Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 29

Framework zur Fußgängersimulation

Stephan Herb

Studiengang:	Informatik
Prüfer:	Prof. Dr. Marc Alexander Schweitzer
Betreuer:	Dr. Stefan Zimmer
Beginn am:	10. September 2012
Beendet am:	12. März 2013
CR-Klassifikation:	G.3, I.6.8, J.2

Kurzfassung

In dieser Arbeit geht es um die praktische Umsetzung des F.A.S.T.-Modells als Framework zur Fußgängersimulation auf mobilen Tablet-Computern. Zusätzliche Komponenten sind in das Modell eingebaut worden, um die Ergebnisse noch realistischer zu gestalten. Dazu zählen unter anderem die Erweiterung des Simulationsgebiets um weitere Stockwerke, sowie die Simulation von Treppen. Eine Sensitivitätsanalyse soll die Möglichkeit bieten, die Ergebnisse noch besser auswerten zu können. Die Implementierung des Modells und der angesprochenen Erweiterungen wird ebenso detailliert beschrieben, wie die Ergebnisse von Testsimulationen des Frameworks. Neben Tests zu Durchführungszeiten und Speicherverbrauch, wurde auch eine Sensitivitätsanalyse durchgeführt und getestet, wo die Grenze für simulierbare Szenarien mit dem vorgelegten System liegt.

Inhaltsverzeichnis

1 Einleitung

Computersimulationen von Gebäude- oder Verkehrsmittelevakuierungen können einen großen Beitrag zur Verbesserungen bestehender oder neuer Fluchtpläne leisten. Denn Übungen mit echten Menschen vor Ort sind teuer, zeitaufwändig und bringen selbst bestimmte Gefahren für die beteiligten Personen mit sich. Neben vielen anderen Modellen, die solche Simulationen beschreiben, wurde das sogenannte „F.A.S.T.“-Modell entwickelt. Damit können recht unkompliziert Evakuierungsszenarien simuliert und die Ergebnisse analysiert werden. Mein Ziel ist es gewesen, dieses Modell in einem Framework umzusetzen und auf mobilen Tablet-Computern lauffähig zu machen. Dadurch soll zum einen die Möglichkeit für Experten eröffnet werden, die Optimierung ihrer Evakuierungspläne direkt vor Ort durchführen zu können und den Verantwortlichen vorzuführen. Sie sind somit auch in der Lage, Ideen und Verbesserungen direkt zu testen und einzuarbeiten. Aber nicht nur Experten soll der mobile Einsatz der Simulationssoftware dienen. Die Demonstration der Software kann Motivation für jüngeres Publikum sein, sich mit der Thematik zu beschäftigen, oder kann ihr Interesse für die Technik und Mathematik hinter dem Programm wecken.
Das folgende Kapitel befasst sich ausschließlich mit dem F.A.S.T.-Modell. Denn bevor es um die Erweiterung und Implementierung des Modells gehen kann, muss das Modell vorgestellt werden. Da es sich bei dem Modell um einen zellulären Automaten handelt, muss vor allem die Funktionsweise näher beschrieben. Aber auch Details werden angesprochen. So widmet sich ein Abschnitt den sogenannten Floor-Fields, deren gespeicherte Informationen Zeit bei der Simulationsdurchführung sparen. Auch die Regeln, nach denen eine Simulationsrunde aufgebaut ist, sind Teil dieses Kapitels. Hier liegt ein Hauptaugenmerk auf den Wahrscheinlichkeitsformeln, mit denen es zu den Änderungen im Automaten kommt. Zusätzlich widmet sich ein Abschnitt den Statistiken und Ausgaben, die eine Simulation mittels F.A.S.T. produziert und wie man sie darstellen kann.
In Kapitel drei geht es um Erweiterungen und Anpassungen, die am F.A.S.T.-Modell vorgenommen wurden. Zu ihnen zählen unter anderem spezielle Treppen- und Teleportzellen, mit deren Hilfe die Simulationsergebnisse realitätsnaher werden sollen. Ein entscheidender Zusatz für die Simulation, ist der Einsatz einer Sensitivitätsanalyse, mit deren Hilfe die Ergebnisse der Simulationen besser eingeschätzt und die simulierten Szenarios besser bewertet werden können. Die Umsetzung des Modells als Framework für einen mobilen PC machte allerdings auch manche Anpassungen nötig, vor allen in der Art und Weise, wie die Simulationsregeln Einfluss auf die Bewegungen der Personen haben, oder wie mit mehreren Ebenen in einem Simulationsgebiet verfahren werden soll.
Das vierte Kapitel befasst sich mit der tatsächlichen Implementierung des Modells. Zu Anfang des Kapitels wird erläutert, welche Plattform und Programmiersprache für die Programmierung verwendet wurde. Nachdem ein Überblick über die Programmteile der Anwendung erfolgt ist, gliedert sich der Rest des vierten Kapitels in drei Teile. Zuerst wird ein detaillierter

Blick auf die Initialisierung einer Simulation geworfen. Er enthält auch eine Beschreibung des Aufbaus der Konfigurationsdateien, die ein Szenario definieren. Als nächstes wird die Durchführung einer Simulation beschrieben. Dabei wird vor allem auf die Bewegung der Objekte und deren Suche nach besetzbaren Zellen eingegangen. Zusätzlich wird das gesamte Kapitel mit Algorithmen, erklärenden Bildern und Codeausschnitten unterstützt. Der letzte Teil des Kapitels legt den Blickpunkt auf die Programmierung der Ergebnisspeicherung und -darstellung.
Im letzten Kapitel sind Testergebnisse des Frameworks dargestellt. Das Programm wurde mit unterschiedlichen Szenarien auf dem mobilen Tablet-PC getestet, um dessen Leistung und Grenzen aufzuzeigen. So soll vor allem eine Einschätzung möglich gemacht werden, welche Faktoren die Simulationszeit und den Verbrauch an Arbeitsspeicher beeinflussen. Ein weiterer Test soll zeigen, wie die vorliegende Hardware die Größe der zu simulierenden Gebiete eingrenzt. Es werden auch zwei Probleme angesprochen, die im direkten Zusammenhang mit längeren Simulationszeiten stehen. Ein abschließendes Beispiel einer Sensitivitätsanalyse soll ihre Möglichkeiten im Bewerten der eingesetzten Simulationsparameter und den daraus resultierenden Ergebnissen aufzeigen.

2 Überblick über das F.A.S.T.-Modell

F.A.S.T. steht für „Floor field- and Agentbased Simulation Tool“. Es ist ein Modell zur Fußgängersimulation und wurde 2007 an der Universität Duisburg-Essen von Tobias Kretz entwickelt [Kre06]. Es baut auf einigen früheren Simulationswerkzeugen auf ([KKN+03], [Gat05]). In diesem Kapitel will ich das F.A.S.T.-Modell vorstellen. Dabei gehe ich auf einige Teilaspekte näher ein, wie z. B. die Floor-Fields, den Aufbau einer Simulationsrunde und die Ausgaben.

2.1 Grundprinzip

Das Modell basiert auf einem zellulären Automaten. Somit sind sowohl die Zeit, als auch der Raum diskretisiert. Es gibt unterschiedliche Typen von Zellen, sowie Agenten, die in einzelne Gruppen aufgeteilt werden können. Als „Agent“ wird in diesem Modell eine Personen bezeichnet. Ein Agent besetzt immer genau eine Zelle. Im Gegensatz zu normalen Zellen oder Ausgangszellen, können die speziellen Wandzellen nicht durch Agenten besetzt werden. Sie dienen als eine Art Hindernis für die Agenten auf ihren Wegen über das Zellenfeld. Mit Verweis auf [Dre67], wird die Zellengröße mit „ungefähr $40 \times 40 cm^2$, der kleinsten Fläche, die ein Fußgänger einnimmt“, angegeben. Ein Zeitschritt wird hier als „Runde“ bezeichnet. Mit jeder Runde bewegen sich die Agenten über das Feld der Zellen. Dabei wird ihre Bewegung von mehreren randomisierten Regeln beeinflusst. Auf Grund individueller Parameter jedes Agenten, wirken sich die Regeln unterschiedlich auf die Bewegungen aus. Die eben angesprochenen Gruppen fassen alle Agenten mit denselben Parameterwerten zusammen. Jeder Agent kann sich trotzdem unabhängig von Gruppenzugehörigkeiten bewegen. Das Ziel jedes Agenten ist es am Ende einer Runde eine Ausgangszelle zu besetzen. Über diese Art von Zellen kann ein Agent das Simulationsgebiet verlassen und hat somit auch keinen Einfluss mehr auf die übrigen Agenten. Bezogen auf die Realität kann man davon sprechen, dass er „erfolgreich entkommen“ ist. Das Ziel der Simulation ist erreicht, wenn der letzte verbliebene Agent eine Ausgangszelle besetzt hat und mit dem Ende der Runde keine Agenten mehr im Simulationsgebiet übrig sind.

2.2 Floor-Fields

Im F.A.S.T.-Modell werden sogenannte Floor-Fields eingesetzt. Sie wurden bereits in früheren Modellen zur Fußgängersimulation verwendet ([BKSZ01], [KS02], [NKNS04]). Die ursprüngliche Idee ist, dass diese Felder durch die Agenten auf bestimmte Weise verändert werden und

es dadurch zu einer Änderung der Übergangswahrscheinlichkeiten des Automaten kommt. Die Bewegung der Agenten wird dadurch gesteuert, indem Zellen bevorzugt werden, deren Eintrag im Floor-Field größer ist [BKSZ01].
Für das F.A.S.T.-Modell wurde dieses Prinzip übernommen. Vereinfacht kann man sagen, dass ein Floor-Feld Informationen zu jeder einzelnen Zelle speichert. Sie sind damit genauso groß wie das Simulationsgebiet. Je nachdem, welchem Zweck ein Floor-Field in der Simulation dient, bleiben die Werte in ihm über die Zeit hinweg konstant, oder werden durch die Agenten und globale Einflüsse in jeder Runde aktualisiert. Gerade die Floor-Fields, die Informationen bereithalten, die sich nicht mehr ändern, führen zu einer deutlichen Beschleunigung der Simulationszeit. Im Folgenden sind drei unterschiedliche Floor-Fields aufgeführt, die zum Teil im F.A.S.T.-Modellbenutzt werden und in meiner Implementierung eingebaut sind.

2.2.1 Statisches Floor-Field

In [Kre06, S. 20 ff.] und [Kre06, S. 37 ff.] wird das statische Floor-Field beschrieben. Es speichert Distanzwerte zwischen Zellen. Für jede Zelle werden die kürzesten Entfernungen zu jedem Ausgang berechnet und im entsprechenden Feld des Floor-Fields gespeichert. Gemessen wird die Distanz in „Anzahl Zellen“, wobei die Wertemenge nicht auf die Ganzen Zahlen beschränkt ist. Da es im Simulationsgebiet auch Wandzellen geben kann, die eine direkte Verbindung zweier Zellen verhindern, werden diese Hindernisse in den Berechnungen miteinbezogen. Die kürzeste Entfernung zwischen zwei Zellen setzt sich folglich oftmals aus mehreren Teilstrecken zusammen, welche um die Hindernisse herum führen. Die Distanzwerte müssen ein Mal zu Beginn der Simulation berechnet werden. Danach wird auf nur noch lesend auf sie zugegriffen. Eine detaillierte Beschreibung der Distanzberechnungen, sowie den Modifikationen der Algorithmen ist in Kapitel 3.5 und 4.3.3 vorhanden. Abbildung 2.1 zeigt eine Visualisierung eines statischen Floor-Fields für eine der Ausgangszellen.

2.2.2 Dynamisches Floor-Field

In F.A.S.T. wird außerdem das sogenannte „dynamische Floor-Field“ eingesetzt ([Kre06, S. 24 f.] und [Kre06, S. 28 f.]). Während die Werte des statischen Floor-Fields konstant über die Zeit der Simulation hinweg bleiben, verändern sich die Daten dieses Floor-Fields entsprechend der Agentenbewegungen. Jeder Agent hinterlässt durch seine Bewegung eine „Spur“. Dies drückt sich durch eine erhöhte Besetzungshäufigkeit der besuchten Zellen aus. Das dynamische Floor-Field speichert Vektoren, die diese Bewegungen in x- und y-Richtung aufsummieren. Je mehr Agenten mit hoher Geschwindigkeit und annähernd derselben Richtung über eine Zelle laufen, desto größer werden die Vektorkomponenten. Abbildung 2.2 zeigt eine Situation, in der sich zwei Agenten nacheinander von einer Zelle wegbewegen und damit den entsprechenden Vektoreintrag verändern. Der gezeigten Berechnung liegt folgende Vorschrift zu Grunde: Ein Agent, der sich von einer Zelle an Position (a, b) zu einer Zelle an Position (x, y) bewegt, addiert $(x - a, y - b)$ auf das Feld im dynamischen Floor-Field an Position (a, b) [Kre06, S. 24]. Das dynamische Floor-Field wird nicht nur durch die Agenteninteraktionen beeinflusst, sondern auch durch das Verwischen („diffusion“) und Verblassen („decay“). Beides sorgt dafür,

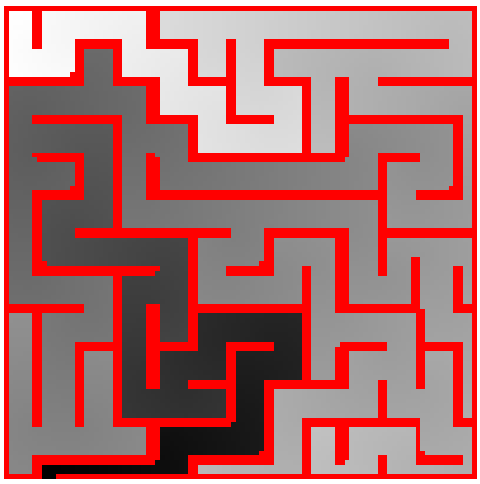


Abbildung 2.1: Beispiel für ein statisches Floor-Field. Wandzellen sind rot eingefärbt. Je weiter eine Nicht-Wandzelle vom Ausgang entfernt ist, desto heller ist ihr Grauwert.

dass die Agentenspuren mit der Zeit verwaschen und irgendwann komplett verschwinden, wenn sich keine weiteren Agenten mehr über die entsprechenden Zellen fortbewegen. In F.A.S.T. gibt es deshalb zwei globale Parameter, die mit α (Verwischung) und δ (Verblassen) bezeichnet werden. Sie geben die Wahrscheinlichkeit an, mit der „alle Werte beider Komponenten“ [Kre06, S. 24] des dynamischen Floor-Field diffundieren bzw. zerfallen. Eine Diffusion findet nur zwischen den entsprechenden x- und y-Komponenten benachbarter Zellen statt. Es gelten in diesem Fall nur horizontal und vertikal gelegene Zelle als benachbart. Eine Diffusion zwischen diagonal benachbarter Zellen ist folglich nicht möglich. Ist die Vektorkomponente der einen Zelle negativ, führt die Diffusion zu einer Verringerung des Wertes der anderen Zelle, unabhängig, ob ihr Wert positiv oder negativ ist und umgekehrt, für einen positiven Ausgangswert. Beim Verblassen gilt allgemein, dass sich der Betrag der Komponente verringert. Ein negativer Wert nähert sich somit gleichermaßen der Null an, wie ein positiver.
Nachdem alle Agenten ihre Bewegungen für die Runde abgeschlossen haben, wird das dynamische Floor-Field entsprechend den Wahrscheinlichkeitswerten von α und δ verändert. Abbildung 2.8 zeigt eine Momentaufnahme des dynamischen Floor-Fields während einer Simulation.

2.2.3 Floor-Field für Wanddistanzen

Wände bzw. Wandzelle haben einen Einfluss auf die Bewegung der Agenten (siehe 2.3.2). Deshalb ist es notwendig, von einer Zelle zu wissen, wie nahe sie an einer Wandzelle liegt. Ein Floor-Field, das diese Distanzen speichert, ist im F.A.S.T.-Modell nicht explizit beschrieben. Eine Verwendung bei der Implementierung des Modells liegt allerdings nahe, betrachtet man vor allem die zeitlichen Vorteile von Floor-Fields. Die Distanzen ändern sich im Verlaufe der Simulation nicht und können deshalb wie die Werte des statischen Floor-Fields zu Beginn der

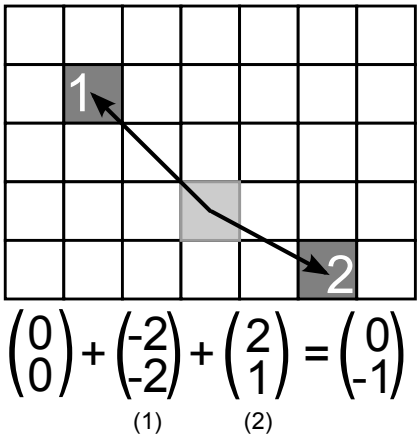


Abbildung 2.2: Beispiel für eine Änderung des dynamischen Floor-Fields: Zwei Agenten bewegen sich in unterschiedliche Richtung von derselben Zellen weg. Ihre Bewegungsvektoren werden auf den Eintrag an der Position der hellgrauen Zelle im dynamischen Floor-Field addiert.

Simulation einmalig berechnet werden. In Abbildung 2.10 sieht man eine Darstellung dieses Feldes und in Kapitel 4.3.3 sind weitere Details zur Implementierung verfügbar.

2.3 Aufbau einer Simulationsrunde

Eine Simulationsrunde besteht aus drei Abschnitten:

1. Die Agenten wählen einen Ausgang.
2. Sie entscheiden sich für eine Zielzelle, die sie nach dieser Runde besetzen wollen.
3. Alle Agenten führen ihre Bewegungen aus.

Abbildung 2.3 zeigt das Schema einer Runde. Es basiert auf der Abbildung in [Kre06, S. 12]. Zuerst wählt jeder Agent einen Ausgang aus. Da er meistens aus einer Gruppe von Ausgangszellen besteht, wird nicht nur der Ausgang selbst ausgewählt, sondern auch gezielt eine seiner Ausgangszellen. Diese Zelle dient dem Agenten als Fernziel, dem er Runde für Runde näher kommen will. Im nächsten Schritt wählt jeder Agent eine Zielzelle für sich aus. Diese stammt aus der Menge aller Zellen, die ein Agent mit seiner aktuellen Geschwindigkeit und von seiner derzeitigen Position aus in dieser Runde erreichen kann. Die Auswahl des Ausgangs und der Zielzelle kann unabhängig von den anderen Agenten erfolgen. Das Modell erlaubt hier den Einsatz nebenläufiger Prozesse zur Beschleunigung der beiden Abschnitte. Die Bewegungen jedes Agenten zu seiner Zielzelle finden allerdings sequentiell statt (siehe dazu auch Kapitel 2.3.3). Im Gegensatz zu den ersten beiden Abschnitten, in denen die Auswahl durch wahrscheinlichkeitsbasierte Formeln getroffen wird, ist der dritte Abschnitt rein deterministisch. Wie [Kre06, S. 28 ff.] zeigt, gibt es verschiedene Ansätze die Agentenbewegungen

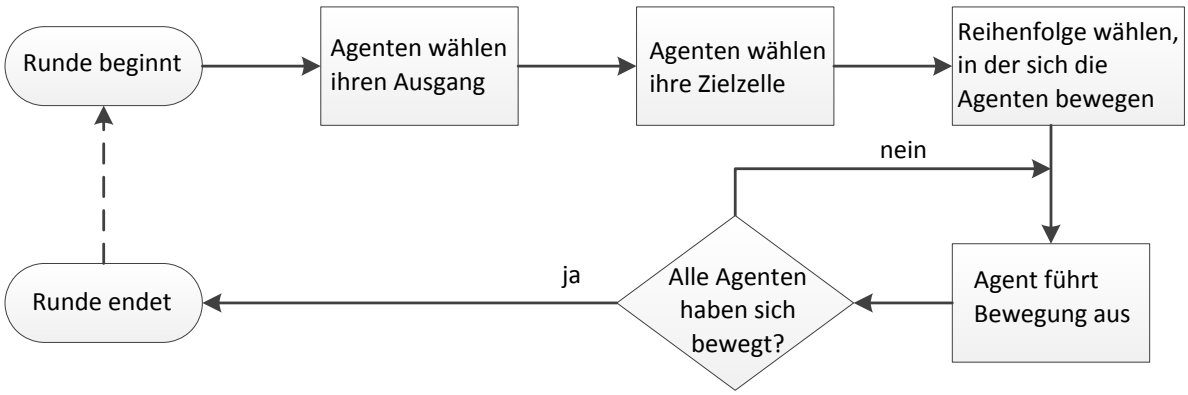


Abbildung 2.3: Schematischer Aufbau einer Simulationsrunde

durchzuführen. Es muss grundsätzlich eine Reihenfolge der Agenten festgelegt werden, in der sie ihre Schritte durchführen. Erst wenn alle Agenten ihre Bewegungen vollendet haben, endet die Runde und die nächste kann beginnen.

