsen zur besseren Darstellung verschieden ist. Eigentlich wäre die Kurve wesentlich steiler.

Gleichung 5.1 zeigt das von Excel berechnete Näherungs-Polynom zweiter Ordnung. Das Bestimmtheitsmaß im betrachteten Bereich beträgt 99,7%. Die Näherung ist damit sehr gut.

$$y = 0,0259x^2 + 0,9755x - 5,9667 \tag{5.1}$$

Obwohl der quadratische Term mit 0,0259 einen geringen Faktor besitzt, wird er schon relativ früh (etwa bei 40 Fußgängern) dominant. Folglich würde bei einer großen Erhöhung der Fußgängeranzahl, etwa auf 1000, die Rechenzeit immens hoch werden. Eine Extrapolation ergibt in diesem Fall etwa 7,5 Stunden, was aber nur eine grobe Schätzung ist, da wir uns weit außerhalb des zulässigen Extrapolationsradius befinden.

Das Ergebnis ist in dieser Form aber nicht überraschend. Es verdeutlicht lediglich das bekannte Problem, das mit Hilfe der Cell-List gelöst werden soll.

5.2.2 Mit Cell-List

Die Mittelung der zweiten 5 Messreihen (siehe Zusatztabelle B.6 bis B.10 in Anhang B) ergibt die Fußgänger-Rechenzeit-Abhängigkeit mit aktivierter Cell-List. Berechnet werden die Mittelwerte analog zu 5.2.1. In tabellarischer Form geribt sich folgendes Ergebnis:

Fußgänger [-]	Rechenzeit [s]
6	2
12	4
18	5
24	8
30	10
36	12
42	17
48	18
54	21
60	24

Tabelle 5.2: Mittelung der Messungen mit Cell-List

Die grafische Auswertung in Excel (Microsoft Office) sieht folgendermaßen aus:

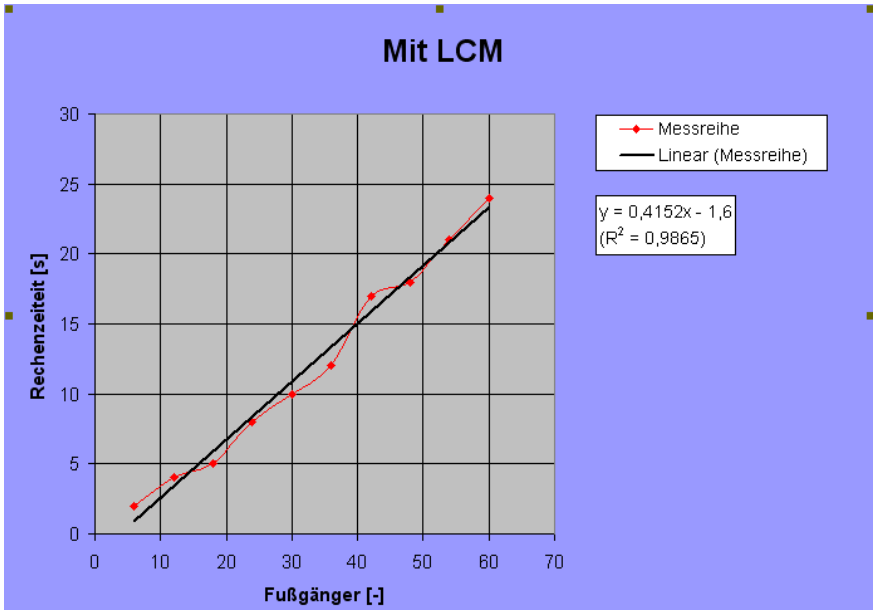


Abbildung 5.5: Grafische Auswertung mit Cell-List

Die Messwert-Kurve (**Messreihe**) verhält sich jetzt eindeutig linear. Auch hier ist wieder der unterschiedliche Maßstab der beiden Achsen zu beachten, aber umgekehrt als im vorangegangenen Diagramm. Die Kurve ist eigentlich flacher.