2.3.1 Wahl eines Ausgangs

Die Wahl eines Ausgangs folgt bei F.A.S.T. einem randomisierten Prozess [Kre06, S. 11 ff.]: Es werden die Wahrscheinlichkeiten für jeden einzelnen Ausgang berechnet. Aus dieser Menge wird einer, entsprechend der Gewichtung der Einzelwahrscheinlichkeiten ausgewählt. Zur Berechnung der einzelnen Wahrscheinlichkeiten, gibt das F.A.S.T.-Modell folgende Formel vor ([Kre06, S. 12]):

$$p_E^A = N \cdot \frac{(1 + \delta_{AE}\kappa_E)}{S(A,E)^2} \tag{2.1}$$

Die einzelnen Variablen und Terme haben folgende Bedeutung:

- A steht für den aktuellen Agenten.
- E bezeichnet den Ausgang, der gerade betrachtet wird.
- $\delta_{AE} = 1$, wenn A den Ausgang E in der Runde zuvor bereits gewählt hatte. Sonst ist $\delta_{AE} = 0$
- κ_E ist eine Eigenschaft von A und gibt an, wie beharrlich der Agent bei einer einmal getroffenen Entscheidung für einen Ausgang bleibt. Das „E“ bezeichnet in diesem Fall nicht den Ausgang, sondern dient zur Unterscheidung der anderen κ_X (siehe die Formeln 2.3 bis 2.8).
- $S(A,E)$ gibt die Entfernung zwischen der aktuellen Position von A und E an. Dieser Wert ist im statischen Floor-Field S gespeichert.
- N dient als Normalisierungskonstante. Sie soll sicherstellen, dass $\sum_E p_E = 1$ gilt.

In Kapitel 4.4.2 wird eine Anpassung der Formel 2.1 beschrieben, die durch die Gruppierung von Ausgangszellen zu Ausgängen nötig wird.

2.3.2 Wahl einer Zielzelle

Die Geschwindigkeit eines Agenten hat entscheidenden Einfluss auf die Anzahl der Zellen, zwischen denen er sich für eine Zielzelle entscheiden kann. Deshalb soll sie hier kurz definiert werden. In [Kre06, S. 13] entspricht die „dimensionlose Geschwindigkeit, der Anzahl an Zellen, die er (der Agent) sich während einer Runde fortbewegen darf". Die Größe einer Zelle ist vom Modell mit circa $40 \times 40 cm^2$ vorgegeben (vgl. 2.1). Somit hängt die Geschwindigkeit einzig von der Definition des Zeitschritts einer Runde ab. Soll beispielsweise eine Runde $1s$ lang sein, entspricht eine Geschwindigkeit von $4\frac{Zellen}{Runde}$ einer realen Geschwindigkeit von $1,6\frac{m}{s}$. Ist der Zeitschritt dagegen $10ms$, gilt für die reale Geschwindigkeit $160\frac{m}{s}$.
Wie auch schon die Wahl des Ausgangs, ist die Wahl der Zielzelle ein wahrscheinlichkeitsbasierter Prozess. In diesem Fall werden allerdings die Wahrscheinlichkeiten aller Zellen gesammelt, die vom Agenten erreichbar sind. Aus dieser Menge wählt er eine Zelle aus. Es gibt drei Kriterien, nach denen eine Zelle als erreichbar gilt:

- Sie ist keine Wandzelle. Wandzellen können nicht besetzt werden.
- Sie liegt im Bereich der Zellen, die mit der aktuellen Geschwindigkeit des Agenten besetzt werden können.
- Sie ist nicht bereits durch einen anderen Agenten besetzt.

Die folgende Formel und alle darin enthalten Teilformeln sind aus [Kre06, S. 24] entnommen und im Folgenden beschrieben. Die Wahrscheinlichkeit p_{xy}, mit welcher der Agent A die Zelle an Position (x, y) als Zielzelle auswählt, wird berechnet durch:

$$p_{xy} = N p^S_{xy} p^D_{xy} p^I_{xy} p^W_{xy} p^P_{xy}, \tag{2.2}$$

wobei die p^X_{xy} für die einzelnen Faktoren stehen, welche die Wahrscheinlichkeit der Zelle beeinflussen. Sie sind alle nachfolgend erklärt. N ist eine Normalisierungskonstante, die garantieren soll, dass $\sum_{xy} p_{xy} = 1$ gilt.

Einfluss des statisches Floor-Fields

Das statische Floor-Field speichert die kleinsten Distanzen aller Zellen zu allen Ausgangszellen (siehe 2.2.1). Die folgende Formel beschreibt den Einfluss, indem sie die Entfernung einer Zelle zum gewählten Ausgang des Agenten miteinbezieht ([Kre06, S. 24]):

$$p^S_{xy} = e^{-\kappa_S S^E_{xy}}. \tag{2.3}$$

Die Variable κ_S ist eine Agentenkonstante. Alle Teilformeln von (2.2) und die Formel für die Wahl des Ausgangs (2.1) benutzen verschiedene Agentenkonstanten κ_X. Sie können individuell für jeden Agenten oder jede Agentengruppe zu Beginn der Simulation gesetzt

werden. Jeder Agent speichert sich für sich ab. Die Aufgabe dieser Konstanten ist es, die Einflüsse auf die Gesamtwahrscheinlichkeit von Formel (2.2) untereinander zu gewichten. S^E_{xy} ist der Eintrag im statischen Floor-Field, der die Entfernung zwischen der Zelle an Position (x, y) und dem Ausgang E speichert. E entspricht dem Ausgang, den der Agent im Schritt zuvor ausgewählt hat.
Mit dieser Formel werden Zellen, die näher am Ausgang liegen, attraktiver für den Agenten. Zellen, die ihn weiter vom gewählten Ausgang wegführen würden, erhalten eine geringere Attraktivität.

Einfluss des dynamischen Floor-Fields

Das dynamische Floor-Field speichert die Bewegungensspuren der Agenten über die einzelnen Zellen ab (siehe 2.2.2). Es speichert sowohl die Richtung, als auch implizit den Geschwindigkeitsbetrag. Das Floor-Field kann damit die Richtung der Agenten steuern: Zellen, über die sich viele, schnelle Agenten in der gleichen Richtung bewegt haben, besitzen große Komponentenwerte im dynamischen Floor-Field. Sie sind damit attraktiver für neue Agenten, als Zellen, über die nur wenige oder langsame Agenten gelaufen sind bzw. bei denen jeder Agent eine andere Laufrichtung hatte.
Die Formel, die hinter dieser Überlegung steckt, ist folgende ([Kre06, S. 24 f.]):

$$p^D_{xy} = e^{\kappa_D(D_x(x,y)(x-a)+D_y(x,y)(y-b))} \tag{2.4}$$

Dabei ist κ_D eine der Agentenkonstanten, mit der dieser Einfluss gewichtet wird. (a, b) bezeichnet die Position des Agenten, während (x, y) für die Position der potenziellen Zielzelle steht. $D_x(x, y)$ bzw. $D_y(x, y)$ gibt die X- bzw. Y-Komponente des Vektors im dynamischen Floor-Field für die Zelle (x, y) an.

Einfluss der Trägheit

An dieser Stelle spielt die Physik und der Bewegungsapparat der Menschen eine entscheidende Rolle. Nach [Kre06, S. 25] ist das Beschleunigen und Verzögern für einen Menschen praktisch sofort möglich; geht man von 1-Sekunden-Zeitschritten aus. Dagegen ist eine Richtungsänderung um 90° bei gleichbleibender Geschwindigkeit sehr viel schwieriger für einen Menschen durchführbar. Die folgende Formel soll diese Überlegungen umsetzen und den Einfluss der Trägheit auf die Zielzellenwahl widerspiegeln:

$$p^I_{xy} = e^{-\kappa_I F^{xy}_c}, \tag{2.5}$$

wobei κ_I für eine der Agentenkonstanten steht und F^{xy}_c die Fliehkraft für die Zelle an Position (x, y) beschreibt. Durch diese Formel werden Zellen, die in der aktuellen Laufrichtung des Agenten liegen, attraktiver, als solche, die einen größeren Winkel zur Laufrichtung aufweisen. Zur Berechnung der Fliehkraft wird die Geschwindigkeit des Agenten während der aktuellen Runde benötigt. Sie ist zu diesem Zeitpunkt allerdings nicht bekannt. Das liegt an der Diskretisierung des Raumes und der Zeit ([Kre06, S. 25]: Eine Geschwindigkeit ist

nur definiert, wenn sich der Agent tatsächlich bewegt. Sie hängt von der Zielzelle ab, die in diesem Schritt erst noch gesucht wird. Es wird deshalb der Durchschnittswert aus der letzten Agentengeschwindigkeit und der wahrscheinlichen Geschwindigkeit am Ende der Runde genommen. Die Formel zur Berechnung der Fliehkraft lautet dementsprechend wie folgt ([Kre06, S. 25 ff.]):

$$F_c^{xy} = (v_{next} + v_{last}) \sqrt{\frac{1}{2}\left(1 - \frac{\begin{pmatrix} \Delta x_{t+1} \\ \Delta y_{t+1} \end{pmatrix}\begin{pmatrix} \Delta x_t \\ \Delta y_t \end{pmatrix}}{\left\|\begin{pmatrix} \Delta x_{t+1} \\ \Delta y_{t+1} \end{pmatrix}\right\|\left\|\begin{pmatrix} \Delta x_t \\ \Delta y_t \end{pmatrix}\right\|}\right)}, \tag{2.6}$$

wobei

- v_{last} die Geschwindigkeit ist, mit der sich der Agent in der vergangenen Runde bewegt hat,
- v_{next} die Geschwindigkeit des Agenten bezeichnet, mit der er die Zelle (x, y) erreichen kann,
- Δx_{t+1} bzw. Δy_{t+1} die relative Entfernung angibt, zwischen der in dieser Runde theoretisch erreichbaren Zelle und der aktuellen Position des Agenten (in x- und y-Richtung getrennt),
- Δx_t bzw. Δy_t die relative Entfernung angibt, zwischen der aktuellen Agentenposition und der in der vergangenen Runde (in x- und y-Richtung getrennt).

In Abbildung 2.4 sieht man die Ergebnisse der Formel (2.5) für einen Teil der Zellen mit Geschwindigkeit $v_{last} = 3$.

Einfluss benachbarter Wände

In [Kre06, S. 27] wird der Einfluss benachbarter Wände als „Sicherheitsabstand“ beschrieben, den die Agenten zu Wänden einhalten. Somit werden Zellen, die in der Nähe von Wandzellen liegen, unattraktiver für Agenten. Menschen wollen vor allem in Fluchtsituationen nicht nahe an Wänden oder anderen Hindernissen entlanglaufen ([Kre06, S. 27], [NKNS04, S. 7]). Beide Quellen führen folgende Formel zur Berechnung dieses Einflusses auf:

$$p_{xy}^W = e^{-\kappa_W W_{xy}}. \tag{2.7}$$

κ_W ist eine weitere Agentenkonstante und W_{xy} gibt die Entfernung zu der Wandzelle an, die am nächsten an der Zelle (x, y) liegt. Ist die Zelle weiter als W_{max} von einer Wandzelle entfernt, gilt $p_{xy}^W = 1$ und die Wandzellen haben keinen Einfluss mehr auf die Wahl dieser Zelle.

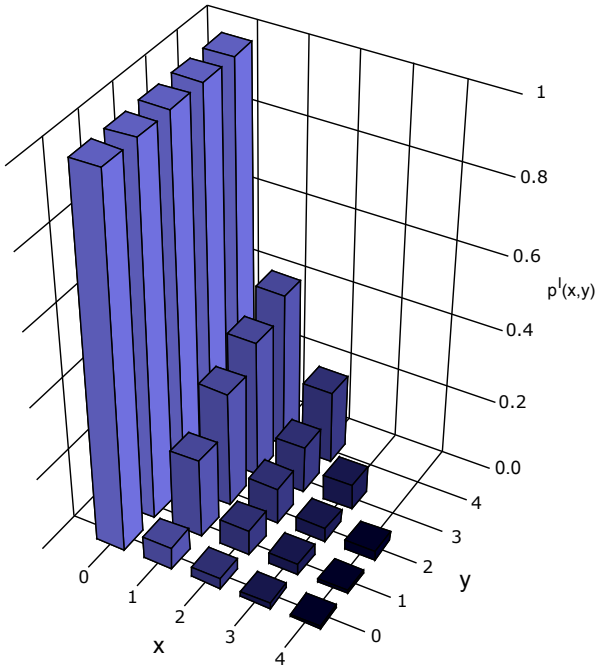


Abbildung 2.4: Ergebnisse von p^I_{xy} (Formel 2.5) für Agent mit Geschwindigkeit $v_{last} = 3$. Der Agent bewegte sich von $(0, -3)$ nach $(0, 0)$ in der Runde zuvor. Es ist nur ein Quadrant darstellt.

Einfluss benachbarter Agenten

Ähnlich wie die Nähe zu Wänden und Hindernissen, meiden Menschen in Fluchtsituationen größere Menschenmengen; sofern sie eine Wahl haben [Kre06, S. 27 f.]. Die folgende Formel macht Zellen unattraktiv, wenn ihre Nachbarschaft durch viele Agenten besetzt ist:

$$p^P_{xy} = e^{-\kappa_P N_P(x,y)} \tag{2.8}$$

Die Agentenkonstante κ_P dient auch hier der Gewichtung des Einflusses. $N_P(x, y)$ gibt die Anzahl der Agenten an, die sich in der unmittelbaren Nachbarschaft der Zelle (x, y) befinden. Der Agent, von dem die Zielzellensuche ausgeht, wird hier nicht mitgezählt. Deshalb wird dieser Einfluss auch ignoriert, wenn keine Agenten in der Nähe sind ($N_P(x, y) = 0 \Rightarrow p^P_{xy} = 1$). Auch in einer Situation, in der alle erreichbaren Zellen von vielen Agenten umgeben sind verschwindet dieser Einfluss aus der Zielzellensuche: Alle Zellen werden durch die Formel gleichermaßen unattraktiv gemacht. Die Wahl der Zielzelle hängt damit von anderen Faktoren ab.

2.3.3 Bewegung zur Zielzelle

Haben alle Agenten ihre Zielzelle ausgewählt, werden sie in diesem Schritt versuchen, sich zu ihr zu bewegen. In [Kre06, S. 28 ff.] werden verschiedene Möglichkeiten beschrieben, dies zu realisieren:

- **Direktes Hinspringen:** Der Agent springt direkt von seiner jetzigen Zelle zur Zielzelle. Dazwischenliegende Zellen haben keinen Einfluss auf die Bewegung des Agenten und werden umgekehrt von ihm auch nicht verändert. Dieses Vorgehen ist vergleichsweise einfach umzusetzen. Allerdings gibt es auch ein Problem: Wenn zwei oder mehr Agenten dieselbe Zielzelle ausgewählt haben, kann nur einer von ihnen am Ende der Runde auf diese besetzen. Es entsteht ein sogenannter „Konflikt“. Mit der Wahrscheinlichkeit μ bleibt er ungelöst und keiner der betroffenen Agenten darf sich zur Zielzelle bewegen. Andernfalls wird einer von ihnen per Zufall ausgewählt. Dieser darf sich als einziger auf die Zielzelle bewegen. Alle anderen bleiben auf ihrer aktuellen Zelle stehen.
- **Zelle für Zelle hinbewegen:** Bei dieser Vorgehensweise bewegt sich ein Agent zu seiner Zielzelle, indem er eine Reihe einzelner Sprünge zu direkt angrenzenden Zellen ausführt. Die Zellen, die er auf dem Weg zur Zielzelle besucht, gelten bis zum Abschluss der Runde als besetzt. Sie können nicht von anderen Agenten betreten werden. Diese Art der Agentenbewegung gilt als realistischer, ist allerdings komplizierter in der Umsetzung [Kre06]. Zudem können Konflikte hier auf zweierlei Weise entstehen: Zum einen durch gemeinsame Zielzellen verschiedener Agenten (wie zuvor), aber auch durch sich kreuzende Pfade zu unterschiedlichen Zielzellen.
  Das Modell schlägt eine etwas abgewandelte Form dieses Vorgehens vor. Da ich in meiner Implementierung von diesem Vorschlag abgewichen bin und stattdessen eine leicht modifizierte Version der ersten Variante umgesetzt habe, verweise ich für weitere Details zum Modell-Vorgehen auf die entsprechende Literaturstelle [Kre06, S. 28 ff.].

Die Bewegungen der Agenten ist ein sequentieller Teil der Runde. Aus diesem Grund gibt es verschiedene Möglichkeiten eine Reihenfolge der Agenten zu bestimmen. Die folgenden vier sind aus [Kre06, S. 30] entnommen.

Konstante Reihenfolge: Zu Beginn der Simulation wird die Reihenfolge festgelegt. Sie bleibt danach über die gesamte Länge der Durchführung hinweg dieselbe.

Verschobene Reihenfolge: Es existiert eine feste Reihenfolge für die Bewegungen der einzelnen Agenten. Sie wird vor Beginn der Simulation festgelegt. Allerdings wird jede Runde die Nummer des Agenten, der beginnen darf, durch einen festen oder zufälligen Wert verschoben. Aus der Reihenfolge 1,2,3,4 wird in der nächsten Runde z. B. 3,4,1,2 gemacht.

Zufällige Reihenfolge: Die Reihenfolge wird in jeder Runde neu, zufällig bestimmt.

Vollkommen zufällige Reihenfolge: Diese Variante unterscheidet sich nur dann von der vorhergehenden, wenn sich die Agenten schrittweise zu ihrer Zielzelle bewegen. In allen vorherigen Varianten führt ein Agent **alle** seine Schritte aus, bevor der nächste Agent an der Reihe ist. Bei dieser Variante wird nicht nur die Agentenreihenfolge in jeder Runde zufällig gewählt, sondern auch die Anzahl der ausgeführten Schritte eines Agenten. Bis ein Agent

seine Zielzelle erreicht hat, kann er folglich mehrmals an der Reihe gewesen sein. In [Kre06, S. 30] wurde dieses Schema als Standard für alle Berechnungen verwendet.

2.4 Ausgaben

Eine Simulation, wie sie hier nach dem F.A.S.T.-Modell durchgeführt wird, erzeugt eine Vielzahl Daten und Ergebnisse. Einen wichtigen Teil stellt daher auch die Visualisierung und Aufbereitung dieser Ergebnisse dar. Viele Resultate können in Form von Graphen und Bildern dargestellt werden, andere Statistiken benötigen eine schlichte Textform. Eine grafische Darstellung der Simulation macht es möglich, einige der Ausgaben bereits zur Laufzeit anzuzeigen. Die folgende Auflistung ist aus [Kre06, S. 83-92] entnommen und enthält die, meiner Meinung nach, wichtigsten Ausgaben einer Simulation.