Gleichung 5.2 zeigt wieder das von Excel berechnete Näherungs-Polynom. Dieses ist jetzt aber nur noch erster Ordnung und stellt somit eine Gerade dar. Die Bestimmtheit ist mit 98,7% ähnlich gut wie bei den Messungen ohne Cell-List.

$$y = 0,4152x - 1,6 \tag{5.2}$$

Die y-Werte wachsen damit deutlich geringer als vorher. Das Extrapolations-Beispiel (1000 Fußgänger) ergibt im linearen Fall nur eine Rechenzeiteit von knapp 6 Minuten. Es dient aber auch hier nur zur Veranschaulichung ohne mathematische Beweiskraft.

Damit ist der Nutzen der Cell-List zur Reduzierung der Rechenzeit bewiesen. Die theoretischen Überlegungen waren folglich korrekt.

5.2.3 Ergebnis - Vergleich

Die Ergebnisse lassen sich nun auf qualitativer und quantitativer Ebene vergleichen. Zweiteres ist zwar, wie in 5.1.3 erklärt, eigentlich nicht sinnvoll, erlaubt aber trotzdem einige interessante Schlussfolgerungen.

Qualitativ

Zum Ergebnis des qualitativen Vergleichs gibt es nicht viel zu sagen. Aus einer Parabel ist, wie erwartet, eine Gerade geworden. Die Komplexität des Simulations-Algorithmus hat sich damit von quadratischer auf lineare Ordnung reduziert.

Quantitativ

In Grafik 5.6 auf Seite 36 sind nun die beiden Rechenzeit-Kurven aus 5.2.1 und 5.2.2 in einem Diagramm zusammengefasst. Der Maßstab auf beiden Achsen ist diemal gleich gewählt.

Was man bei sehr genauem Hinschauen erkennt, ist, dass sich die Rechenzeit schon bei 6 Fußgängern unterscheidet. Der für die Simulation mit Cell-List eingesetzte Computer verfügt somit über mehr Rechenleistung.
Im weiteren Verlauf entfernen sich die beiden Kurven sehr schnell voneinander. Die Rechenzeit ohne Cell-List steigt rapide an und übersteigt schnell den **Echtzeit-Grenzwert($T_{grenz,echt}$)**, der bisher noch nicht betrachtet wurde. Dabei handelt es sich um einen Rechenzeit-Wert in [s], der nicht überschritten werden darf, wenn man in Echtzeit simulieren möchte, was auch eines unserer Ziele ist. Bestimmen lässt sich dieser Wert im betrachteten Testszenario als die Zeit, die ein Fußgänger bei Maximalgeschwindigkeit (v_{max}) für die Messstrecke (l_{mess}) braucht.

$$T_{grenz,echt} = l_{mess}/v_{max} = 50[m]/1,3[m/s] = 38,5[s] \tag{5.3}$$

Nun kann man mit Hilfe von Grafik 5.6 kontollieren, wann diese 38,5[s] erreicht werden. Bei dem Test ohne Cell-List ist das bei etwa 27 Fußgängern der Fall. Die andere Kurve erreicht den Wert im betrachteten Bereich garnicht. Das bedeutet, im Testszenario ist mit aktivierter Cell-List Echtzeit-Simulation möglich.