2.4.1 Statistiken

Nach jedem einzelnen Durchlauf wird die Anzahl der Runden dokumentiert, bis 95% und 100% der Agenten entkommen sind. Abschließend wird aufgeführt, wie viele Runden es mindestens, höchstens und durchschnittlich pro Durchlauf gedauert hat, bis alle 95% bzw. 100% der Agenten fliehen konnten. Zusätzlich wird die Standardabweichung vom Durchschnittswert ausgegeben.
Für jeden Ausgang wird dokumentiert, wie viele Agenten über ihn während den einzelnen Durchläufen entkommen sind. Nach Beendigung der Simulation gibt es eine Auflistung der Ausgänge, sortiert nach ihrer ID, mit der Angabe der Minimum-, Maximum- und Durchschnittswerte. In [Kre06, S. 90] wird diese Ausgabe in Form eines Graphen beschrieben, bei dem die ID der Ausgänge auf der Abszisse verteilt sind und die Anzahl, der über die Ausgänge geflüchteten Agenten, auf der Ordinate aufgetragen sind.
Ein weiterer Vorschlag von [Kre06] ist die Ausgabe der Namen von Agentengruppen, deren Mitglieder zu den letzten drei noch nicht entkommenen Agenten zählen.

2.4.2 Graphen

Der **Evakuierungsgraph** ist ein sehr aussagekräftiger Graph in der Analyse einer Simulation. Er spiegelt den Verlauf der Evakuierung wider, indem er für jede Runde die Anzahl der bereits entkommenen Agenten anzeigt. Je größer die Steigung der Kurve ist, desto mehr Agenten entkommen pro Runde. Die Kurve ist monoton steigend. Gewöhnlich ist der Verlauf zu Beginn eines Durchlaufs sehr flach, da sich nur wenig Agenten in der Nähe von Ausgängen aufhalten. Das gleiche gilt für das Ende eines Durchlaufs, wenn nur noch wenige Agenten im Gebiet unterwegs sind und damit die Dichte abnimmt. Abbildung 2.5 zeigt einen exemplarischen Evakuierungsgraphen nach mehreren vollendeten Simulationsdurchläufen. Wie auch in [Kre06, S. 88], kann der Graph mit zwei zusätzlichen Kurven dargestellt werden: Den Maximal- und Minimalwerten. Sie sind im Beispielgraph grün bzw. rot eingefärbt. Die Maximalwerte

geben an, wie viele Agenten zu der entsprechenden Runde, über alle bisherigen Durchläufe gerechnet, maximal entkommen sind. Analoges gilt für die Minimalwerte. Man kann im Beispielgraph sehen, dass die grüne Kurve nicht monoton ist. Das liegt an der Definition der Maximalwerte: Wenn zu einer Runde x in einem Durchlauf alle Agenten entkommen sind, hat die Kurve dort ihr Maximum erreicht, der Durchlauf ist nach Definition des Modells beendet und es werden auch keine neuen Werte mehr für diesen Graph aufgenommen. Wenn nun in einer Runde $x+1$ in jedem der nachfolgenden Durchläufen die Anzahl die entkommenen Agenten kleiner als 100% ist, ist der Maximalwert an dieser Stelle kleiner als der in Runde x. Das Modell gibt an dieser Stelle nicht explizit vor, wie die Maximal- und Minimalwerte aufzunehmen sind, so dass auch eine Darstellungsvariante möglich ist, in der die grüne Kurve konstant fortläuft, sobald der Maximalwert erreicht wurde. Dadurch ist auch die Monotonie dieser Kurve sichergestellt.
Ein weiterer wichtiger Graph ist der Verlauf des **Durchschnittsflusses**. Er zeigt für jede Runde den aktuellen durchschnittlichen Fluss der Agenten an. Er kann laut [Kre06, S. 88 ff.] auf unterschiedliche Arten berechnet werden. Eine davon bezieht den Fortschritt der Agenten im statischen Floor-Field mit ein. Im Modell ist keine Variable zum Speichern des Durchschnittsflusses angegeben. Ich habe mich für $\tilde{f}$ entschieden. Mit der folgenden Formel kann der durchschnittliche Fluss berechnet werden:

$$\tilde{f} = \max\left(0, \frac{\sum_{a \in A} (\Delta S) \cdot |A|}{|\text{freie Zellen}|}\right), \tag{2.9}$$

wobei ΔS den angesprochenen Fortschritt im statischen Floor-Field bezeichnet, d. h. die Differenz aus alter und neuer Distanz zwischen der Ausgangszelle und dem Agenten. Die Anzahl, der noch in der Simulation befindlichen Agenten, ist mit $|A|$ bezeichnet und $|\text{freie Zellen}|$ gibt die Differenz von besetzbaren und durch Agenten besetzte Zellen an. Siehe Abbildung 2.6 für einen entsprechenden Beispielgraphen.

2.4.3 Bilder

Die **aktuelle Geschwindigkeit der Agenten** zählt zu den Ausgaben, die jederzeit während der Simulation aktualisiert und angezeigt werden können. Die Agenten werden entsprechend ihrer aktuellen Geschwindigkeit eingefärbt. Eine grüne Farbe entspricht dabei der individuellen Maximalgeschwindigkeit eines Agenten. Rot dagegen dem Stillstand des Agenten. Alle Werte dazwischen werden linear von Rot nach Gelb und von Gelb nach Grün interpoliert. In Abbildung 2.7 ist ein Beispiel zu sehen.
Das **dynamische Floor-Field** (siehe Kapitel 2.2.2) kann ebenfalls visualisiert werden. Da es Vektoren speichert, kann man bei der Visualisierung auf einen Trick zurückgreifen, der in [Kre06, S. 43] beschrieben ist: Die Farbe der Zelle richtet sich nach der Richtung und der relativen Länge des Vektors, im Vergleich zum längsten Vektor des dynamischen Floor-Fields. Die Richtung gibt den Farbton vor, während die Länge die Farbhelligkeit und -sättigung bestimmt. Diese beiden Werte eignen sich hervorragend für den Einsatz des HSV-Farbraumes (*H*ue, *S*aturation, *V*alue). Ein Beispiel für solch eine Visualisierung ist in Abbildung 2.8 zu betrachten.
Eine weitere Ausgabe, die bereits während der Simulationsdurchführung angezeigt werden

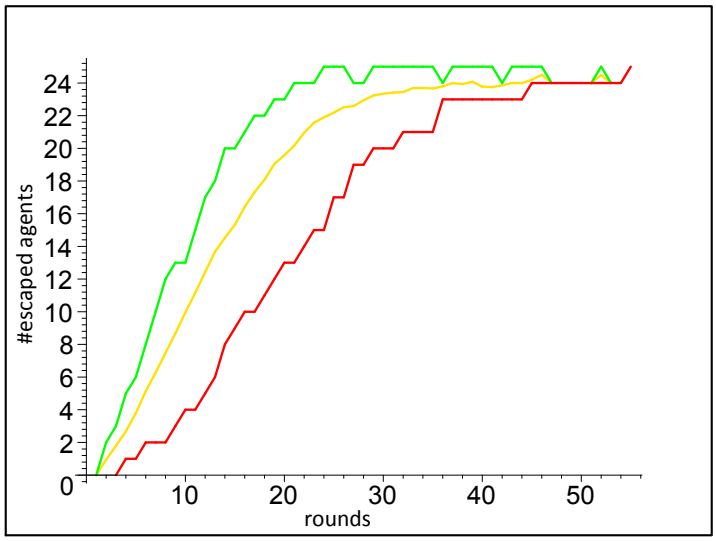


Abbildung 2.5: Ein Beispiel für einen Evakuierungsgraphen. Die grüne Kurve zeigt die Maximalwerte, die rote die Minimalwerte und die gelbe die Durchschnittswerte.

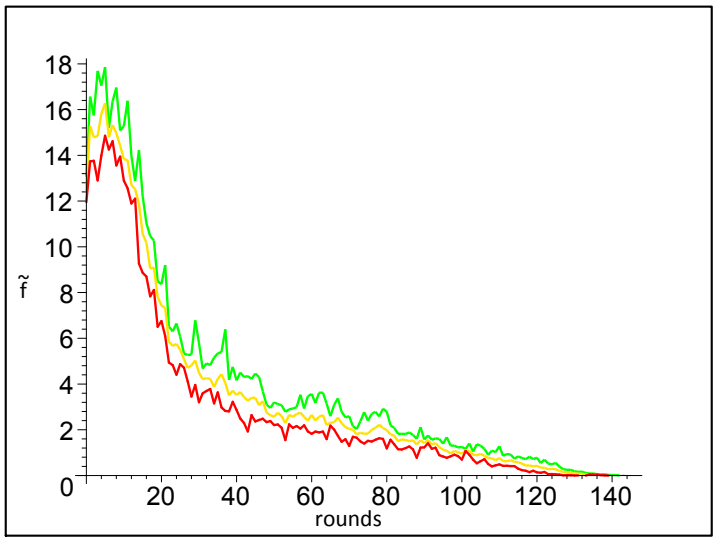


Abbildung 2.6: Ein exemplarischer Graph des Durchschnittsflusses der Agenten. Die grüne Kurve zeigt die Maximalwerte, die rote die Minimalwerte und die gelbe die Durchschnittswerte.

Abbildung 2.7: Dieses Beispiel zeigt eine Visualisierung der Agentengeschwindigkeiten. Der Farbverlauf von Rot über Gelb nach Grün entspricht der relativen Geschwindigkeit eines Agenten von 0 bis v_{max}.

kann, ist die Visualisierung der **lokalen Dichte** der Agenten. Hier gibt die Farbe einer Zelle die Besetzung der jeweils direkten Nachbarzellen an. Entscheidend ist der Quotient aus der „Anzahl direkt angrenzender besetzter Zellen“ und den „insgesamt besetzbaren Zellen“. Wandzellen werden dabei ignoriert. Als direkte Nachbarzelle gelten die diagonal, horizontal und vertikal anliegenden Zellen um die betrachtete Zelle herum. Ist der Quotient 0, wird die Zelle grün eingefärbt, denn alle benachbarten Zellen sind frei. Rot wird die Zelle eingefärbt, wenn der Quotient bei 1 liegt, denn das bedeutet, dass alle Zellen um die betrachtete Zelle herum durch Agenten besetzt sind. Für die entsprechenden Zwischenwerte wird der Farbwert linear interpoliert. Ein Beispiel ist in Abbildung 2.9 zu sehen.
Auch das **Floor-Field der kürzesten Wanddistanzen**, kann dargestellt werden. Ist das Maximum aller Werte dieses Floor-Fields bekannt (greatestWallDist), können die Farbwerte für jede einzelne Zelle mit folgender Formel berechnet werden:

$$\mathsf{RGBcolor} = (0, 0, \frac{\mathsf{cell}_{\mathsf{WallDist}}}{\mathsf{greatestWallDist}} \cdot 255),$$

wobei $\mathsf{cell}_{\mathsf{WallDist}}$ die Wanddistanz der aktuellen Zelle ist und RGBcolor ein Tripel mit den Farbkomponenten (Rot, Grün, Blau), jeweils mit Werten im Intervall $[0, 255]$. Je weiter eine Zelle von einer Wandzelle entfernt liegt, desto blauer wird sie dargestellt. Zellen, welche direkt an eine Wandzelle angrenzen, sind annähernd schwarz eingefärbt. Die dazwischen liegenden Zellen erhalten einen linear interpolierten Farbwert. Abbildung 2.10 zeigt ein Beispielbild.

1

Abbildung 2.8: Beispiel für ein dynamisches Floor-Field. Wandzellen sind weiß eingefärbt. Der Farbton der anderen Zellen hängt von der Richtung des Vektors ab, die Helligkeit vom Betrag des Vektors.

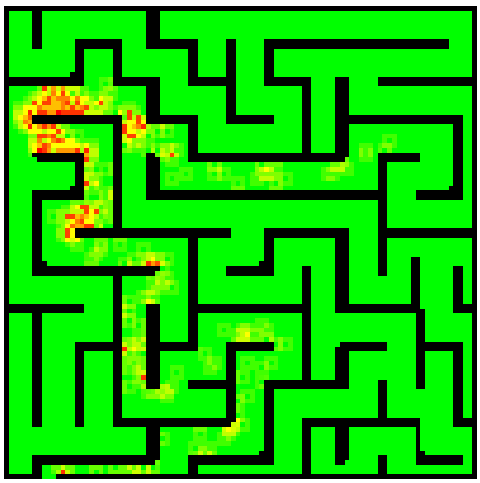


Abbildung 2.9: Eine exemplarische Visualisierung der lokalen Agentendichte. Grün sind Zellen, bei denen alle besetzbaren Nachbarzellen unbesetzt sind. Je mehr von ihnen durch Agenten belegt sind, desto roter werden sie eingefärbt.

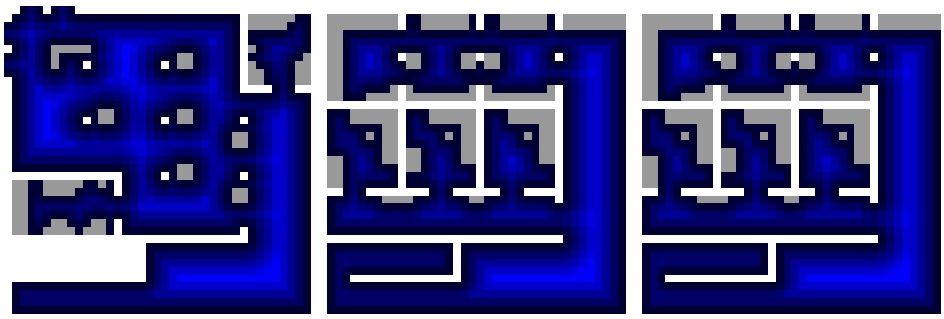


Abbildung 2.10: Beispiel für ein Floor-Field für Wanddistanzen. Wandzellen sind je nach Typ mit einem unterschiedlichen Grauwert eingefärbt(siehe 3.2). Für alle anderen Zellen gilt: Je weiter entfernt sie von einer Wandzelle sind, desto bläulicher ist ihre Farbe.

3 Erweiterungen

Das F.A.S.T.-Modell soll dazu beitragen realistische Simulationen durchzuführen. Die Ergebnisse wurden mit echten Experimenten verglichen, um die Wahl der Parameter und das Verhalten der Agenten zu optimieren [Kre06, S. 97-148]. Dennoch kann es in manchen Szenarien notwendig sein, zusätzliche Elemente in die Simulation hineinzubringen, um die Ergebnisse realistischer zu gestalten. Einige Ideen sind bereits in der Beschreibung des Modells genannt, wie z. B. Treppen, zusätzliche Stockwerke oder Anziehungspunkte ([Kre06, S. 30 ff.]). Ich habe diverse Erweiterungsideen in meinem Programm umgesetzt. Zusätzlich implementierte ich die Möglichkeit einer Sensitivitätsanalyse, mit deren Hilfe die Resultate besser eingeschätzt werden können. In diesem Kapitel will ich diese Erweiterungen einzeln vorstellen. Manche von ihnen führten dazu, dass auch grundsätzliche Teile des Modells angepasst werden mussten, wie z. B. die Distanzberechnungen. Diese Änderungen werden ebenfalls in diesem Kapitel thematisiert.

3.1 Treppen

In der Beschreibung des F.A.S.T.-Modells werden Treppen als eine von vielen Möglichkeiten vorgestellt, um die Ergebnisse realistischer zu gestalten [Kre06, S. 30 ff.]. Sie sollen „das Grundverhalten der Agenten nicht beeinflussen“, sondern dienen lediglich zur Verbesserung der Realitätsnähe von Szenarien. Bis jetzt wurden Wandzellen, Ausgangszellen und normale Zellen (durch Agenten besetzbare Zellen, ohne spezielle Eigenschaften) vorgestellt; nun erweitern Treppenzellen das Modell.
Eine Treppenzelle wirkt sich auf die Geschwindigkeit der Agenten aus. In [Kre06, S. 30 f.] wird von einer Reduktion um ca. 50% gesprochen, basierend auf vorhandenen ([Wei92]) und selbst durchgeführten Messungen ([Kre06, S. 103-110]). Die Geschwindigkeitsänderung eines Agenten tritt ein, wenn er zu Beginn einer Runde eine Treppenzelle besetzt. Der kleinste Wert, den die Agentengeschwindigkeit durch eine Treppenzelle annehmen kann, ist allerdings 1. Würde diese Untergrenze nicht existieren, könnte sich ein Agent, dessen Geschwindigkeit durch die Treppenzelle 0 erreicht, nie mehr von ihr wegbewegen.
Die Eigenschaften einer Treppenzelle, die im Modell vorgegeben sind, wurden zum Teil geändert, um eine Flexibilisierung des Einsatzes des Zelltyps zu erreichen: Die starre Reduktion der Agentengeschwindigkeiten um 50% wurde aufgehoben und stattdessen ein neuer Parameter eingeführt. Dessen beliebig setzbarer Wert steuert die Veränderung der Agentengeschwindigkeit. Zusätzlich ist es möglich, jeder Treppenzelle einen individuellen Wert zuzuweisen. Ich nenne diesen Parameter f_{stair}. Dieser Faktor wird mit der aktuellen Agentengeschwindigkeit (v_{cur}) beim Betreten von Treppenzellen multipliziert. Für $0 \leq f_{stair} < 1$ wird v_{cur} reduziert,

allerdings mit der Untergrenze: $v_{cur} \geq 1$. Für $f_{stair} = 1$ wird die Geschwindigkeit nicht beeinflusst, während es für $f_{stair} > 1$ zu einer Beschleunigung des Agenten kommt; auch über dessen Maximalgeschwindigkeit v_{max} hinweg. Sobald ein Agent die beschleunigenden Treppenzellen verlässt, ist auch der Effekt zu Ende und der Agent wird in seinen Bewegungen wieder von seiner Maximalgeschwindigkeit begrenzt.
Die Vorteile des dynamischen Parameters sind folgende: Auf Grund der freien Wahl des Wertes, kann von der starren Interpretation einer „Treppe“ abgewichen werden. Treppenzellen könnten beispielsweise eine Notfallrutsche ($f_{stair} \gg 1$), eine Rolltreppe ($f_{stair} > 1$), einen Erdwall ($f_{stair} < 1$) oder eine Leiter ($f_{stair} \ll 1$) simulieren. Und da es möglich ist für einzelne oder Gruppen von Treppenzellen die Parameter individuell zu setzen, kann es die eben aufgezählten „Treppenobjekte“ gleichzeitig in einem Szenario geben.

3.2 Dynamische Wandparameter

Es gibt einen Einfluss benachbarter Wände auf die Wahrscheinlichkeit einer Zelle bei der Zielzellensuche (siehe 2.3.2). Man kann dabei von einem „Sicherheitsabstand“ [Kre06, S. 27] reden, den die Agenten zwischen sich und einer Wand einhalten wollen. Wie groß dieser Abstand ist, regelt zum einen die globale Variable W_{max}, aber auch die individuelle Agentenkonstante κ_W. Das Problem der ersten Variablen ist, dass sie grundsätzlich für jeden Agenten gilt. Die zweite Variable hat das Problem, dass sie nicht unterscheidet, um welche Art von Wandzelle es sich handelt.
Erweiterungen sollen dazu dienen, dass die Szenarien und damit auch die Ergebnisse realistischer werden. Zu einem realistischen Szenario gehören für mich unter anderem auch verschiedene Arten von Wänden. Um nur ein paar Beispiele zu nennen: Dünne, halbhohe Bürotrennwände, Glasfassaden, Balkongeländer, Hecken, Zäune und viele andere mehr. Grundsätzlich haben alle Wandzellen gemein, dass sie als einziger Zelltyp nicht von Agenten besetzt werden können. Diese Forderung eröffnet Spielraum für weitere Interpretationen: Auch Autos, Schränke, Stühle und andere Gegenstände lassen sich mittels Wandzellen realisieren. Sie müssen nur groß genug sein, um in das Zellenschema zu passen und unbeweglich genug, um ihre starre Position zu rechtfertigen.
Meine Erweiterung ist ein dynamischer Parameter für Wandzellen (f_{wall}). Er verändert die Auswirkung der Teilwahrscheinlichkeit p_W benachbarter Wände. Somit lassen sich die verschiedenen Typen von Wänden realisieren, die unterschiedlichen Einfluss auf das Verhalten eines Agenten haben sollen. Der Parameter fließt direkt in die Formel (2.7) ein und verändert sie zu folgender, neuer Formel:

$$p_{xy}^{W} = e^{-\kappa_W W_{xy} f_{wall}}. \tag{3.1}$$

Für $f_{wall} < 1$ wird p_W verstärkt; die entsprechende Zelle also attraktiver gemacht. Für $f_{wall} > 1$ wird sie unattraktiver für einen Agenten.
Während κ_W individuell für den Agenten gilt, egal um welche Wand es sich handelt, ist es bei f_{wall} genau umgekehrt: Hier kann jede Wandzelle einen anderen Faktor bekommen; dieser wirkt sich auf jeden Agenten gleichermaßen aus. Der Standardparameter W_{max} ist hiervon

allerdings nicht betroffen. Ist ein Agent weiter als W_{max} von einer Wandzelle entfernt, gilt immer $p_W = 1$, unabhängig vom Wandzellenfaktor.