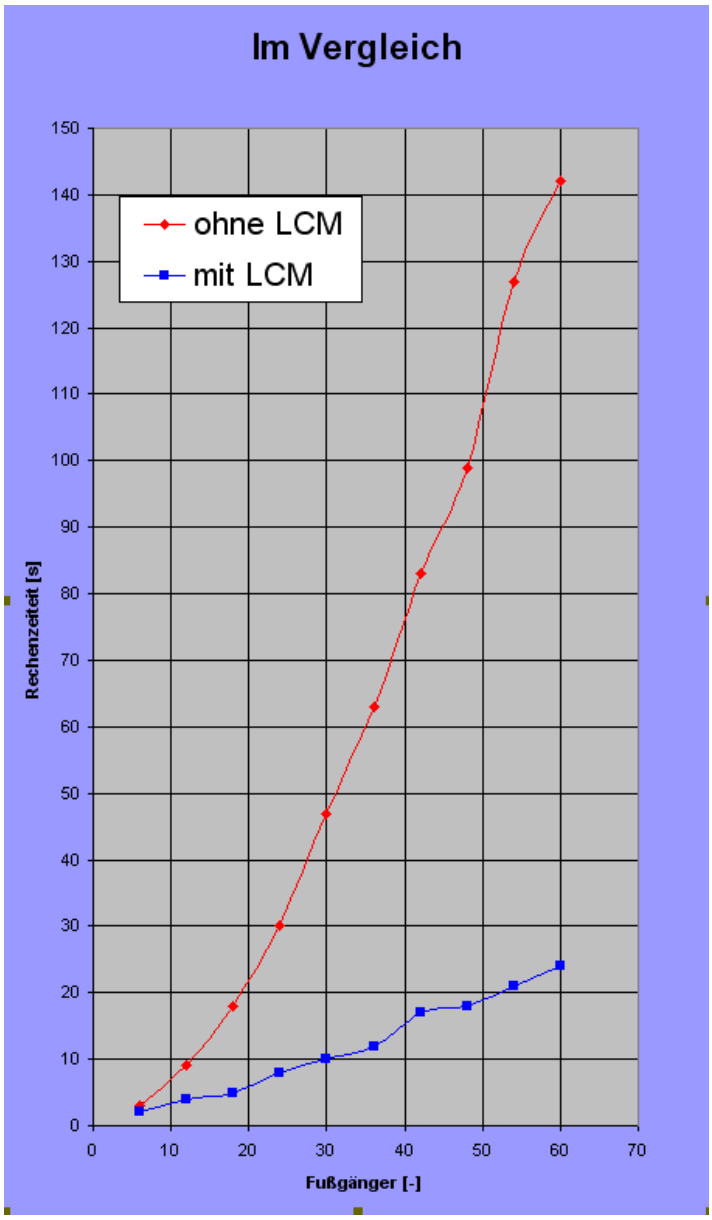


Abbildung 5.6: Grafische Auswertung mit und ohne Cell-List

Kapitel 6

Zukünftige Entwicklungen

In dieser Arbeit ging es haupsächlich darum, zu evaluieren, ob sich die Implementierung eines kompletten Linked-Cell-Modells in das Gesamt-Projekt lohnt. Und wie erwartet hat sich die Cell-List (bzw. das Linked-Cell-Model) als effektiver Weg zur Reduzierung der Rechenzeit erwiesen. Weiterer Entwicklungsarbeit steht somit nichts mehr im Wege.

6.1 Weiterführende Arbeit an der Cell-List

Die bisher in dieser Arbeit geschilderte Implementierung der Cell-List ist nur für Testzwecke geeignet. Bei realen Simulationen sieht die Sache ganz anders aus. Um dort ähnliche Rechenzeit-Ergebnisse zu erzielen, ohne die Qualität des Ergebnises negativ zu beeinflussen, ist noch einiges an Entwicklungsarbeit notwendig.

Die nächsten wichtigen Entwicklungs-Schritte sind:

- Integrieren der Einstellungen der Cell-List in die GUI. Es handelt sich dabei um die bisher direkt im Quellcode vorzunehmenden Einstellungen (Aktivierung, Zellengröße, Effektiver Radius)

- Testen der optimalen Einstellungen. Das schließt unter anderem das Anpassen von Radius und Zellengröße an die Reichweite der Potenziale des Helbing-Modells ein

- Veränderung der Form des effektiven Bereichs. Damit sollen Blickfeld, toter Winkel im Rücken und andere sensorische Einflüsse des Fußgängers in das Modell integriert werden. Das erfordert unter anderem Anpassungen am Nachbar-Such-Algorithmus der Cell-List

- Integrieren der Hindernisse in die Cell-List in einer eigenen Struktur.

6.2 Weiterführende Arbeit am Simulator

Natürlich wird nicht nur an der Cell-List weitergearbeitet. Das ganze Projekt befindet sich noch am Anfang der Entwicklung.

Die nächsten größeren Teil-Projekte, die anstehen, sind:

- Lösung der restlichen Probleme des Helbing-Modells
- Verbesserung der Rechenzeit um mehrere Größenordnungen durch Optimierung von Berechnungen und Implementierung von weiteren Modellen, analog zur Cell-List
- Erstellen einer eigenen Visualisierung mit Auswert-Funktionalität
- Integrieren beweglicher Hindernisse (Tore, Autos, etc.)

Anhang A

Source Code LCM

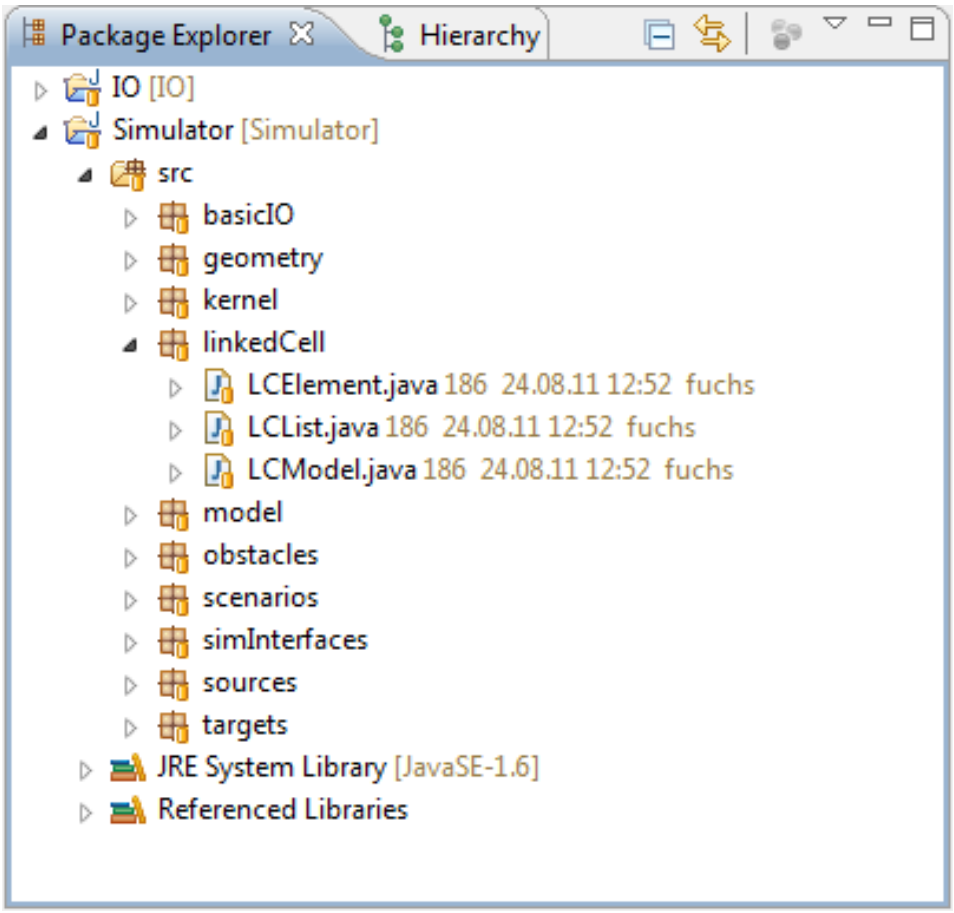


Abbildung A.1: linkedCell-Paket im package explorer

Kompletter Source-Code der Cell-List-Implementierung (Formatierung teilweise verändert).

Enthält die drei Klassen LCElement, LCList und LCModel, die im linkedCell-Paket zusammengefasst sind (siehe Abb.A.1).