3.3 Mehrere Stockwerke

Eine weitere Idee von [Kre06, S. 32] ist die Einführung weiterer Stockwerksebenen. Eine notwendige Erweiterung des Simulationsgebiets in die dritte Dimension (ein „diskretisiertes Volumen“[Kre06, S. 32]), kam aus damaliger Sicht nicht in Frage, da es „zu teuer im Hinblick auf den Arbeitsspeicherverbrauch“ sei. Der Einsatz meiner Software auf Tablet-PCs lässt dieses Argument weiterhin bestehen (siehe 4.1). Die Erweiterung des Simulationsgebiets um zusätzliche Stockwerke, ist auch im zweidimensionalen Raum durchführbar. Die Idee von F.A.S.T. ist es, bestimmte Stockwerks-Ausgangszellen zu platzieren, die einen Agenten eine Ebene nach oben oder unter befördern und Stockwerks-Eingangszellen, die - als Gegenpart dazu - Agenten in die entsprechende Ebene aufnehmen. Weitere Details, z. B. wie man sich eine Ebene vorzustellen hat, sind nicht genannt. Ich habe mich für diese Erweiterung in meiner Implementierung entschieden. Allerdings ist die Umsetzung eine etwas andere, als vom Modell vorgeschlagen; auch bedingt durch die knappen Ausführungen in [Kre06]. Meine Idee ist folgende:
Das Simulationsgebiets wird in verschiedene Bereiche aufgeteilt. Ein Bereich entspricht somit einem Stockwerk oder einer Ebene, um es allgemein zu formulieren. Zuvor hat eine Ebene das gesamte Gebiet eingenommen. Nun kann es beispielsweise sein, dass das erste Stockwerk den linken, oberen Bereich einnimmt, das zweite Stockwerk den rechten, unteren und die dritte Ebene den linken, unteren Bereich. Ein viertes Stockwerk existiert nicht und somit bleibt der rechte, obere Bereich ungenutzt. Eine Ebene ist so konzipiert, dass man sie nur durch bestimmte Zellen erreichen und verlassen kann. Diese Verbindungszellen erhielten von mir den Namen "Teleportzellen". Das hat folgenden Hintergrund: Sieht man sich das gesamte Zellenfeld an, stellt man fest, dass ein Agent beim Betreten einer Teleportzelle, direkt in einen anderen Teil des Gebiets „teleportiert“ wird, auch wenn er sich theoretisch gar nicht bewegt hat. Eine Teleportzelle hat immer genau eine Partnerzelle (ebenfalls vom Typ „Teleportzelle“). Bewegt sich ein Agent über eine Teleportzelle, wird er zu deren Partnerzelle verschoben, ungeachtet seiner Geschwindigkeit. Die Distanz zwischen den beiden Zellen ist mit Null definiert, da sie im Grunde genommen denselben realen Platz einnehmen. Ein Agent kann somit **innerhalb einer Bewegung**, über diese Zellen, Stockwerke betreten und/oder verlassen. Durch die Nulldistanz zwischen den verbundenen Teleportzellen, fließen bei der Zielzellensuche des Agenten die Zellen des neuen Stockwerks direkt mit ein. Kann ein Agent folglich die Teleportzelle seines aktuellen Stockwerks erreichen, kann er auch die Teleportzelle des anderen Stockwerks erreichen.
Die Einführung der Teleportzelle als neuen Zellentyp, führt viele weitere Veränderungen mit sich. Vor allem die Distanzberechnungen und die Zielzellensuche musste ich dementsprechend anpassen. Für Details dazu siehe Kapitel 3.5 und 3.6.

3.4 Sensitivitätsanalyse

Neben den bereits beschriebenen Erweiterungen, welche die Ergebnisse einer Simulation realistischer gestalten sollen, habe ich mich dazu entschlossen, auch eine Erweiterung im Hinblick auf die Ergebnisanalyse in meine Applikation einzubauen. Es handelt sich dabei um die Möglichkeit einer Sensitivitätsanalyse. Mit ihr ist es möglich, die Auswirkungen von Parameteränderungen am Ergebnis nachzuverfolgen. Das kann zum einen Aufschluss darüber geben, wie gut die Parameter für das simulierte Szenario gewählt wurden, zum anderen zeigt es aber auch, wie robust das Simulationsgebiet für andere Situationen ausgelegt ist. Es kann z. B. sein, dass ein Stadion sehr schnell evakuiert werden kann, wenn sich alle Personen mit gleicher, langsamer Geschwindigkeit zu den Ausgängen bewegen. Wenn es allerdings größere Schwankungen in den Geschwindigkeiten gibt, kann es zu Staus und damit Verzögerungen kommen. Diese Schwankungen können mit der Sensitivitätsanalyse erzeugt werden. Der resultierende Evakuierungsgraph und andere Ergebnisse geben daraufhin Aufschluss darüber, wie gut oder schlecht diese Störungen von den Fluchtwegen aufgefangen werden können.
Die Sensitivitätsanalyse ist ein optionaler Teil der Simulation. So kann der Benutzer wählen, ob er das Szenario im Demonstrationsmodus („Demo-Modus“) simulieren lassen möchte, oder mit der Sensitivitätsanalyse verknüpfen möchte (sogenannter „Experten-Modus“). Die Analyse wird konfiguriert, indem eine beliebige Anzahl an Regeln definiert wird. Vor jedem Simulationsdurchlauf werden diese auf die entsprechenden Parameter angewendet, so dass neue Parameterwerte für jeden Durchlauf zur Verfügung stehen. Um eine genaue Analyse der Ergebnisse zu gewährleisten, werden alle Regelanwendungen und Parameterwahlen in einer Datei dokumentiert. Grundsätzlich können mit Hilfe der Regeln fast alle Simulationsparameter verändert werden: Die Entfernungsgrenze von Wandzellen für den entsprechenden Einfluss auf die Zielzellensuche (W_{max}), die Parameter α und δ zum Ändern des dynamischen Floor-Fields, sämtliche Agentenparameter, sowie die Faktoren einzelner Treppen- oder Wandzellen. Eine Regel besteht selbst aus verschiedenen Parametern die vom Benutzer bei ihrer Erstellung gesetzt werden. Sie sind nachfolgend aufgelistet:

- **Typ:** Eine erste grobe Filterung, um welche Gruppe von Parametern es sich handelt. Hier hat der Benutzer die Wahl zwischen „Allgemein“, „Agent“, „Treppe“ und „Wand“
- **Gruppe:** Außer für den Typ „Allgemein“ gibt dieser Wert an, für welche Gruppe von Objekten die Regel gelten soll. Bei Agenten bezieht sich die Gruppe auf die in der Konfigurationsdatei (4.3.1) definierten Gruppen. Bei Treppen und Wänden werden die Einzelobjekte nach ihren Farbwerten gruppiert (4.3.2).
- **Wahrscheinlichkeitsverteilung:** Die Wahl des Parameters hängt von der verwendeten Wahrscheinlichkeitsverteilung ab. Hier hat der Benutzer die Wahl zwischen vier Funktionen: Gleichverteilung, Normalverteilung, log-Normalverteilung und Dreiecksverteilung (auch Simpson-Verteilung).
- **Funktionsparameter:** Sie geben der Funktion der Wahrscheinlichkeitsverteilung vor, in welchem Bereich ein Wert gewählt werden soll. Für die Gleichverteilung muss ein Minimal- und Maximalwert angegeben werden. Für die beiden Normalverteilungen muss der Erwartungswert und die Varianz angegeben werden. Die Dreiecksverteilung

benötigt sowohl einen Minimal- und Maximalwert, als auch einen dazwischenliegenden „wahrscheinlichsten“ Wert, der sozusagen die Spitze des Dreiecks bildet.

Die Abbildungen 3.1 bis 3.4 zeigen die Dichte- und Verteilungsfunktion aller vier verwendeten Wahrscheinlichkeitsverteilungen.

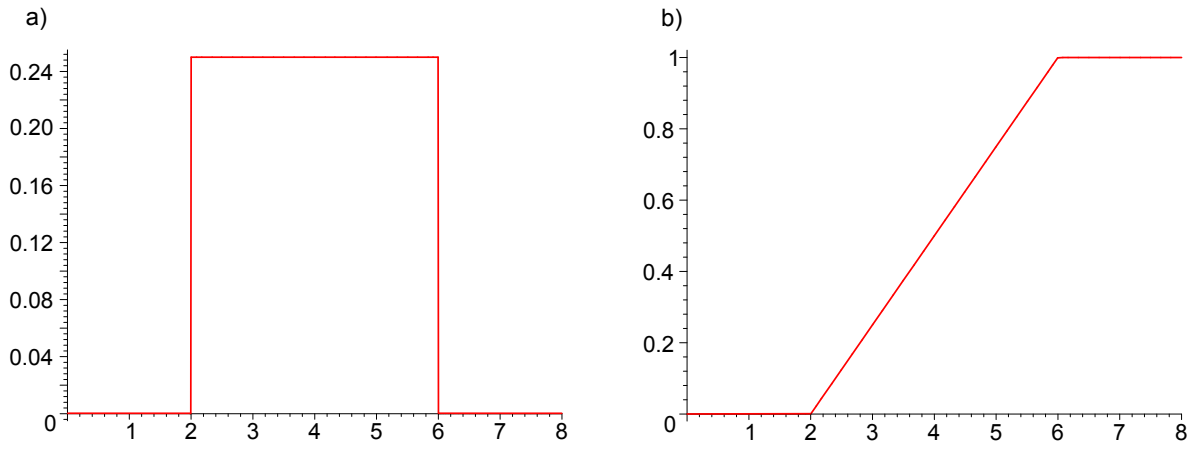


Abbildung 3.1: Gleichverteilung. Dichtefunktion (a) und Verteilungsfunktion (b) für $min = 2$ und $max = 6$.

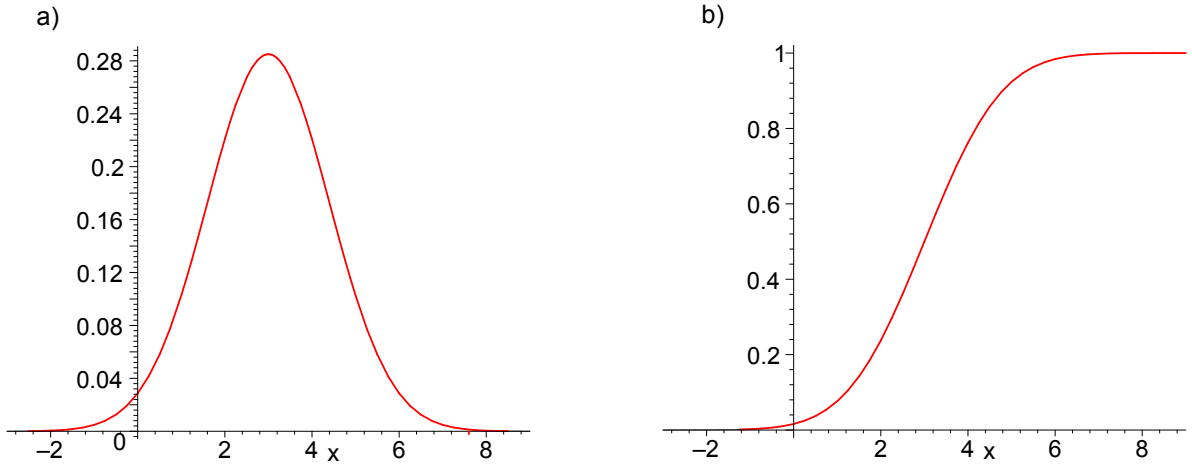


Abbildung 3.2: Normalverteilung. Dichtefunktion (a) und Verteilungsfunktion (b) für Erwartungswert $\mu = 3$ und Varianz $\sigma = 1,4$.

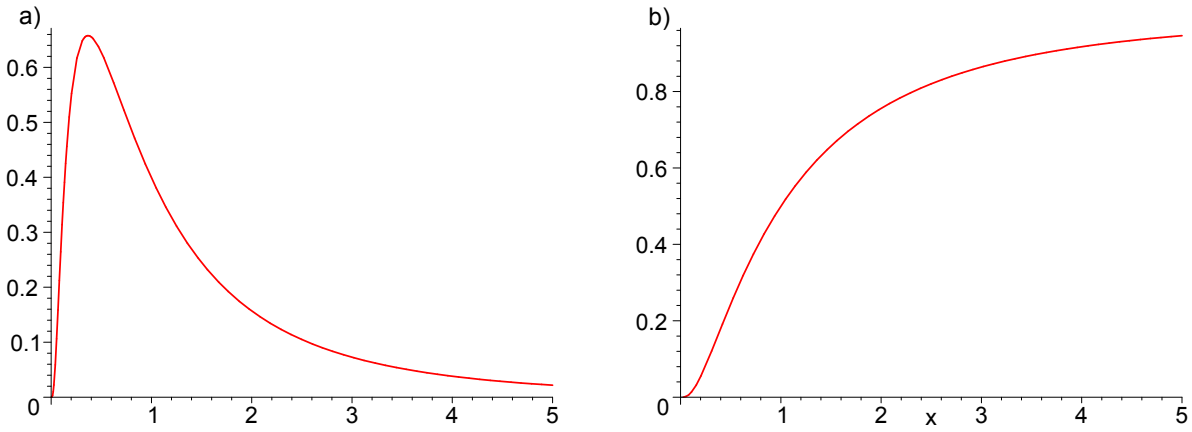


Abbildung 3.3: Logarithmische Normalverteilung. Dichtefunktion (a) und Verteilungsfunktion (b) für Erwartungswert $\mu = 0$ und Varianz $\sigma = 1$.

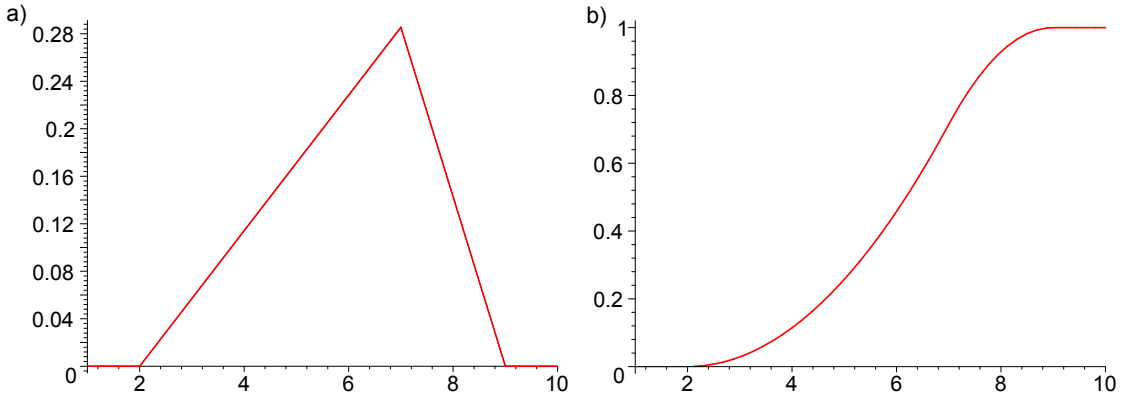


Abbildung 3.4: Dreiecks- oder Simpsonverteilung. Dichtefunktion (a) und Verteilungsfunktion (b) für $min = 2$, $max = 9$ und dem wahrscheinlichsten Wert bei 7.

3.5 Modifizierte Distanzberechnungen

In Kapitel 2.2.1 wurde bereits die Wichtigkeit des statischen Floor-Fields angesprochen. In jeder Runde werden für die Ausgangs- und Zielzellensuche jedes Agenten sehr häufig Distanzen zu Ausgangszellen benötigt. Würde man diese Berechnungen jedes Mal neu durchführen, verlängert sich die Zeit für die Durchführung einer Simulation sehr stark. Da Zellen mit Ausnahme der Agenten ihre Positionen nicht verändern, bleiben auch die Distanzen untereinander und zu den Ausgangszellen, über die gesamte Simulationsdauer hinweg, konstant. Deshalb ist das statische Floor-Field von so entscheidender Bedeutung im Hinblick auf die Zeitersparnis. Damit die Zeit, die am Anfang zur Berechnung des statischen Floor-Fields benötigt wird,

möglichst kurz gehalten wird, sind bereits in [Kre06, S. 20 ff.] viele Lösungen vorgeschlagen. Einige davon habe ich in meiner Implementierung übernommen.
Die Hinzunahme der Teleportzellen erfordert zusätzliche Veränderungen an den Distanzberechnungen. In den folgenden Abschnitten gehe ich deshalb auf die einzelnen Algorithmen ein und erkläre, was ich an ihnen im Vergleich zum vorgeschlagenen Algorithmus des F.A.S.T.-Modells verändert habe. In Kapitel 4.3.3 gibt es weitere Details zur Implementierung dieser Algorithmen.

3.5.1 Grundidee

Bevor es um die Modifizierungen der einzelnen Algorithmen geht, soll dargestellt werden, welcher Algorithmus grundsätzlich hinter der Berechnung der Distanzen steckt und wie ich ihn für meine Implementierung aufgeteilt habe.
Das Hauptziel des Algorithmus [Kre06, S. 20 ff.] ist die Berechnung der kürzesten Entfernung aller Zellen zu allen Ausgangszellen. Als Nebeneffekt erhalten die normalen Zellen auch Distanzen zwischen sich und sogenannten Knotenzellen. Darauf gehe ich gleich genauer ein. All diese Informationen werden letztlich im statischen Floor-Field gespeichert, so dass ein schneller Zugriff während des Simulationsverlaufs garantiert wird. In [Kre06] wird Dijkstras Algorithmus [Dij59] dazu verwendet. Normalerweise bedeutet dies, dass man aus dem Simulationsgebiet einen Graphen erstellen müsste. Jede Zelle ist dabei ein Knoten. Eine Kante wird dort hinzugefügt, wo sich zwei Zellen gegenseitig „sehen“ können. Mit „sichtbar“ ist hier gemeint, dass auf der geraden Linie zwischen den beiden Zellen keine Wandzelle liegt. Dieser sogenannte „Sichtbarkeitsgraph“ dient als Ausgangspunkt für den Algorithmus. Nach [Kre06] gibt es eine effizientere Möglichkeit, den Algorithmus einzusetzen: Anstatt aus jeder Zelle einen Knoten zu machen, reicht es, sich auf bestimmte „Knotenzellen“ zu reduzieren. Es muss allerdings sichergestellt werden, „dass jede Zelle, von mindestens einer korrekt gewählten Knotenzelle aus, sichtbar ist“[Kre06, S. 21]. Die Bestimmung dieser Knotenzellen ist in Kapitel 3.5.2 und 4.3.3 beschrieben. Sind alle notwendigen Knotenzellen gefunden, ist garantiert, dass jede Nicht-Knotenzelle von mindestens einem Knoten aus sichtbar ist. Versucht nun ein Agent sich auf kürzestem Weg um eine Wandzelle zu bewegen, existiert an jeder Stelle, an der der Agent seine Richtung ändern muss, eine Knotenzelle. Somit enthält der Sichtbarkeitsgraph der Knotenzellen die kürzesten Wege vorbei an Hindernissen (Wänden) [Kre06, S. 22]. Die Distanzberechnungen werden damit in zwei Teile gespalten: Zuerst werden die kürzesten Entfernungen zwischen den Knotenzellen und den Ausgängen berechnet. Danach ergibt sich die kürzeste Entfernung einer Nicht-Knotenzelle zu einer Ausgangszelle als die kleinste aller möglichen Distanzen über seine sichtbaren Knotenzellen, d. h. (Distanz zur Knotenzelle) + (Distanz der Knotenzelle zur Ausgangszelle).
Der gesamte Algorithmus wird bei mir in verschiedene Teilalgorithmen aufgespalten:

1. Die Knotenzellen werden bestimmt.
2. Alle Zellen werden mit ihren sichtbaren Knotenzellen verknüpft, d. h. jede Zelle speichert intern die Referenz und Distanz zu allen sichtbaren Knotenzellen ab.

3. Die kürzesten Distanzen zwischen allen Knotenzellen und allen Ausgangszellen werden berechnet.
4. Für jede Nicht-Knotenzellen wird ihre Entfernung zu allen Ausgangszellen berechnet.

3.5.2 Bestimmung der Knotenzellen

Ziel des ersten Teilalgorithmus ist es, die Knotenzellen korrekt zu bestimmen. Um die optimale, d. h. minimale, Anzahl von Knotenzellen zu bestimmen, müsste man eine Analyse der Geometrie des Gebiets durchführen [Kre06, S. 21]. Daraus entsteht ein minimaler Sichtbarkeitsgraphen, bei dem „die Knoten die konvexen Ecken der Hindernisse sind“. Hier schlägt [Kre06] eine Idee für eine einfachere Umsetzung vor: Nimmt man eine leicht größere Zahl von Knotenzellen in Kauf, könne man in diesem Fall viel schneller vorgehen. Es genüge, immer nur die direkten Nachbarzellen jeder Wandzelle zu untersuchen. Der Zelltypen aller Nachbarzellen (normale oder Wandzellen) entscheiden darüber, welche der normalen Nachbarzellen Knotenzellen sein können[Kre06, S. 21]. Zusätzlich wird vorgeschlagen, das Simulationsgebiet mit weiteren Voraussetzungen zu versehen. Es sollen Wandzellen verboten werden, die ausschließlich diagonal mit mindestens einer weiteren Wandzelle verbunden sind. Dies führt zu inkorrekt platzierten Knotenzellen. Meiner Ansicht nach ist diese Forderung vor allem im Hinblick für die Zielzellensuche wichtig. Ansonsten könnte sich ein Agent durch die „Lücke“ in der Wand bewegen, die in der Realität natürlich nicht vorhanden ist. Auch wenn dieses Vorgehen nicht zu Fehlern in der Simulation führt, widerspricht es dem zu erwartenden Verhalten der Agenten. Dies sollte man beim Erstellen der Wandzellen eines Szenarios immer bedenken.
Mit diesen Vereinfachungen und Forderungen im Hintergrund, erhält man einen einfachen und schnellen Algorithmus, um die Knotenzellen zu bestimmen. Seine Implementierung habe ich in 4.3.3 beschrieben und ist auch in [Kre06, S. 21 f.] nachzuschlagen. Eine Erweiterung von mir ist folgende: Zusätzlich zu den Knotenzellen, die mit diesem Algorithmus gefunden werden, füge ich alle Teleportzellen der Menge hinzu. Der oberen Definition nach, sieht eine Teleportzelle ihre Partnerzelle normalerweise nicht, da meistens Wände zwischen den Ebenen vorhanden sind (vgl. 3.3). Für die Zellen einer Ebene ohne Ausgangszelle gilt dann: Ohne verbindende Knotenzelle, haben sie keine Entfernung zu den Ausgangszellen. Das ist nicht möglich. Um zu garantieren, dass für jede Ebene die Distanzen zu allen Ausgangszellen berechnet werden können, müssen die Teleportzellen zu Knotenzellen gemacht werden.

3.5.3 Verknüpfung sichtbarer Knotenzellen

Dieser Teilalgorithmus soll dafür sorgen, dass jede Zelle eine Liste von sichtbaren Knotenzellen samt Distanzen zu ihnen enthält. Das Modell bietet keinen Hinweis, wie dies bewerkstelligt werden soll. Deshalb beschreibe ich meinen Algorithmus an dieser Stelle etwas ausführlicher. Konkrete Details zur Implementierung in Java finden sich in Kapitel 4.3.3.
Das Ergebnis des Algorithmus ist im Prinzip ein Sichtbarkeitsgraph. Die Zellen bzw. Knoten, die sich gegenseitig sehen, sind mit einer Kante verbunden. Die Kantengewichte entsprechen

den Distanzen zwischen den Zellen. Die Ebenen des Simulationsgebiets sind Teilgraphen, die wiederum durch die Teleport-Knotenzellen miteinander verbunden sind. Der Algorithmus wird deshalb eine Ebene nach der anderen bearbeiten. Das hat auch den Grund, dass in einer separaten Menge Referenzen auf die Randzellen der Ebene gespeichert werden und diese Menge möglich klein sein sollte, damit der Algorithmus effizient arbeitet. Eine Randzelle ist hier eine Zelle, die auf mindestens einer Seite den Rand der Ebene bildet. Der Algorithmus iteriert über die Knotenzellen der Ebene. Von jeder Knotenzelle ausgehend, wird ein „Strahl“ zu jeder Randzellen geschossen. Ein für diesen Einsatz angepassten Bresenham-Algorithmus [Cun], untersucht zellenweise die Linie von Knoten- zu Randzelle. Sollte eine Wandzelle erreicht werden, bricht er ab. Bei jedem Erreichen einer Nicht-Wandzelle, wird ihr das aktuelle (*Knotenzelle*, *Distanz*)-Paar hinzugefügt. Nachdem alle Randzellen „beschossen“ wurden, ist eindeutig bestimmt, welche Zellen diese Knotenzelle sieht und welche nicht. Wurde über alle Knotenzellen gezählt, existiert diese Information auch für alle weiteren Zellen der Ebene. Der Teilgraph ist damit komplett abgearbeitet und der Algorithmus geht zur nächsten Ebene über.

3.5.4 Kürzeste Distanzen zu Ausgangszellen

Der Vollständigkeit halber seien hier noch die beiden Teilalgorithmen zusammengefasst, die letztendlich die kürzesten Distanzen aller Knoten- und Nicht-Knotenzellen zu den Ausgangszellen berechnen. Sie orientieren sich allerdings vollständig an den Ideen von [Kre06] sowie des Dijkstra-Algorithmus und besitzen keine Modifikationen von mir.
Für jede Ausgangszelle wird der Algorithmus von Dijkstra gestartet. Die Ausgangszelle ist dabei jeweils der Startknoten. Die restlichen Knoten des zu durchlaufenden Graphen, sind die Knotenzellen. Auf die Funktionsweise des Dijkstra-Algorithmus möchte ich hier nicht näher eingehen; es sei auf entsprechende Literatur, z. B. [Dij59], verwiesen. Nachdem er für jede Ausgangszelle abgeschlossen ist, kennen bereits alle *Knoten*zellen ihre kürzesten Distanzen zu jeder Ausgangszelle.
Der letzte Teilalgorithmus sorgt dafür, dass auch alle Nicht-Knotenzellen die kleinsten Distanzen zu den Ausgangszellen bekommen. Für jede Ausgangszelle, iteriert er über alle Nicht-Knotenzellen des gesamten Simulationsgebiets. Die minimale Distanz einer solchen Zelle zur Ausgangszelle berechnet sich dann folgendermaßen: Bilde jeweils die Summe aus *(Distanz von Zelle zu einer ihrer sichtbaren Knotenzellen)* und *(Distanz dieser Knotenzelle zur Ausgangszelle)*. Das Minimum aus allen Einzelsummen ist gleichbedeutend mit der geringsten Entfernung der Zelle zur Ausgangszelle. In Kapitel 4.3.3 und dem darauffolgenden gibt es weitere Details zur ihrer Implementierung.

3.6 Modifizierte Zielzellensuche

Das Ziel des Algorithmus ist es, alle Zellen zu bestimmen, die vom Agenten in der aktuellen Runde erreicht werden können. Die Menge der Zellen hängt dabei von seiner derzeitigen Position und aktuellen Geschwindigkeit ab.

Da die Agentengeschwindigkeit eine entscheidende Rolle bei der Suche nach Zielzellen hat, folgen einige wichtige Definitionen. Sie weichen zum Teil von den im Modell beschriebenen ab. Das hat zum einen den Grund, dass die Erweiterung der Treppenzellen eine Anpassung der neuen Geschwindigkeit nötig macht, zum anderen die Geschwindigkeit hier grundsätzlich anders berechnet wird, als im Modell vorgeschlagen.

- v_{max} ist die maximale Geschwindigkeit, die ein Agent von alleine erreichen kann. Sie ist ein individueller Agentenparameter.
- v_{cur} ist die aktuelle Geschwindigkeit eines Agenten. Für sie gilt im Normalfall: $1 \leq v_{cur} \leq v_{max}$. Ist der Geschwindigkeitsfaktor einer Treppenzelle größer als 1, kann dies einmalig dazu führen, dass $v_{cur} > v_{max}$ ist, sollte der Agent zu Beginn einer Runde auf ihr stehen und $f_{stair} > 1$ sein.
- $\widetilde{v}_{next} = \max(1, (\min(v_{cur} + 1, v_{max})) \cdot f_{stair})$ bezeichnet die Geschwindigkeit des Agenten zu Beginn der Zielzellensuche.
- $v_{next} = \left\| \begin{array}{c} \Delta x_{t+1} \\ \Delta y_{t+1} \end{array} \right\|$ ist die tatsächliche Geschwindigkeit des Agenten bei einer Bewegung zur Zelle (x, y), mit der Differenz der Position der Zielzelle und des Agenten (Δx_{t+1} bzw. Δy_{t+1}). Da die Geschwindigkeit eines Agenten ganzzahlig ist, wird der entsprechende Wert gerundet: $v_{next} = \lfloor v_{next} + 0,5 \rfloor$.

Zu Beginn des Algorithmus wird geprüft, ob sich der Agent auf einer Treppenzelle befindet. Ist dies der Fall, wird seine Geschwindigkeit entsprechend des Faktors f_{stair} der Treppenzelle angepasst (siehe Formel für $\widetilde{v}_{next}$). Dieser Wert entspricht der maximalen Distanz, die eine potenzielle Zielzelle zur Startzelle haben darf.
Die Stockwerkserweiterung im F.A.S.T.-Modell sieht vor, dass Agenten, die am Ende einer Runde auf einer Stockwerks-Ausgangszelle stehen, eine Ebene nach oben oder unten transportiert werden [Kre06, S. 32]. Dies handhabe ich in meiner Implementierung anders: Agenten können sich **innerhalb einer Runde** über Stockwerksgrenzen hinaus bewegen. Dies erfordert allerdings einen komplexeren Algorithmus zum Finden der gültiger Zielzellen. Er funktioniert nach folgendem Prinzip: Ausgehend von der Startzelle des Agenten, werden stufenweise die direkten Nachbarzellen untersucht. Sollten die Zellen vom Agenten aus sichtbar sein, wird die euklidische Distanz zwischen ihnen berechnet und in der Nachbarzelle zwischengespeichert. Ist die betrachtete Zelle nicht sichtbar, wird die Distanz von ihr zur letzten sichtbaren Zelle, über welche die Suche zu ihr gelangt ist, berechnet und auf die Distanz jener sichtbaren Zelle zur Startzelle addiert. Mit dieser Technik werden Hindernisse (sprich: Wandzellen) auf kürzestem Weg umgangen. Der, nach außen hin steigende, Distanzwert ist gleichzeitig das Abbruchkriterium für die Suche. Sollte eine Zelle eine größere Distanz zur Startzelle aufweisen, als $\widetilde{v}_{next}$, wird sie nicht in die Menge der zu untersuchenden Zellen aufgenommen. Auf Grund dessen, dass einmal besuchte Zellen für die weitere Suche ignoriert werden, kann es zu keinen Endlosschleifen oder Umwegen kommen. Irgendwann ist die Menge der zu untersuchenden Zellen leer und die Suche endet. Der Algorithmus baut gewissermaßen einen Sichtbarkeitsgraphen auf, wendet allerdings noch im Aufbau den Dijkstra-Algorithmus darauf an.
Eine Besonderheit bilden nun die Teleportzellen. Die Distanz zwischen einer Teleportzelle

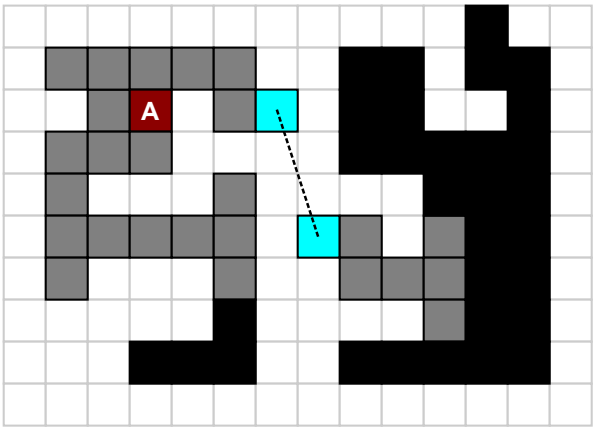


Abbildung 3.5: Ergebnis der Zielzellensuche für Agent auf Zelle A: Seine eigene Zelle, die beiden türkisfarbenen Teleportzellen und alle grauen Zellen kann der Agent mit seiner Geschwindigkeit $v = 7$ in dieser Runde erreichen. Weiße Zellen sind Wandzellen, schwarz sind normale, aber für den Agenten in dieser Runde unerreichbare Zellen.

und ihrer Partnerzelle ist Null (siehe 3.3). Wird nun eine Teleportzelle vom Algorithmus in die Menge der zu betrachtenden Zellen aufgenommen, wird ihre Partnerzelle ebenfalls hinzugefügt. Der Algorithmus merkt, wenn die Distanz zwischen zwei verknüpften Teleportzellen berechnet wird und addiert somit 0 auf den Distanzwert. Damit ist es problemlos möglich auch Zellen auf anderen Stockwerken als Zielzellen zu erkennen, obwohl sie auf Grund ihrer absoluten Positionswerte weit voneinander entfernt liegen. Abbildung 3.5 zeigt das Ergebnis einer Zielzellensuche mit $\widetilde{v}_{next} = 7$ von Zelle A aus startend. Die grauen Zellen wurden in die Menge der möglichen Zielzellen aufgenommen.
Das dynamische Floor-Field (2.3.2), sowie die Berechnung des Einflusses der Trägheit (2.3.2) benötigen einen Vektor mit der relativen Bewegung des Agenten in x- und y-Richtung. Die Zielzellensuche summiert deshalb nicht nur den Distanzwert auf, sondern auch den Bewegungsvektor. Da die Distanz zwischen verknüpften Teleportzellen Null beträgt, wird der Nullvektor auf den Bewegungsvektor addiert. Der Distanzwert entspricht der euklidischen Norm des Bewegungsvektors. Es würde somit ausreichen, nur den Bewegungsvektor zu speichern. Da der Betrag allerdings an vielen Stellen des Programms benötigt wird, speichern ihn die Zelle gesondert ab und verzichten auf ein regelmäßiges Neuberechnen. Der zusätzliche Speicherplatzverbrauch ist im Vergleich zu der benötigten Rechenzeit vernachlässigbar klein.
Am Ende des Algorithmus existiert eine Menge von Zellen. Jede von ihnen könnte theoretisch vom Agenten in dieser Runde besetzt werden. Die Zuweisung der Wahrscheinlichkeiten an die einzelnen Zellen, sowie die tatsächliche Wahl der endgültigen Zielzelle, wird als nächster Schritt vom Programm ausgeführt. Für Details zu diesen beiden Aktionen siehe 4.4.2.

4 Programmaufbau

Während sich Kapitel 3 ausschließlich mit den Erweiterungen und Modifikationen des F.A.S.T.-Modells in meinem Programm befasst hat, widmet sich dieses Kapitel der Implementierung im Detail. Neben der Beschreibung aller Programmteile, will ich auch die Algorithmen, Datenstrukturen und Denkweisen genauer vorstellen, welche die einzelnen Aufgaben der Simulation und des Benutzerinterfaces übernehmen. Neben Pseudocode habe ich deshalb auch Auszüge aus dem echten Java-Programmcode integriert.

4.1 Auswahl der Entwicklungsplattform

Die Implementierung des Frameworks soll auf einem mobilen Tablet-PC lauffähig sein. Die Auswahl des Betriebssystems wurde von mir frühzeitig auf Windows (Microsoft) und Android (Google) reduziert, sowie die möglichen Programmiersprachen auf C++ und Java. Hauptsächlich deshalb, weil ich bereits einige Erfahrung mit beiden Sprachen gesammelt hatte und deshalb einschätzen konnte, dass eine Umsetzung des Projekt mit einer von ihnen gut möglich ist.
An den Tablet-PC wurden verschiedene Ansprüche gestellt: Ein Multicore-Prozessor zur Beschleunigung der intensiven Rechenvorgänge wurde gefordert. Da die Simulation sehr viele Daten bereithalten muss, lag ein besonderes Augenmerk auf dem Hauptspeicher. Seine Kapazität sollte am oberen Limit des derzeit angebotenen liegen, mindestens aber 512 Megabyte. Details, wie die Akkulaufzeit, die Eingabemöglichkeiten oder das Datenspeichervolumen wurden größtenteils vernachlässigt. Bei der Displaygröße wurde lediglich auf eine „übliche“ Größe für Tablet-PCs geachtet (> 7 Zoll in der Bildschirmdiagonalen).
Ich entschied mich schlussendlich für das „Samsung Galaxy Note 10.1 WiFi“ mit Android Betriebssystem (Version 4.1.2) [Sam] Zum Zeitpunkt der Entscheidung waren die Tablet-PCs, auf denen Windows als Betriebssystem laufen, technisch gesehen im Rückstand. Dagegen besitzt das Samsung Tablet einen NVidia Tegra 3 1,4 GHz Prozessor mit 4 Kernen, sowie 2 GB RAM und ein Display mit 10,1 Zoll in der Bildschirmdiagonalen.
Mit dem Entschluss für den Samsung Tablet-PC stand auch das darauf installierte Android als Betriebssystem fest. Um eine „App“ - wie die Anwendungen auf Android-Geräten meistens genannt werden - zu erstellen, wird von der Systemseite aus vorgeschlagen das Android Software-Development-Kit (SDK) zu benutzen, welches unter anderem das „Android Developer Tool“ (ADT) Plugin für Eclipse enthält. Es ist eine vollständige integrierte Entwicklungsumgebung (IDE) für Java für die Programmierumgebung *Eclipse*. Sowohl das ADT, als auch Eclipse werden von Android zum Erstellen und Debuggen von Android-Anwendungen empfohlen. Vor allem das ADT soll vieles vereinfachen [An2].