A.1 Klasse LCElement

```
package linkedCell;

import java.util.ArrayList;
import java.util.List;

import simInterfaces.ILinkedCell;
import simInterfaces.IPedestrian;

/**
 * Einzelnes Zellenelement für die LC-Struktur. Enthält eine Liste
 * an Fußgängern, die sich aktuell in der Zelle befinden, sowie
 * die Liste der wirkungseffektiven Nachbarn.
 */
public class LCElement implements ILinkedCell {

    // ATTRIBUTES
    // ------------------------------------------------------------

    // Position im LCM Gitter. (x,y)
    private int[] gridPos = { 0, 0 };
    // Benachbarte Zellen. Anzahl abhängig von Wirkradius.
    //Enthält für Berechnung auch immer sich selbst.
    private List<LCElement> buddyList;
    // Liste an Fußgängern in der Zelle
    private List<IPedestrian> pedList;

    // CONSTRUCTOR
    // ------------------------------------------------------------

    /**
     * Konstruktor für einzelnes LC-Zellenelement.
     * @param gridPosition
     *            Position im Gitter (x,y)
     */
    public LCElement(int[] gridPosition) {
        buddyList = new ArrayList<LCElement>();
        pedList = new ArrayList<IPedestrian>();
        this.gridPos = gridPosition;
    }

    // ----------
```

```
// METHODS
// ------------------------------------------------------------------

/**
 * Gibt die Position der Zelle im LCM-Gitter zurück.
 *
 * @return int[ ]: {x,y}
 */
public int[] getGridPos() {
    return gridPos;
}

// ----------

/**
 * Fügt eine benachbarte Zelle hinzu.
 */
public void addNeighbour(LCElement buddy) {
    buddyList.add(buddy);
}

// ----------

/**
 * Gibt eine Liste benachbarter Zellen zurück.
 */
public List<LCElement> getNeighbours() {
    return buddyList;
}

// ----------

/**
 * Fügt der Zelle einen Fußgänger hinzu.
 */
public void addPed(IPedestrian ped) {
    pedList.add(ped);
}

// ----------
```

```
    /**
     * Entfernt einen Fußgänger aus der betrachteten Zelle.
     * Erzeugt dazu eine neue pedList, die den zu entfernenden
     * Fußgänger nicht mehr enthält.
     */
    public void removePed(IPedestrian ped) {
        List<IPedestrian> listTemp;
        listTemp = new ArrayList<IPedestrian>();

        for (IPedestrian tempPed : pedList) {
            if (tempPed != ped)
                listTemp.add(tempPed);
        }

        pedList.clear();
        pedList = listTemp;
    }

    // ----------

    /**
     * Gibt eine Liste von Fußgängern zurück, die sich aktuell
     * in der Zelle aufhalten.
     */
    public List<IPedestrian> getResidents() {
        return pedList;
    }
}
```