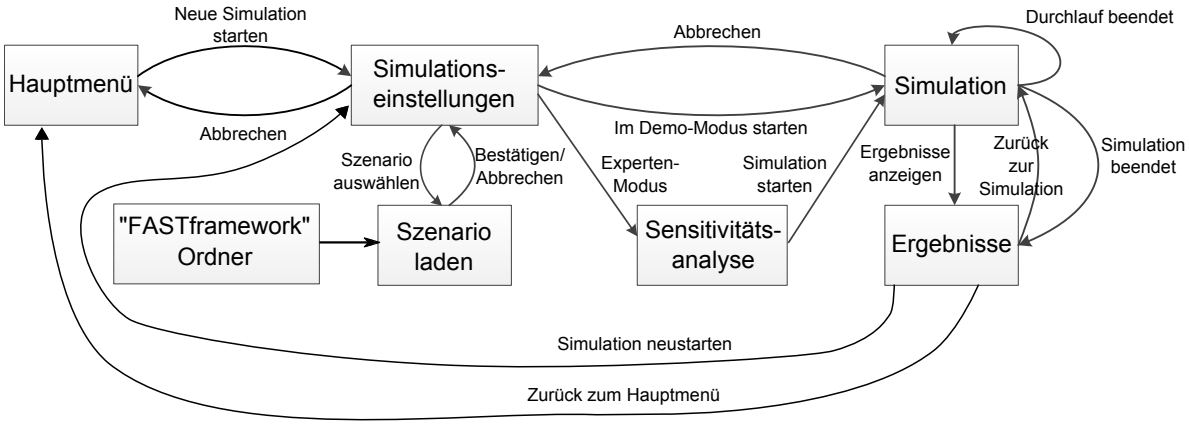


Abbildung 4.1: Übersicht über das Programm. Die Boxen entsprechen den „Aktivitäten“, die beschrifteten Pfeile kennzeichnen Aktionen des Benutzers

Der Einsatz von C++ oder einer anderen Programmiersprache als Java gestaltet sich auf Android-Geräten schwierig. Das hat mehrere Gründe: Zum einen laufen Android-Programme auf einer virtuellen Maschine. Diese akzeptiert nur ihren spezifischen Code, der wiederum praktisch nur durch das Android SDK erstellt und kompiliert werden kann. Um nativen Code in einem Programm unterzubringen, stellt Android das sogenannte NDK (Native Developement Kit) zur Verfügung. Mit dessen Hilfe können Komponenten in nativem Fremdcode unter Android lauffähig gemacht werden. Damit könnte man also C++-Code einbringen. Dies eignet sich aber nur, wenn der Fremdcode sehr hardwarenah und effizient ist, denn die Optimierungen des Compilers und der virtuellen Maschine sorgen auch bei normalem Java-Code für eine effiziente Ausführung. So beschreibt es zumindest Android [An1]. Gleichzeitig nimmt man sich mit dem NDK die Möglichkeit, bestimmte Funktionalitäten der Android API verwenden zu können. Da die Anwendung zum Teil notwendigerweise mit Java geschrieben werden muss und der mögliche C++-Teil, ebenfalls mit Java realisiert werden kann, kam ich zu der Entscheidung alles komplett in Java zu schreiben.

4.2 Übersicht über Programmteile

In diesem Abschnitt will ich einen Einblick in die Struktur und den Aufbau meines Programms geben. Abbildung 4.1 zeigt ein Schema der Programmteile und der Aktionen, die zu einem Wechsel des aktiven Programmteils führen. Im folgenden beschreibe ich ihre Funktionen innerhalb des Programms und welche Interaktion durch den Benutzer möglich sind. Im Android-Sprachgebrauch handelt es sich bei den einzelnen Menüs um „Aktivitäten“. Diesen Begriff werde ich im weiteren Verlauf ebenfalls verwenden. Die „Hauptmenü“-Aktivität ist der Ausgangspunkt des Programms. Neben der Möglichkeit sich Informationen über die Anwendung anzeigen zu lassen, kann man hier eine neue Simulation beginnen und die „App“ beenden. Jede Simulation beginnt in der Aktivität der „Simulationseinstellungen“. Bevor ein

Szenario gestartet werden kann, müssen ein paar Einstellungen vorgenommen werden. Die wichtigste ist die Auswahl eines Simulationsgebiets und der entsprechenden Konfiguration. Dies geschieht in der „Szenario laden“-Aktivität. Eine Liste enthält dort alle XML-Dateien, die das Programm im anwendungsspezifischen Ordner „FASTframework“ findet. Dieser Ordner wird vom Programm angelegt, sollte er noch nichts existieren. Er beinhaltet alle Benutzerdaten: Sowohl die Bild- und Konfigurationsdateien der Szenarien, als auch die gespeicherten Resultate der einzelnen Simulationen. Neue Dateien für Szenarien müssen vom Benutzer in diesen Ordner kopiert werden. Wählt der Benutzer eine Datei in der Liste aus, wird der Inhalt der Konfigurationsdatei eingelesen und analysiert. Handelt es sich um ein gültiges Szenario wird ein Vorschaubild des Simulationsgebiets angezeigt.
Neben der Wahl des Szenarios, ist die Anzahl der Durchläufe eine essentielle Angabe. Sie legt fest, wie oft die Simulation mit den gewählten Parametern durchgeführt werden soll, bevor sie als beendet gilt. Zusätzlich gibt es zwei optionale Einstellungen: Wenn eine **grafische Ausgabe** gewünscht wird, sieht der Benutzer während der Simulation die Agenten, wie sie sich über das aktuell ausgewählte Stockwerk bewegen. Der Benutzer hat die Möglichkeit, sich durch die einzelnen Stockwerke zu schalten. Die zweite Option unterscheidet zwischen einem Demonstrationsmodus (kurz Demo-Modus) und einem Expertenmodus. Der letztgenannte Modus beinhaltet eine Sensitivitätsanalyse (siehe 3.4). In der „Sensitivitätsanalyse“-Aktivität, kann der Benutzer für den eben angesprochenen Expertenmodus die Regeln zur Parameterwahl definieren. Eine Liste speichert die Regeln und zeigt sie für den Benutzer an. Sie können auch einzeln wieder entfernt werden. Über diese Aktivität wird die Simulation auch gestartet.
Die Hauptaktivität ist die „Simulations“-Aktivität. Hier wird der Fortschritt der aktuellen Simulation angezeigt. Hat der Benutzer die grafischer Ausgabe als Option gewählt, wird hier das aktuell von ihm gewählte Stockwerk angezeigt, auf dem die Agenten zu sehen sind. Unabhängig von der grafischen Ausgabe, kann die Simulation in dieser Aktivität jederzeit pausiert oder vorzeitig beendet werden. Sowohl während der Simulation, als auch am Ende, kann der Benutzer die (Zwischen-)Ergebnisse in einer gesonderten Aktivität betrachten: Die „Ergebnis“-Aktivität zeigt eine Reihe verschiedener Statistiken in Form von Graphen und Bildern zur aktuellen Simulation. Sie sind über eine Liste auswählbar. Statistiken, die nur textuell darstellbar sind, werden in einer extra Datei gespeichert und können mit einem externen Textprogramm betrachtet werden. Daneben kann der Benutzer wieder zur Simulation zurückkehren und diese fortsetzen, sofern sie noch nicht beendet ist, oder eine neue starten. Die Rückkehr zum Hauptmenü ist ebenfalls möglich.

4.3 Initialisierung der Simulation

Jede Simulation muss vor dem Start initialisiert werden. Ihre Erstellung gliedert sich in drei bis vier Teile: Zuerst muss die Konfigurationsdatei eingelesen werden. Danach wird das Simulationsgebiet, das durch ein Bild definiert ist, eingelesen und daraus die Zellen erstellt. Für das statische Floor-Field müssen die kürzesten Distanzen aller Zellen zu allen Ausgangszellen berechnet werden. Außerdem muss für jede Zelle die nächstgelegene Wandzelle gefunden werden. Dies geschieht im dritten Teil der Initialisierung. Der vierte und letzte Part wird zusätzlich auch vor jedem einzelnen Durchlauf durchgeführt: Wird die Simulation

im Expertenmodus ausgeführt, müssen vor jedem Durchlauf (und damit auch vor dem ersten) die Regeln der Sensitivitätsanalyse angewandt werden. Dieser Schritt fällt im Demo-Modus weg.
Die folgenden Abschnitte behandeln diese vier Unterteilungen der Initialisierungsphase im Detail.

4.3.1 Konfigurationsdatei

Die Konfiguration einer Simulation besteht aus zwei Teilen: Einer Bilddatei, in der der Aufbau des Simulationsgebiets beschrieben ist (4.3.2) und einer XML-Datei, in der verschiedene Parameter und zusätzliche Informationen zum Gebiet definiert sind: Globale Simulationsparameter, die Agentenkonstanten, die Faktoren der einzelnen Treppen- und Wandzellen, aber auch die Position und Größe der einzelnen Stockwerke, sowie Lage der Teleportzellen und ihrer Partnerzellen. Mit dieser Datei befasse ich mich in diesem Abschnitt.
Das Listing 4.1 stellt eine Beispieldatei dar. Hier ist jeder Parameter exemplarisch mit mindestens einem Wert vertreten. Auch wenn vieles in der Datei selbsterklärend ist, will ich im Folgenden die einzelnen Tags näher erläutern.

- `<FASTframework>`: Dieser umschließende Tag **muss** vorhanden sein, sonst wird die Datei als ungültig bezeichnet und es wird nichts eingelesen.
- `<common>`: Hier sind vier allgemeine Einstellungen gesammelt: `<area>` speichert den relativen Pfad und Dateinamen, der zu dieser Simulation gehörenden Bilddatei. Das Wurzelverzeichnis ist dabei der *FASTframework*-Ordner. `<maxwalldist>` verlangt einen Wert für den maximalen Sicherheitsabstand der Agenten gegenüber von Wänden (2.3.2). Die Tags `<alpha>` und `<delta>` entsprechen den gleichnamigen Variablen zur Änderung des dynamischen Floor-Fields (2.2.2). Sie benötigen einen Wert im Bereich $[0, 1]$.
- `<floors>`: Innerhalb dieses *Tag*s werden die Definitionen der einzelnen Stockwerke gesammelt. Diese sind jeweils in `<floor>`-Tags gespeichert. Zu den Definitionen gehört der *Name des Stockwerks*. Er macht das Stockwerk für den Benutzer identifizierbar. Außerdem muss die Angabe der Position (`<position>`) und Größe (`<dimension>`) definiert sein. Die Position entspricht dem linken oberen Pixel des Stockwerks in der Bilddatei, mit $(0, 0)$ als oberster, linker Pixel. Die Dimension entspricht der Anzahl Pixel des Stockwerks in jeweils beiden Richtungen.
- `<agents>`: Die Definitionen der Agentengruppen werden innerhalb dieses Tags gespeichert. Eine Gruppe wiederum, wird mittels des `<group>`-Tags definiert, deren Anzeigename im Attribut `name` angegeben wird. Die einzelnen Parameter der entsprechenden Agenten sind als Attribute des `<params>`-Tags definiert. Die Attributnamen entsprechen den Agentenkonstanten, bzw. `v_max` und `v_start` der Maximal- bzw. Startgeschwindigkeit der Agenten. Die Zuordnung der Parameter an die Agenten, erfolgt durch den Farbwert, den ein Agent in der Bilddatei erhält. Alle Agenten mit derselben Farbe gehören zur selben Gruppe. Die Farbwerte werden aufsteigend sortiert und nacheinander den hier definierten Gruppen zugewiesen. Das hat den Grund, dass der Ersteller der Bilddatei

mehr Freiräume beim Vergeben der Farben hat und eine Änderung der Farbe nicht unbedingt zu einer Änderung der Konfigurationsdatei führen muss.

- `<teleports>` *(optional)*: Hier werden die Teleportzellen definiert. Zwei Zellen werden jeweils innerhalb eines `<teleport>`-Tags definiert: `<from>` gibt die Position der einen Zelle an, wobei dieselbe Regel gilt, wie bei der Stockwerkskonfiguration: $(0, 0)$ ist der oberste, linke, gültige Pixel. `<to>` gibt entsprechend die Position der Partnerzelle an.
- Das Framework bietet die Möglichkeit der dynamischen Parameterwahl für Treppen- und Wandzellen. Das kann mittels der Sensitivitätsanalyse geschehen, oder direkt hier in der Konfigurationsdatei. `<stairs>` bzw. `<walls>`: Der Wert für das `default`-Attribut

Listing 4.1: Beispiel einer Konfigurationsdatei

```
<?xml version="1.0" encoding="UTF-8"?>
<FASTframework>
 <common> <!-- common settings -->
  <area name="area.png" />
  <maxwalldist value="4.0" />
  <alpha value="0.05" />
  <delta value="0.10" />
 </common>
 <floors> <!-- floor configuration -->
  <floor name="1st floor: Foyer">
   <position x="0" y="0" />
   <dimensions w="50" h="40" />
  </floor>
  <floor name="2nd floor: A&amp;amp;B Electronics">
   <position x="50" y="0" />
   <dimensions w="35" h="70" />
  </floor>
 </floors>
 <agents> <!-- agent configuration -->
  <group name="employees">
   <params v_max = "4" v_start = "1" k_e = "6.0" k_s = "2.5" k_d = "1.0" k_i = "0.4" k_w =
       "0.2" k_p = "0.3" />
  </group>
  <group name="security">
   <params v_max = "6" v_start = "3" k_e = "6.0" k_s = "2.5" k_d = "0.5" k_i = "1.2" k_w =
       "0.01" k_p = "0.0" />
  </group>
 </agents>
 <teleports> <!-- teleportation cells configuration --> <!-- optional -->
  <teleport>
   <from x="49" y="2" />
   <to x="50" y="17" />
  </teleport>
 </teleports>
 <stairs default="0.5"> <!-- stair configuration --> <!-- optional -->
  <id color="0x006600" factor="0.15" />
  <id color="#00FF00" factor="2.5" />
 </stairs>
 <walls default="1.0"> <!-- wall configuration --> <!-- optional -->
  <id color="0xFFFFFF" factor="1.5" />
  <id color="#888888" factor="0.5" />
 </walls>
</FASTframework>
```

entspricht dem Standardfaktor. Er wird allen Treppen- bzw. Wandzellen zugewiesen, für die es keine explizite Definition gibt. Eine solche Definition kann mit dem `<id>`-Tag erfolgen: Das Attribut `color` spezifiziert den Farbwert der Zellen in der Bilddatei und das Attribut `factor` den entsprechenden Wert, der diesen Zellen zugewiesen werden soll. So bekommen z. B. alle Wandzellen, die $0xFFFFFF$ als Farbwert in der Bilddatei bekommen haben, den Faktor $1,5$ zugewiesen, während Treppenzellen, die als Farbwert ♯005400 haben, den Standardfaktor $2,0$ bekommen, da für sie keine Definition vorliegt. Zu beachten ist folgendes: Gültige Werte für das `color`-Attribut sind hexadezimale Zahlen, die mit „0x“ oder „♯“ beginnen und normale dezimale Ganzzahlen. Der Farbwert ist als RGB-Code zu interpretieren.

Alle XML-Dateien, die im „FASTframework“-Ordner liegen, werden in der „Szenario-Laden“-Aktivität in einer Liste aufgeführt. Wählt man einen Eintrag aus, wird die XML-Datei geladen und ausgewertet. Alle eingelesenen Werte werden auf Existenz und Gültigkeit hin überprüft. Sollten wichtige Parameter fehlen, wird das Einlesen gestoppt und eine Fehlermeldung mit entsprechenden Hinweisen ausgegeben. Es kann allerdings nicht alles überprüft werden: Da zu diesem Zeitpunkt lediglich die Existenz der Bilddatei geprüft wurde, das Bild selbst aber noch nicht verarbeitet wurde, kann die Gültigkeit der Stockwerks- und Teleportzellenangaben nicht geprüft werden. Auch ein Vergleich der Anzahl von Agentengruppen und Agentenfarbwerten kann erst später erfolgen. Somit kann es passieren, dass ein Szenario scheinbar erfolgreich geladen wird, obwohl es mit dieser Konfiguration nicht oder nur fehlerhaft simuliert werden kann. Der Benutzer muss an dieser Stelle selbst die entsprechende Sorgfalt aufbringen. Ist mit der Konfigurationsdatei (vorerst) alles in Ordnung, erscheint ein Vorschaubild des Simulationsgebiets auf der rechten Seite der Aktivität. Die eingelesen Werte werden im Erfolgsfall in einem Paket (`Bundle`) gespeichert und zu den nachfolgenden Aktivitäten transportiert. In der „Sensitivitätsanalyse“-Aktivität werden die Informationen zum Teil weiterverarbeitet, in der „Simulations“-Aktivität schließlich, werden alle Informationen gebraucht, um z. B. die Zellen-Objekte zu erstellen.

4.3.2 Bilddatei des Simulationsgebiets

Die Bilddatei soll der Definition des Simulationsgebiets dienen. Jeder Pixel mit seinem Farbwert entspricht dabei einer Zelle. Damit die Bilddatei korrekt eingelesen werden kann, muss sie folgende Voraussetzungen erfüllen:

- Datenformat: PNG (RGBA 32-Bit, unkomprimiert)
- 1 Pixel ≡ 1 Zelle $(\approx 40cm^2)$
- Stockwerke müssen immer rechteckig sein, bedingt durch die Definitionen in der Konfigurationsdatei.
- Nicht benötigte Teile des Bildes sollten zur Beschleunigung der Simulation in Wandzellenfarben angemalt werden.
- Eine Wandzelle sollte nach Möglichkeit im Norden, Süden, Westen oder Osten an eine andere Wandzelle angrenzen (3.5.2).

Zellentyp	Spektrum der gültigen Farben		
	in RGB	in hexadezimal	in Worten
Normal	(0, 0, 0)	0*x*000000	Schwarz
Wand	(1, 1, 1) bis (255, 255, 255)	0*x*010101 bis 0*xFFFFFF*	Grauwerte von 1 bis 255
Treppe	(0, 1, 0) bis (0, 255, 0)	0*x*000100 bis 0*x*00*FF*00	Grünwerte von 1 bis 255
Teleport	(0, 255, 255)	0*x*00*FFFF*	Türkis (Cyan)
Ausgang	(0, 0, 255) bis (255, 0, 255)	0*x*0000*FF* bis 0*xFF*00*FF*	Blau mit Rotwerten von 0 bis 255
Agent	(255, 0, 0) bis (255, 255, 0)	0*xFF*0000 bis 0*xFFFF*00	Rot mit Grünwerten von 0 bis 255

Tabelle 4.1: Farbtabelle: Welcher Zelltyp bzw. welche Zellengruppe benötigt welchen Farbwert

Tabelle 4.1 enthält die Vorschriften für die Vergabe der Farbwerte in der Bilddatei. Alle Farben, die von den hier definierten abweichen, gelten als *ungültig*. Die entsprechenden Zellen werden zu Wandzellen der Farbe 0*xFFFFFF* (weiß). Nachdem die Konfigurationsdatei eingelesen wurde, öffnet die „Szenario-Laden"-Aktivität die Bilddatei. Diese liest der Reihe nach die einzelnen Farbwerte der Pixel ein und speichert sie in einem normalen Integer-Array, das ich nachfolgend als *Farb-Array* bezeichnen möchte. Die Reihenfolge des Einlesens ist dabei von links oben nach rechts unten zu interpretieren. Die tatsächliche Verarbeitung der Farbwerte, um daraus entsprechende Zellen zu generieren, wird erst beim Erstellen der „Simulations"-Aktivität vorgenommen. Bis dahin werden die Informationen über das Bild - die Abmessungen und das Farb-Array - in einem Paket (Android-Datenstruktur `Bundle`) weitergereicht. Das hat den Hintergrund, dass ich die Erstellung der Simulationsobjekte an einer zentralen Stelle haben wollte. Da viele Teile der Simulation erst zum späteren Zeitpunkt initialisiert werden können, habe ich die Generierung der Zellen ebenfalls nach hinten verschoben. Bei der späteren Verarbeitung, wird über das Farb-Array iteriert. Für einen einfacheren Zugriff im späteren Simulationsverlauf, werden die aus den Farbwerten erstellten Zellen in einem zweidimensionalen Array („Zellen-Array") gespeichert (Vgl. 4.3.3).
Je nach Farbe, werden unterschiedliche Zellen erstellt. Mit Ausnahme von Agenten und Ausgangszellen, geschieht dies direkt im aktuellen Iterationsschritt. Zusätzlich wird die Position und die Farbe der Zelle gesetzt. Das Objekt wird nach Fertigstellung an entsprechender Position im Zellen-Array gespeichert. Agenten- und Ausgangszellenobjekte können nicht direkt erstellt werden, weil sie meistens in Gruppen zusammengefasst sind. Eine Gruppe wird (wie schon in 4.3.1 angesprochen) durch die gemeinsame Farbe im Bild definiert. Da beim ersten Durchlauf über die Farbwerte die **Reihenfolge** der Gruppen noch ermittelt wird, können die Agenten und Zielzellen noch nicht ihren entsprechenden Gruppen zugeordnet werden. Deshalb wird zuerst folgendes gemacht: Die Farbwerte der Agenten- und Ausgangsgruppen werden jeweils getrennt in einer aufsteigend sortierten Menge gespeichert,

in der Mehrfacheinträge nicht vorkommen können. In Java erledigt diese Aufgabe eine `TreeSet<Integer>`-Datenstruktur. Die Anzahl der Elemente dieser Menge entspricht damit der Anzahl der unterschiedlichen Gruppen. Daneben wird der Farbwert zusammen mit der Position als Tripel in einer Liste gespeichert (`ArrayList<int[3]>`); jeweils getrennt für Agenten und Ausgangszellen. Es wird implizit angenommen, dass ein Agent initial auf einer leeren Zelle steht. Deshalb wird für diesen Typ zusätzlich eine solche leere Zelle an der Position des Agenten erstellt und dem Zellenarray hinzugefügt. Falls eine Farbe ungültig ist, wird an ihrer Position eine Wandzelle mit dem Standardfaktor erstellt. Der Faktor richtet sich nach dem Standardwert, der in der Konfigurationsdatei vergeben wurde. Ist dort keiner vorhanden oder ungültig, gilt $1,0$ als Standardwert.
Wenn der Durchlauf durch das Farb-Array vollendet ist, müssen die Agenten und Ausgangszellen erstellt und ihren jeweiligen Gruppen zugeordnet werden. Als erstes werden die Ausgangsobjekte (`Exit`-Klasse) erstellt. Jede Ausgangszelle speichert eine Referenz auf ein solches Ausgangsobjekt, das damit die Gruppe für mehrere Ausgangszellen darstellt. Ihre Identifikationsnummer (ID) entspricht dabei der Position des Farbwertes in der sortierten Menge. Danach wird über die Tripel-Liste der Ausgangszellen hochgezählt. Die Zelle wird generiert und ihre Position und Farbe an Hand der Werte des Tripels gesetzt. Mit Hilfe des Farbwertes wird die ID der entsprechenden Ausgangsgruppe ermittelt und die Referenz auf das Ausgangsobjekt gesetzt. Zum Abschluss wird die Ausgangszelle im Zellenarray gespeichert. Listing 4.2 zeigt hierfür den entsprechenden Quellcodeausschnitt. Agenten

Listing 4.2: Erstellung der Ausgangszellen

```
ArrayList<int[]> exit_pixelvalues = new ArrayList<int[]>();
// (...)
for (int[] it : exit_pixelvalues)
{
   // it[0] = y-position, it[1] = x-position, it[2] = color
   Cell_Exit cell = new Cell_Exit(new Point(it[0], it[1]));
   cell.setColor(it[2]);
   int exit_id = exit_map.get(it[2]);
   cells.get(it[1]).put(it[0], cell);
   cell.setExitReference(exits.get(exit_id));
}
```

werden nicht im Zellenarray gespeichert. Da sie permanent ihre Position verändern und gesondert auf sie zugegriffen wird, werden sie in einem assoziativen Array gespeichert (`SparseArray<ArrayList<Agent>>`), wobei der Schlüssel die ID der Agentengruppe und der Wert eine Liste der Agentenobjekte der entsprechenden Gruppe ist. Beim Iterieren über die Agenten-Tripel wird nun ein Agentenobjekt erstellt. Die Position ist im Tripel definiert. Der ebenfalls dort gespeicherte Farbwert, dient zum Ermitteln der Agentengruppe. Ihre ID gibt den Speicherort des Agenten im *SparseArray* vor. Außerdem werden die, in der Konfigurationsdatei definierten und in der Agentengruppe gespeicherten, Parameter gesetzt. Siehe dazu auch Listing 4.3. Als letzten Schritt müssen die Teleportzellen mit ihren Partnerzellen verknüpft werden. Auch wenn sie bereits alle erstellt wurden, die Referenzen auf ihre Partner können erst jetzt gesetzt werden. Hier helfen die Definitionen aus der Konfigurationsdatei. Sie werden der Reihe nach durchgegangen. Es wird auf die Teleportzelle im Zellen-Array an der „from“-Position

Listing 4.3: Erstellung der Agenten (Ausschnitt)

```
ArrayList<int[]> agent_pixelvalues = new ArrayList<int[]>();
// (...)
for (int[] it : agent_pixelvalues)
{
   int agentgroup_id = agent_map.get(it[2]);
   double[] params = agentgroups.get(agentgroup_id).params;

   // parameters for Agent-constructor:
   // position, ID of agent group, agent-parameters, reference to cell-array and
       underlying cell
   Agent cell = new Agent(new Point(it[0], it[1]), agentgroup_id, params, cells,
       cells.get(it[1]).get(it[0]));
   cell.setColor(it[2] | 0xFF000000);
   ArrayList<Agent> temp = agents.get(agentgroup_id);
   if (temp == null)
   {
      temp = new ArrayList<Agent>();
      agents.put(agentgroup_id, temp);
   }
   temp.add(cell);
}
```

zugegriffen und ihr die Referenz auf die Teleportzelle an der „to“-Position zugewiesen. Das gleiche gilt in umgekehrter Richtung.
An vielen Stellen kann es dazu kommen, dass die Informationen aus der Bilddatei nicht mit denen aus der Konfigurationsdatei übereinstimmen. Die Angaben der Teleportzellen können falsch sein, so dass an den definierten Positionen keine Teleportzellen erstellt wurden. Ein Verknüpfen der Teleportzellenpartner ist damit unmöglich. Es kann auch sein, dass mehr Agentengruppen im Bild vorhanden sind, als in der XML-Datei definiert wurden. Oder es wurden z. B. keinerlei Ausgangszellen im Bild definiert. Alle diese Gründe führen zu einem Abbruch der Initialisierung, da ein korrektes Ausführen des Simulation unmöglich oder fehlerhaft ist. Der Abbruch wird dem Benutzer über Fehlermeldungen mitgeteilt und er wird zurück zur „Simulationseinstellungen“-Aktivität geführt.

4.3.3 Distanzberechnungen

In Kapitel 3.5 ging ich bereits auf die Algorithmen zur Distanzberechnung ein. Der Fokus lag allerdings auf den Modifikationen der Originalalgorithmen. In diesem Abschnitt will ich detaillierter auf die Implementierung der Algorithmen eingehen.

Wichtige Datenstrukturen

Im Vorfeld traten bereits oft abstrakte Objekte wie das Zellen-Array, die Knotenzellenmenge oder die einzelnen Stockwerksebenen auf. Für die Implementierung des F.A.S.T.-Modells

müssen diese abstrakten Gebilde als konkrete Datenstrukturen definiert werden. Deshalb will ich in diesem Teil des Kapitels die wichtigsten Datenstrukturen präsentieren und erläutern, warum ich mich für sie entschieden habe. Da Java die Programmiersprache der Wahl war, führe ich auch direkt Datenstrukturen auf, die in dieser Sprache vorhanden sind.

- **SparseArray**<T>: Eine neue Datenstruktur von Android. Vergleichbar mit Java-HashMaps, allerdings sind die Keys primitive Integer-Werte (*int*s). Dadurch entfällt der Aufwand von Wrapper-Variablen, wodurch Zugriffe auf Elemente von SparseArrays schneller sind, als bei HashMaps. Daneben gibt es noch weitere neue Datenstrukturen, wie z. B. **SparseIntArray**, SparseBooleanArray und LongSparseArray. Eine ausführliche Beschreibung dieser, sowie aller anderen Objekte, findet man in [An3].

- class Cell: Die Oberklasse aller Zelltypen. Sie enthält allgemeine Eigenschaften und Funktionen aller Zellen. Davon abgeleitet sind alle folgenden Zelltypen: Cell_Exit, Cell_Wall, Cell_Teleport, Cell_Stair, Agent extends Cell. Die Agent-Objekte nehmen eine besondere Rolle dabei ein: Sie werden auf Grund ihrer Beweglichkeit gesondert gespeichert und besitzen eine Vielzahl spezifischer Funktionen, unter anderem zum Finden von Zielzellen oder eines neuen Ausgangs.

- **SparseArray**<**SparseArray**<Cell>> cells: Ein zweidimensionales assoziatives Array (row-first). Es speichert die Zellobjekte des Simulationsgebiets. Nach der Erstellung des Arrays während der Initialisierung der Simulation, gibt es keine Veränderungen mehr. Es finden nur noch lesende Zugriffe statt, diese allerdings auf beliebige Indizes. Der Zugriff wird durch die primitiven Integer-Keys und das Hashing der Elemente allerdings beschleunigt.

- **SparseArray**<**ArrayList**<Agent>> agentsById: Dieses assoziative Array verknüpft die Agentengruppen mittels ihrer Identifikationsnummern mit einer Liste der Agenten, die zu der Gruppe gehören. Jeder Agent gehört zu genau einer dieser Listen. Die direkte Filterung der Agenten nach Gruppe, erspart in vielen Situationen im Programm ein vollständiges Durchlaufen aller Agenten, wenn z. B. Parameter gruppenweise geändert werden müssen oder die Anzahl der Agenten einer Gruppe gefragt ist.

- **HashSet**<Cell> nodeCells: Die Menge der Knotenzellen. Hier werden Referenzen auf Zellen aus dem vorigen cells-Array gespeichert. Oftmals muss entschieden werden, ob ein Element in dieser Menge enthalten ist. Dies wird durch das Hashen der Elemente sehr schnell erledigt. Gleichzeitig garantiert das **HashSet**, dass nur jeweils ein Exemplar eines Objekt in der Menge enthalten ist. Eine vorherige Prüfung fällt damit weg, was zur Übersichtlichkeit des Programmcodes beiträgt. Da die Reihenfolge der Objekte keine Rolle spielt, kann hier auf den Einsatz von **SparseArray**s oder **HashMap**s verzichtet werden.

- **HashSet**<Cell> exitCells: Die Menge der Ausgangszellen. Hier sprechen dieselben Argumente für den Einsatz der **HashSet**-Datenstruktur, wie auch schon für nodeCells. Da an manchen Stellen im Programmcode gesondert auf die Ausgangszellen zugegriffen werden muss, habe ich sie gesondert gespeichert, obwohl die originalen Ausgangszellenobjekte in cells gesichert sind.

- `XML_Floor`: Eine Container-Klasse für Ebenen. Ein Objekt dieser Klasse speichert den Namen (`String`), die Position und die Abmessungen (`Point`-Objekte) der Ebene. Die Werte der Position und Abmessung sind als Pixelpositionen im Bild zu interpretieren. Die Informationen werden aus der Konfigurationsdatei herausgelesen und für spätere Zugriffe aus anderen Teiles des Programms in diesen Klassenobjekten gespeichert.
- **`ArrayList`**`<XML_Floor> floors`: Dieses Array sammelt alle Stockwerksinformationen. Der Array-Index entspricht dabei der Reihenfolge ihres Auslesens aus der Konfigurationsdatei und damit auch gleichzeitig der ID der Ebene. Vor allem die Distanzberechnungsalgorithmen benötigen diese Informationen, um entweder über die einzelnen Ebenen zu iterieren, oder um die Lage zweier Zellen zu vergleichen.

Finden von Knotenzellen

Der erste Teilalgorithmus für die Distanzberechnungen bestimmt die Knotenzellen im Simulationsgebiet. Siehe dazu auch 3.5.2. Als erstes wird über alle Zellen iteriert und für jede Zelle die Nachbarzellen untersucht. Das hat zur Folge, dass für die Randzellen eine Sonderbehandlung notwendig ist, denn hier liegen manche Nachbarzellen außerhalb des gültigen Array-Bereichs. Achtet man nicht darauf, wirft das Programm sofort `IndexOutOfBounds`-Exceptions. Um dem vorzubeugen wird der obere, untere, linke und recht Zellenrand, sowie die mittleren Zellen jeweils gesondert untersucht. Ein weiteres Problem ist die Übersetzung der Forderung, wann eine Zelle eine Knotenzelle ist, in Programmcode: Wenn eine Zelle n 1 gemeinsame Kante mit der Wandzelle c hat und weniger als 2 gemeinsame Ecken mit anderen Nachbarzellen von c hat, dann ist n eine Knotenzelle. In Java-Code übersetzt, ergibt dies mehrere boolsche Ausdrücke, in denen die entsprechenden Kombinationen auf Wandzellen-Zugehörigkeit mittels `instanceof` geprüft werden. Als letzten Schritt in diesem Algorithmus muss nochmals über alle Zellen iteriert werden und alle Zellen, die („`instanceof`“) Teleportzellen sind, in die Menge der Knotenzellen aufgenommen werden. Der dazugehörige Pseudocode steht in Algorithmus 4.1.

Verknüpfe Zellen mit sichtbaren Knotenzellen

Wie bereits in 3.5.3 beschrieben, dient dieser Algorithmus zwei Zwecken: Zum einen werden Referenzen auf alle sichtbaren Knotenzellen in allen Zellen gespeichert. Zum anderen werden auch die Distanzen zu den referenzierten Knotenzellen berechnet und gespeichert.
In einer äußeren Schleife wird über die einzelnen Stockwerke iteriert. Die `floors`-Datenstruktur liefert die Informationen über sie, so dass nur der Bereich des Stockwerks in `cells` angeschaut wird. Erstes Ziel des Algorithmus ist es, eine Menge mit Referenzen auf die Randzellen der Ebene zu füllen; eine Randzelle, bildet mindestens an einer Seite den Rand des Stockwerks. Das geschieht durch einfache For-Schleifen und Zugriffe auf das Zellen-Array. Sind die Randzellen bestimmt, wird über alle Knotenzellen iteriert. Da jede Zelle die Nummer des Stockwerks speichert, auf der sie liegt, kann ein einfacher Vergleich dafür sorgen, dass Knotenzellen ignoriert werden, die nicht im aktuellen Stockwerk liegen. Die verbleibenden Knotenzellen

Algorithmus 4.1 Bestimme Knotenzellen

```
function DetermineNodeCells(cells, nodeCells)
    h ← |cells|, w ← |cells[0]|
    for (y = 0 → h − 1) do
        for (x = 0 → w − 1) do
            c ← cells[y][x]
            if (c type of Cell_Wall) then
                for all (neighbor cell n of c) do
                    if (n shares 1 edge with c and < 2 corners with surrounding cells of c) then
                        nodeCells ← nodeCells ∪ n
                    end if
                end for
            end if
        end for
    end for
    nodeCells ← nodeCells ∪ {t ∈ cells : t type of Cell_Teleport}
end function
```

dienen als Startpunkt für eine weiter For-Schleife, die nun über die Randzellen iteriert. Auf diese werden die Sichtstrahlen von der Knotenzelle ausgehend geschossen. Dieser „Schuss“ ist im Programmcode als Funktionsaufruf eines angepassten Bresenham-Algorithmus zu sehen. Listing 4.4 zeigt genau diesen Programmausschnitt, der gesamte Algorithmus ist in Pseudocode 4.2 schematisch dargestellt.

Listing 4.4: Verknüpfe sichtbare Zellen

```
for (Iterator<Cell> n_it = nodeCells.iterator(); n_it.hasNext(); )
{
   Cell nodeCell = n_it.next();
   if (nodeCell.getFloorLvl() != fnmb) //fnmb = current floor number
      continue;
   for (Cell bCell : borderCells)
      bresenhamNode(nodeCell, bCell, cells, nodeCells);
}
```

Erweiterter Bresenham-Algorithmus

Der Pseudocode 4.4 zeigt den Standard-Bresenham-Algorithmus [Cun]. Diesen habe ich erweitert, so dass er Auskunft darüber gibt, ob sich zwei Zellen gegenseitig sehen. Zusätzlich wird die Distanz zwischen der Startzelle und den auf dem Weg befindlichen Zellen berechnet. Da die Startzelle eine Knotenzelle ist, wird die Distanz mit der Referenz auf die Knotenzelle als Paar in der anderen Zelle gespeichert. Diesen Zusatzcode habe ich an das Ende der inneren For-Schleife angefügt. Dort sind die Berechnungen der neuen Zellenposition abgeschlossen

Algorithmus 4.2 Verknüpfe Zellen mit sichtbaren Knotenzellen

```
function LinkCellDists(floors, cells, nodeCells)
    for all (f ∈ floors) do
        borderCells ← {c ∈ cells : c is in boundary belt of cells of f}
        for all (n ∈ nodeCells) do
            if (n not located in f) then
                continue with next n
            end if
            for all (b ∈ borderCells) do
                BresenhamNode(n, b, cells, nodeCells)
            end for
        end for
    end for
end function
```

Algorithmus 4.3 Bresenham-Algorithmus für Knotenzellen

```
function BresenhamNode(src, dest, cells, nodeCells)
    initialize variables for Bresenham's algorithm (see 4.4)
    while (src ≠ dest) do
        move pixel by pixel towards dest
        c ← cells[src.y][src.x]
        if (c type of Cell_Wall) then
            return
        else
            if (src not a node cell of c) then
                dist ← ||src − c||
                add (src, dist) to c
            end if
        end if
    end while
end function
```

und ein weiterer Zählschritt steht bevor. Sollte eine Wandzelle auf dem Weg der beiden Grenzzellen liegen, wird der Algorithmus ohne weitere Berechnungen an der Stelle des Fundes abgebrochen. Denn dann ist klar, dass es keinen Sichtkontakt zwischen den folgenden Zellen und der Startzelle geben kann.

Bestimmung der Distanzen von Knotenzellen zu Ausgängen

Wenn dieser Algorithmus eingesetzt wird, kennt jede Zelle ihre sichtbaren Knotenzellen und die Distanzen zu ihnen. Es fehlen aber noch die endgültigen Distanzberechnungen zu den Ausgangszellen. Für die Knotenzellen wird dies nun hier erledigt. Der Pseudocode 4.5 zeigt

Algorithmus 4.4 Algorithmus von Bresenham

function BRESENHAM(src, dst)
$d_x \leftarrow |dst_x - src_x|,\ d_y \leftarrow |dst_y - src_y|$
$$inc_x \leftarrow \begin{cases} -1 & d_x < 0 \\ 0 & d_x = 0 \\ 1 & d_x > 0 \end{cases}$$
$$inc_y \leftarrow \begin{cases} -1 & d_y < 0 \\ 0 & d_y = 0 \\ 1 & d_y > 0 \end{cases}$$
$d_x \leftarrow |d_x|,\ d_y \leftarrow |d_y|$
if $(d_x > d_y)$ **then**
$pd_x \leftarrow inc_x,\ pd_y \leftarrow 0$
$ef \leftarrow dy$
$es \leftarrow dx$
else
$pd_x \leftarrow 0,\ pd_y \leftarrow inc_y$
$ef \leftarrow dx$
$es \leftarrow dy$
end if
$dd_x \leftarrow inc_x,\ dd_y \leftarrow inc_y$
$err \leftarrow es/2$
for $(i = 0 \rightarrow es - 1)$ **do**
$err \leftarrow err - ef$
if $(err < 0)$ **then**
$err \leftarrow err + es$
$src_x \leftarrow src_x + dd_x,\ src_y \leftarrow src_y + dd_y$
else
$src_x \leftarrow src_x + pd_x,\ src_y \leftarrow src_y + pd_y$
end if

here the current cell on the line can be accessed

end for
end function

diesen Algorithmus in einer schematischen Darstellung.
Der Algorithmus iteriert über die Ausgangszellen. Somit erhalten alle Knotenzellen ihre kürzesten Entfernungen zu jeweils einer Ausgangszelle. Der innere Teil der Schleife implementiert Dijkstras Algorithmus [Dij59]. Der Startknoten jeder Suche ist die aktuelle Ausgangszelle $e \in exitCells$. Die Menge Q speichert die Knotenzellen, deren finale Distanz noch unbekannt ist. Bei mir wird Q von einer `HashSet<Cell>` repräsentiert. Es müssen an beliebigen Stellen Elemente hinzugefügt werden, an bestimmten Stellen Elemente gelöscht werden und sehr oft entschieden werden, ob ein Element in Q enthalten ist. Daher bietet sich diese Datenstruktur hervorragend an. D ist ein assoziatives Array, das Knotenzellen (Key) mit ihrer aktuell minimalen Distanz (Value) zu e speichert. Eine effiziente und einfach handhabbare Datenstruktur stellt Java mit der `HashMap<Cell, Double>` zur Verfügung. Der Vorteil beim Hinzufügen neuer Elemente ist folgender: Ist der Schlüssel noch nicht vorhanden, wird das neue Paar einfach hinzugefügt. Ist er dagegen schon vorhanden, wird sein Wert durch den neuen Wert überschrieben. Es muss im Vorfeld nichts abgefragt oder gelöscht werden, was Operationen und Zeit spart.
Den Ablauf des Algorithmus von Dijkstra will ich hier nicht beschreiben, aber auf ein paar Details eingehen. Die Vorbelegung der Distanzwerte der Nicht-Startzellen ist mit Java einfach möglich: `Double.POSITIVE_INFINITY` ist ein exklusiver Status der Double-Objekte, das dem mathematischen „∞“ entspricht. Die Zelle in D, deren Distanz zu Beginn der While-Schleife am geringsten ist, gilt als vollständig bearbeitet. Ihre Distanz ist minimal und somit wird das Distanz-Paar $(e, dist_{min})$ in ihr gespeichert: Jede Zelle hält eine `HashMap<Cell, Double>`, ähnlich zu D, bereit. Dort wird eine Ausgangszellenreferenz mit einer Entfernung verknüpft. Am Ende dieses Teilalgorithmus enthalten die Maps aller Knotenzellen die gewünschten minimalen Distanzen zu den Ausgangszellen. Die Menge N der Nicht-Wand-Nachbarzellen von u entspricht in meiner Version einer schlichten `Set<Cell>`, da lediglich eine Iteration über ihre Elemente notwendig ist. Wie der Algorithmus von Dijkstra es vorsieht, ist die neue minimale Distanz zwischen $n \in N$ und e : $dist_{min} = \min\left(D[u] + \|u - n\|, D[n]\right)$.