Ende der Klasse LCElement

A.2 Klasse: LCList

```
package linkedCell;

import simInterfaces.ILinkedCell;

/**
 * Klasse, die die Zellenstruktur-Struktur darstellt. Beinhaltet
 * das Gitter, das später die einzelnen Zellen enthält.
 * Zur späteren Erweiterung als eigene Klasse konstruiert.
 */
public class LCList implements ILinkedCell {

    // ATTRIBUTES
    // ------------------------------------------------------------

    // Gitterstruktur !!![reihen][spalten] (y,x)
    private LCElement[][] grid;
    private int rows; // Anzahl Reihen (y)
    private int columns; // Anzahl Spalten (x)
    private int effectiveRad;// [Zellen] Wirkradius des LCM

    // CONSTRUCTOR
    // ------------------------------------------------------------

    /**
     * Konstruktor für LC-Listenstruktur.
     * @param cellStruct
     *     : Ausmaße des Szenarios in ganzen Zellen int[ ]={x,y}.
     * @param effectiveRad
     *     :[Zellen] Wirkradius des LCM
     */
    public LCList(int[] cellStruct, int effectiveRad) {

        rows = cellStruct[1];// y
        columns = cellStruct[0];// x
        grid = new LCElement[rows][columns];
        this.effectiveRad = effectiveRad;

        buildListElements();
        setEffectiveNeighbours();
    }

    // ------------
```

```
// METHODS
// ------------------------------------------------------------------

/**
 * Erzeugt die einzelnen Zellen-Elemente in der Gitterstruktur.
 * Konstruktor-interne Methode!
 */
private void buildListElements() {
    LCElement tempElement;
    int i = 0;
    int j = 0;

    for (i = 1; i <= rows; i++) {// y
        for (j = 1; j <= columns; j++) {// x
            tempElement = new LCElement(new int[] {j-1,i-1});
            grid[i - 1][j - 1] = tempElement;// !!!grid:y,x
        }
    }
}

// ----------

/**
 * Legt im Preprocessing die effektiven Nachbarzellen jeder LCM
 * Zelle fest. Wird im Preprocessing ausgeführt, um Rechenzeit
 * zur Laufzeit zu sparen.
 */
private void setEffectiveNeighbours() {
    int i = 0;
    int j = 0;
    int k = 0;
    int l = 0;
    int[] pos = { 0, 0 };

    // Start links oben im effektiven Bereich, Stop rechts
    // unten (Bounding-Box-Struktur).
    int[] start = { -1, -1 };
    int[] stop = { -1, -1 };

    LCElement tempElement;
    LCElement tempNeighbour;

    //Fortsetzung der Funktion auf der nächsten Seite ->
```

```
        // Schleife über alle Zellen
        for (i = 0; i < rows; i++) {
            for (j = 0; j < columns; j++) {

                tempElement = grid[i][j];// Aktuell betrachtete Zelle
                pos = tempElement.getGridPos();// Gitterposition der
                aktuellen Zelle

                // Ermitteln der Gitterposition von Start und Stop
                start[0] = pos[0] - effectiveRad;// x
                start[1] = pos[1] - effectiveRad;// y
                if (start[0] < 0)
                    start[0] = 0;
                if (start[1] < 0)
                    start[1] = 0;
                stop[0] = pos[0] + effectiveRad;// x
                stop[1] = pos[1] + effectiveRad;// y
                if (stop[0] > columns - 1)
                    stop[0] = columns - 1;
                if (stop[1] > rows - 1)
                    stop[1] = rows - 1;

                // Schleife über die ermittelten Nachbarzellen(und sich
                // selbst) und Hinzufügen als Nachbarn.
                for (k = start[1]; k <= stop[1]; k++) {
                    for (l = start[0]; l <= stop[0]; l++) {
                        tempNeighbour = grid[k][l];
                        tempElement.addNeighbour(tempNeighbour);
                    }
                }
            }
        }
    }

    // ----------

    /**
     * Gibt das Zellen-Element an der gewählten Position im Gitter
     * (x, y) zurück.
     */
    public LCElement getElement(int posX, int posY) {
        return grid[posY][posX];
    }
}
```

Ende der Klasse LCList

A.3 Klasse: LCModel

```
package linkedCell;

import geometry.Point2D;

import java.util.ArrayList;
import java.util.List;

import kernel.AbstractPedestrian;

import simInterfaces.ILinkedCell;
import simInterfaces.IPedestrian;

/**
 * Rahmenklasse für das Linked-Cell-Model zur Reduzierung der
 * Simulationszeit.
 * Durch erweitertes Preprocessing wird eine Gitterstruktur
 * (LC-Layer) auf das zu simulierende Szenario aufgelegt. Mit
 * Hilfe dieser Struktur lassen sich die wirksamen Nachbarn eines
 * Fußgängers ermitteln. Dadurch sinkt die Komplexität.
 */
public class LCModel implements ILinkedCell {

    // ATTRIBUTES
    // ------------------------------------------------------------

    // [m] maximale Ausdehnung(x,y) des Szenarios
    private double[] maxExtent = { 0.00, 0.00 };
    // Gitterstuktur für die einzelnen Elemente.
    private LCList cellList;
    // [m] Ausmaße einer Zelle (Breite=Höhe).
    private final double cellSize;
    // Anzahl nötiger Zellen (hor und vert).
    private int[] cellStruct = { 0, 0 };
    private final int effectiveRad;// [Zellen] Wirkradius des LCM

    // ----------
```

```
// CONSTRUCTOR
// ------------------------------------------------------------

/**
 * Konstruktor für LCM. Vor Verwendung buildLC() ausführen!
 * @param scenarioExtent
 *            : Ausmaße des Szenarios in [m] (x,y)
 * @param cellSize
 *            : Länge(=Breite) einer Zelle in [m]
 * @param effRad
 *            : Wirkradius des LCM in [Zellen]
 */
public LCModel(double[] scenarioExtent, double cellSize,
                int effRad) {
    this.cellSize = cellSize;
    effectiveRad = effRad;
    this.maxExtent[0] = scenarioExtent[0];
    this.maxExtent[1] = scenarioExtent[1];
}

// METHODS
// ------------------------------------------------------------

/**
 * Baut die Gitterstruktur des LCM auf. Für möglich Erweiterung
 * des Modells in eigene Funktion ausgelagert.
 */
public void buildLC() {
    cellStruct[0] = (int) (Math.ceil(maxExtent[0] / cellSize));
    cellStruct[1] = (int) (Math.ceil(maxExtent[1] / cellSize));
    cellList = new LCList(cellStruct, effectiveRad);
}

// ----------

/**
 * Fügt dem LCM einen neu erzeugten Fußgänger hinzu. Nur bei
 * der Fußgängergenerierung aufrufen!
 */
public void addNewPed(AbstractPedestrian ped) {
    int i[] = getPedCell(ped);
    ped.setLCPos(i[0], i[1]);
    addPed(ped, i[0], i[1]);
}

// ----------
```

```
/**
 * Fügt dem LCM einen bereits existierenden Fußgänger hinzu.
 * Aufruf nur beim Update der Position!
 * @param posX
 *          : Zelle, der der Fußgänger hinzugefügt werden soll.
 * @param posY
 *          : Zelle, der der Fußgänger hinzugefügt werden soll.
 */
public void addPed(IPedestrian ped, int posX, int posY) {
    LCElement temp;
    temp = cellList.getElement(posX, posY);
    temp.addPed(ped);
}

// ----------

/**
 * Entfernt einen Fußgänger aus einer Zelle des LCM. Aufruf nur
 * beim Update der Position!
 */
public void deletePed(IPedestrian ped) {
    LCElement temp;
    int[] cell = ped.getLCPos();

    temp = cellList.getElement(cell[0], cell[1]);
    temp.removePed(ped);
}

// ----------

/**
 * Neupositionierung eines bereits vorhandenen Fußgängers.
 * Aufruf nur beim Update der Position!
 */
public void updatePedPos(IPedestrian ped) {
    int[] newPos = getPedCell(ped);
    int[] oldPos = ped.getLCPos();

    if (newPos[0] != oldPos[0] || newPos[1] != oldPos[1]) {
        cellList.getElement(oldPos[0],oldPos[1]).removePed(ped);
        cellList.getElement(newPos[0],newPos[1]).addPed(ped);
        ped.setLCPos(newPos[0], newPos[1]);
    }
}

// ----------
```

```
/**
 * Zuordnung des Fußgängers zur korrekten Zelle über die
 * Szenariogeometrie.
 * @return int[ ]={x,y}
 */
public int[] getPedCell(IPedestrian ped) {
    int[] gridPos = { 0, 0 };
    Point2D pos = ped.getPosition();

    gridPos[0] = (int) (Math.ceil(pos.getX() / cellSize));
    gridPos[1] = (int) (Math.ceil(pos.getY() / cellSize));

    return gridPos;
}

// ----------

/**
 * Erzeugt eine Liste mit den wirksamen Nachbarn (sich selbst
 * eingeschlossen) des betrachteten Fußgängers und gibt diese
 * zurück.
 */
public List<IPedestrian> getNeighbours(IPedestrian ped) {
    LCElement pedEle;// Zelle des betrachteten Fußgängers.
    // Liste der Fußgänger in den Nachbarzellen.
    List<IPedestrian> neighList;
    List<LCElement> neighEleList;// Liste der Nachbarzellen.

    neighList = new ArrayList<IPedestrian>();
    pedEle = cellList.getElement(ped.getLCPos()[0],
                    ped.getLCPos()[1]);
    neighEleList = pedEle.getNeighbours();

    for (LCElement ele : neighEleList) {
        neighList.addAll(ele.getResidents());
    }

    return neighList;
}

// ----------
```

```
    /**
     * Überprüft, ob die Position innerhalb des erlaubten
     * Bereichs (LC-Gitter) ist.
     * @param lcPos
     *            : zu überprüfende Position (x,y)
     * @return true = im Bereich
     */
    public boolean validPos(int[] lcPos) {
        if (lcPos[0] < 0 || lcPos[1] < 0)
            return false;
        else if (lcPos[0] > cellStruct[0] - 1 ||
                        lcPos[1] > cellStruct[1] - 1)
            return false;
        else
            return true;
    }
}
```

Ende der Klasse LCModel

Anhang B

Zusätzliche Tabellen

Fußgänger [-]	Rechenzeit [s]
6	4
12	9
18	18
24	29
30	47
36	63
42	83
48	99
54	127
60	142

Tabelle B.1: Messung ohne Cell-List Nr.01

Fußgänger [-]	Rechenzeit [s]
6	3
12	9
18	18
24	30
30	48
36	64
42	84
48	100
54	128
60	142

Tabelle B.2: Messung ohne Cell-List Nr.02

Fußgänger [-]	Rechenzeit [s]
6	3
12	9
18	18
24	30
30	47
36	63
42	83
48	99
54	127
60	142

Tabelle B.3: Messung ohne Cell-List Nr.03

Fußgänger [-]	Rechenzeit [s]
6	4
12	10
18	18
24	30
30	47
36	63
42	82
48	99
54	127
60	142

Tabelle B.4: Messung ohne Cell-List Nr.04

Fußgänger [-]	Rechenzeit [s]
6	3
12	9
18	18
24	30
30	48
36	64
42	83
48	100
54	128
60	142

Tabelle B.5: Messung ohne Cell-List Nr.05

Fußgänger [-]	Rechenzeit [s]
6	2
12	3
18	4
24	8
30	9
36	12
42	17
48	19
54	21
60	24

Tabelle B.6: Messung mit Cell-List Nr.01

Fußgänger [-]	Rechenzeit [s]
6	2
12	4
18	4
24	8
30	10
36	12
42	17
48	19
54	22
60	24

Tabelle B.7: Messung mit Cell-List Nr.02

Fußgänger [-]	Rechenzeit [s]
6	2
12	4
18	5
24	8
30	10
36	12
42	17
48	18
54	21
60	24

Tabelle B.8: Messung mit Cell-List Nr.03

Fußgänger [-]	Rechenzeit [s]
6	2
12	4
18	5
24	7
30	10
36	12
42	17
48	18
54	21
60	23

Tabelle B.9: Messung mit Cell-List Nr.04

Fußgänger [-]	Rechenzeit [s]
6	3
12	4
18	6
24	8
30	10
36	13
42	18
48	19
54	22
60	24

Tabelle B.10: Messung mit Cell-List Nr.05

Die minimalen Zeitunterschiede innerhalb der Serien lassen sich leicht erklären. Das Rechenmodell enthält im Moment keine Zufallsentscheidungen. Da ich immer bei gleichen Bedingungen simuliert habe, sollte es theoretisch keine Unterschiede geben zwischen den einzelnen Messserien.
Die gibt es auch nicht. Es handelt sich dabei um Messfehler, die dadurch auftereten, dass ich nicht in Zehntelsekunden messe. Der Unterschied liegt daher auch immer im Bereich um $\pm 1[S]$.
Die Mittelung über fünf Messserien wäre bei genauerer Zeitmessung nicht notwendig, ist aber gut zur Kontrolle.

Anhang C

DVD

Auf der beigefügten DVD befindet sich folgender Inhalt:

- Quellcode (Java) des kompletten Simulators mit Cell-List-Implementierung (Stand: 30. August 2011)
- Versuchsdaten und Auswertungen
- Die komplette Arbeit, geschrieben in LaTeX (PDF)
- Der Vortrag zur Vorstellung der Arbeit (PDF), gehalten am 24.August 2011

Literaturverzeichnis

[1] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Berlin Heidelberg New York, Berlin, 2007.

[2] H. Wille and et al. Software Lab: Continuous Pedestrian Simulation, 2009.

[3] D. Helbing, P. Molnár, I. Farkas, and K. Bolay. Self-organizing pedestrian movement. *Environment and Planning B: Planning and Design*, 28:361–383, 2001.

[4] Lehrstuhl für Computation and Engineering. *Mitschrift: Bau- und Umweltinformatik*. TUM, München, 2009/2010.

[5] C. Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch*. Galileo Computing, 2010.