Bestimmung der Distanzen normaler Zellen zu Ausgangszellen

Es fehlen vor der Ausführung dieses Algorithmus für die endgültigen Distanzberechnungen noch die Distanzen aller Nicht-Knotenzellen zu allen Ausgangszellen. Dies wird von diesem letzten Teilalgorithmus erledigt. Wie bereits im vorherigen Abschnitt 4.3.3, wird in einer äußeren Schleife über die Ausgangszellen iteriert ($e \in exitCells$). Siehe dazu auch Pseudocode 4.6. Innerhalb dieser Schleife wird über alle Elemente des Zellen-Arrays iteriert ($cell \in cells$). Sollte *cell* eine Wandzelle oder Knotenzelle sein, wird direkt zur nächsten Zelle gesprungen, denn für erstere werden keine Distanzwerte benötigt und für letztere sind sie bereits im vorigen Schritt berechnet worden. Mittels eines Vergleichs durch `instanceof Cell_Wall` bzw. der Abfrage `nodeCells.contains(cell)` ist dies sehr schnell möglich. Im anderen Fall wird über alle Knotenzellen iteriert, die für *cell* sichtbar sind. Sie sind in einer `HashMap<Cell, Double>` im Zellenobjekt gespeichert und wurden im Algorithmus zur Verknüpfung der Zellen hinzugefügt (Algorithmus 4.2). Da es in Java keinen Iterator für `HashMap`s gibt, musste ich mit `MapEntry` arbeiten. Listing 4.5 zeigt den entsprechenden Ausschnitt. Die For-Schleife iteriert über die Paare der Map. Innerhalb dieser Schleife, wird die minimale Distanz zwischen *cell* und *e* stets

Algorithmus 4.5 Distanzen zwischen Knoten- und Ausgangszellen

```
function DetermineExitDists(nodeCells, exitCells)
    for all (e ∈ exitCells) do
        for all (v ∈ nodeCells) do
            D ← D ∪ (v, ∞)
        end for
        D ← D ∪ (e, 0)
        Q ← nodeCells
        while (Q ≠ ∅) do
            (u, dist_min) ← pair with minimum distance in D
            Q ← Q \ u
            add (e, dist_min) to u
            N ← {n : n is direct neighbor of u}
            for all (n ∈ N) do
                if (n ∈ Q) then
                    alt ← D[u] + ||u − n||
                    if (alt < D[n]) then
                        D ← D ∪ (n, alt)
                    end if
                end if
            end for
        end while
    end for
end function
```

aktualisiert, indem das Minimum der bereits existierenden Distanz und der Summe der beiden Distanzen ||*cell*− Knotenzelle || und || Knotenzelle −*e*|| genommen wird. Das resultierende Paar aus Ausgangszelle *e* und Distanz $dist_{min}$ wird in der Ausgangszellen-Map von *cell* gespeichert. Auch diese Map ist eine `HashMap<Cell, Double>`.
Ist die äußerste For-Schleife beendet, kennen alle Nicht-Wandzellen ihre kürzesten Ausgangsdistanzen und haben diese intern gespeichert. Betrachtet man die gespeicherten Daten isoliert, so erhält man das statische Floor-Field.

Listing 4.5: Kürzeste Entfernung für eine Zelle berechnen

```
for (Map.Entry<Cell, Double> nodeEntry : cell.getNodeDists().entrySet())
{
   double new_dist = nodeEntry.getValue() + nodeEntry.getKey().getExitDist(exit);
   if (new_dist < min_dist)
      min_dist = new_dist;
}
```

Algorithmus 4.6 Distanzen normaler Zellen zu Ausgangszellen

```
function DetermineAllExitDists(cells, nodeCells, exitCells)
    for all (e ∈ exitCells) do
        for all (c ∈ cells) do
            if (c type of Cell_Wall ∨ c ∈ nodeCells) then
                continue with next c
            else
                dist_min ← min_{n∈c.nodes} (||c − n|| + ||n − e||)
                add (e, dist_min) to c
            end if
        end for
    end for
end function
```

Bestimmung der Distanz zur nächstgelegenen Wandzelle

Es gibt einen weiteren Algorithmus zur Distanzberechnung, den ich für meine Implementierung brauche. Er wird nicht für die Berechnung der Entfernungen zu den Ausgangszellen benötigt, sondern zur Ermittlung der nächstgelegenen Wandzellen und deren Distanz.
Das Grundprinzip dieses Algorithmus ist, um die betrachtete Zelle Kreise zu ziehen. Sollte eine Wandzelle in einem solchen Kreis vorhanden sein, wird kein weiterer gezogen, sondern nur noch die restlichen Zellen des aktuellen Kreises untersucht. Für jede gefundene Wandzelle wird die Distanz zwischen ihr und der Zentrumszelle berechnet. Ist sie kleiner als das bisherige Minimum, wird die Distanz und Referenz auf die Wandzelle zwischengespeichert. Die Abtastrate des Kreises, d. h. die Anzahl der Punkte, die einen Kreis bilden, werden in Abhängigkeit des Radius mit folgender Formel bestimmt: $points = \lfloor r \cdot 2^{(3+log_{10}(r))} \rfloor$. Damit wird sichergestellt, dass es keine Lücken in den Kreisen gibt, die z. B. durch Rundungsfehler entstehen könnten. Da allerdings W_{max} (maximaler Sicherheitsabstand der Agenten (2.3.2)) meist recht klein gehalten wird, sind die Größenordnungen, ab der die Formel ungenau wird, vernachlässigbar. Es wird über die Punkte des eben beschriebenen Zellenkreises iteriert und mit Hilfe des Winkels $angle = \frac{2\pi}{points}$ und den Winkelfunktionen die Koordinaten der Zellen berechnet:

$$coord_x = x + \lfloor \cos(angle \cdot i) \cdot r + 0,5 \rfloor coord_y = y + \lfloor \sin(angle \cdot i) \cdot r + 0,5 \rfloor,$$

wobei i die Laufvariable über die Punkte ist. Wenn eine Wandzelle im aktuellen Kreis enthalten ist, können nur noch Zellen dieses Kreises die minimale Distanz unterbieten. Das liegt daran, dass es auf Grund der ganzzahligen Zellenpositionen zu Rundungen kommt. Die Zellen des nächst größeren Kreises haben aber eine um mindestens 1 größere Distanz zur Zentrumszelle. Diese ist, abzüglich Rundung, immer größer als die größte Distanz einer Zelle des kleineren Kreises. Deshalb wird der Algorithmus bei Fund einer Wandzelle nach Überprüfung aller Kreiszellen abgebrochen. Eine kleine Optimierung des Algorithmus besteht darin, dass er im Falle $dist_{min} \leq r$ sofort unterbrochen wird. Wenn eine Wandzelle eine Distanz kleiner oder gleich dem Radius zur Zentrumszelle hat, können selbst die Zellen desselben Kreises dies

Algorithmus 4.7 Distanz zur nächstgelegenen Wandzelle

```
function DetermineWallDist(src, floor, cells)
    next ← true
    r ← 1
    dist_min ← ∞
    while (next = true) do
        points ← ⌊r · 2^(3+log10(r))⌋
        angle ← 2·π / points
        for (i = 0 → points − 1) do
            coord_x ← src.x + cos(angle · i) · r
            coord_y ← src.y + sin(angle · i) · r
            if (coordinates inside simulation area ∧ cells[coord_y][coord_x]type of Cell_Wall) then
                next ← false
                dist_new ← ||src − cells[coord_y][coord_x]||
                if (dist_new < dist_min) then
                    dist_min ← dist_new
                    nearestWall ← cells[coord_y][coord_x]
                    exit loop if (dist_min ≤ r)
                end if
            end if
        end for
        r ← r + 1
    end while
    src.setWallDist(dist_min)
    src.setWallReference(nearestWall)
end function
```

nicht weiter unterbieten. Deshalb wird in diesem Fall, sofort die For-Schleife beendet. In jedem Fall steht am Ende sowohl die kleinste Distanz, als auch die betreffende Wandzelle fest. Sowohl der Wert, als auch die Referenz auf die Wandzelle werden in der Zentrumszelle gespeichert.

4.3.4 Anwenden der Regeln

In der „Sensitivitätsanalyse“-Aktivität werden die Regeln definiert, die später vor jedem Durchlauf auf die entsprechenden globalen Variablen, Agenten- oder Zellparameter angewandt werden. Eine solche Regel wird in einem Container-Objekt gespeichert. Alle Container werden wiederum separat für jeden Typus (Allgemein, Agenten, Treppen, Wände) in eigenen Datenstrukturen gespeichert. Das ist notwendig, weil die Position in der Datenstruktur gleichzeitig als Angabe der Parameter-ID oder z. B. der Agentengruppe dient. Ein gemeinsames Speichern aller Regeln in einer einzigen Datenstruktur, würde zusätzliche Angaben im Regel-Container und eine anschließend größere Filterung erfordern. Listing 4.6 zeigt die `RuleContainer`-Klasse, deren Objekte die einzelnen Regeln speichern. Jeder `RuleContainer`

Listing 4.6: Klasse für Regeln

```
public class RuleContainer
{
   public final DISTRIB_FUNCTION_ID functionID;
   public final double a;
   public final double b;
   public final double c;

   private RuleContainer(DISTRIB_FUNCTION_ID funcID, double a, double b, double c)
   {
      this.functionID = funcID;
      this.a = a;
      this.b = b;
      this.c = c;
   }
}
```

beinhaltet die Angabe über die zu verwendende Wahrscheinlichkeitsverteilung, sowie die Werte für den Minimalwert (bzw. den Erwartungswert), den Maximalwert (bzw. die Varianz), sowie den wahrscheinlichsten Wert, falls die Dreiecksverteilung ausgewählt wurde. In allen anderen Fällen bleibt dieser Wert 0 und wird ignoriert.
Innerhalb der kleineren Initialisierungsphase vor einem Simulationsdurchlauf wird eine Funktion aufgerufen, die neue Parameter entsprechend der Regeln erzeugt und zuweist (siehe Listing 4.7). Da die Regeln nach Klassen getrennt gespeichert wurden, wird nacheinander durch die verschiedenen Regellisten durchgegangen. Mit den Parametern, die in einem Regel-Container gespeichert sind, wird ein neuer Wert bestimmt. Dazu gibt es eine weitere Funktion, die je nach Funktions-ID eine andere Wahrscheinlichkeitsverteilung mit den zwei bzw. drei Funktionsparametern verwendet. Der neu bestimmte Wert wird anschließend dem Simulationsparameter, allen Agenten der Agentengruppe oder allen Wand- bzw. Treppenzellen der entsprechenden Farbgruppe zugewiesen. Zusätzlich wird der neue Wert zusammen mit der Angabe der Regel in der Statistik-Ausgabedatei (siehe 4.5.2) gespeichert.

4.4 Simulationsdurchführung

Nachdem zuvor die Konfigurationsdaten und andere Einstellungen (unter anderem auch die Regeldefinitionen für die Sensitivitätsanalyse) in den vorherigen Menüs eingelesen bzw. erstellt wurden, ist die Simulation initialisiert. Nun soll es in diesem Abschnitt um die Durchführung der Simulation gehen.
Ausgangspunkt ist die „Simulation“-Aktivität. Der Benutzer kann dort die Simulation starten, pausieren, abbrechen, zu den bisherigen bzw. endgültigen Statistiken schalten und - wenn eine grafische Ausgabe existiert - zwischen den einzelnen Stockwerken wechseln. Die Simulation besteht aus einer vorher festgelegten Anzahl einzelner Durchläufe. In der „Simulationseinstellungen“-Aktivität kann diese Zahl vom Benutzer eingestellt werden. Ein Durchlauf startet, nachdem die Simulation zum ersten Mal initialisiert wurde bzw. nach

Listing 4.7: Ausschnitt aus Regelanwendungsfunktion

```
// (...)
if (commonRules != null && commonRules.size() > 0)
{
   RuleContainer rCon = commonRules.get(0);
   if (rCon != null)
      this.maxWallDist = SimHelpers.giveNewValue(rCon.functionID, rCon.a, rCon.b,
          rCon.c);
   // (...)
}
if (agentRules != null && agentRules.size() > 0)
{
   for (int i = 0; i < agentsById.size(); i++)
   {
      SparseArray<RuleContainer> rules = agentRules.get(i);
      // (...)
   }
}
if (stairRules != null && stairRules.size() > 0)
{
   // (...)
}
if (wallRules != null && wallRules.size() > 0)
{
   // (...)
}
// (...)
```

erfolgtem Durchlauf reinitialisiert wurde und endet, wenn **alle Agenten** das Simulationsgebiet verlassen haben. Ist ein Durchlauf beendet, gibt es zwei Möglichkeiten: War es der letzte Durchgang, ist die gesamte Simulation zu Ende. Es kann kein weiterer Durchlauf gestartet werden, so dass sich der Benutzer zwischen dem Anzeigen der Resultate, dem Starten einer neuen Simulation und dem Abbruch (Rückkehr zum Hauptmenü) entscheiden kann. War der beendete Durchlauf nicht der letzte, startet ein weiterer. Hierfür muss die Simulation zum Teil neu initialisiert werden. Allerdings ist dafür sehr viel weniger Zeit notwendig, als beim ersten Mal. Es werden lediglich folgende Schritte unternommen: Die Agenten werden neu erstellt, so dass sie identisch sind, mit denen des letzten Durchlaufs. In diesem Zusammenhang werden auch die Bilder der Stockwerke neu gezeichnet, so dass die Agenten auf ihnen wieder zu sehen sind. Das dynamische Floor-Field wird zurückgesetzt, genauso wie die Agentenzähler der Ausgänge. Der Rundenzähler wird ebenfalls auf 0 zurückgesetzt. Sollte die Simulation im Experten-Modus ausgeführt werden, wird an dieser Stelle die Zuweisung der Parameter, die durch die Regeln der Sensitivitätsanalyse neue Werte bekommen, an die entsprechenden Objekte durchgeführt. Wenn eine grafische Ausgabe der Simulation in den Einstellungen gewünscht wurde, muss der Benutzer selbstständig den neuen Durchlauf starten. Andernfalls startet der nächste automatisch, sobald alle gerade beschriebenen Aufgaben erledigt sind. Dieses Verhalten hat folgenden Hintergrund: Ist die grafische Ausgabe nicht gewünscht, versucht das Programm die Simulation so schnell wie möglich durchzuführen, damit die Ergebnisse rasch vorhanden sind. Ein Pausieren zwischen den Durchläufen und die manuelle

Fortsetzung ist hier hinderlich. Bei der grafischen Ausgabe achtet das Programm darauf, nicht mehr als 2 Runden pro Sekunde zu simulieren, so dass die Bewegungen der Agenten auf dem Feld beobachtbar bleiben. Die langsame Geschwindigkeit, sowie das Pausieren der Simulation zwischen zwei Durchläufen kann hier nützlich sein, um dem interessierten Publikum etwas zu erklären, die Zwischenergebnisse zu betrachten oder sich Notizen machen zu können.
Die tatsächliche Durchführung der Simulation, sowie die Anzeige der Stockwerke und das Aktualisieren des Benutzerinterfaces übernehmen zwei von mir erstellte Klassen. Diese beiden werden im folgenden Abschnitt detaillierter beschreiben.

4.4.1 SimView- und SimViewThread-Klasse

Die Klasse `SimView` habe ich abgeleitet von einer neuen Android-Klasse `SurfaceView`. Diese Standardkomponente sorgt dafür, dass ein fester Zeichenbereich in der Aktivität entsteht. In ihr geschieht die grafische Ausgabe. Das tatsächliche Zeichnen der Ausgabe übernimmt das View eigenständig. Meine abgeleitete Klasse erbt diese Funktionalität, dient aber auch mehreren Aufgaben vor, während und nach der Simulation: Einerseits implementiert die `SimView`-Klasse einen sogenannten `SurfaceHolder`. Über ihn bekommen auch Nicht-UI-Threads Zugriff auf die Zeichenfläche. Das ist wichtig, denn Android verbietet Threads, die nicht für die Anzeige und Interaktion mit dem Benutzer zuständig sind (*UI-Threads*), die Änderungen von Menükomponenten. Der später noch näher beschriebene `SimViewThread` kann damit die grafische Ausgabe realisieren, obwohl er nicht zu den *UI-Threads* gehört. Zusätzlich kümmert sich SimView um die Erstellung und Zerstörung des eben angesprochenen SimViewThreads und verarbeitet die Nachrichten, die es von ihm erhält. Dies bewerkstelligt eine sogenannte `Handler`-Klasse, der man die Nachrichten zuschicken kann. Als Teil der Aktivität kann SimView somit Einfluss auf die anderen Views der Aktivität nehmen und z. B. Meldungen bezüglich des Simulationsstatus anzeigen lassen. Dazu ändert es die Texte der entsprechenden `TextViews` ab. Außerdem kümmert es sich um die erforderlichen Maßnahmen, wenn sich der Bildschirm ändert, die App pausiert, gestoppt oder neu geladen wird. Dazu zählen unter anderem: Herunterfahren des SimViewThreads, Neuinitialisierung der Simulation, Neuzeichnen der Grafiken.
Die `SimViewThread`-Klasse habe ich von der Standard `Thread`-Klasse abgeleitet. Ihre Aufgabe ist es, die Simulation voranzutreiben und eine Ausgabe (falls gewünscht) auf den Bildschirm zu zeichnen. Es existiert nur ein Objekt dieser Klasse, das wiederum im einzigen SimView-Objekt instanziiert ist. Die Existenz des Threads ist unabhängig von der Simulation: Wird das SimView-Objekt erstellt, generiert dieses automatisch auch ein SimViewThread-Objekt. Wird die „Simulation“-Aktivität verlassen, wird der SimViewThread gestoppt und zerstört, die Ergebnisse der Simulation und vor allem auch der aktuelle Fortschritt bleiben aber erhalten. Wird die Aktivität wieder betreten, wird ein neues SimViewThread-Objekt erstellt und die Simulation kann dort weitergeführt werden, wo sie pausiert wurde. Das liegt daran, dass das Simulationsobjekt global existiert und somit erst gelöscht wird, wenn explizit eine neue Simulation erstellt wird oder die Anwendung komplett beendet wird. Dieses Vorgehen belastet zwar den Hauptspeicher des Geräts mehr, weil die Daten auch bei Inaktivität der Anwendung vorgehalten werden, aber das Programm ist dafür sehr viel schneller wieder benutzbar, wenn man es wieder aktiviert.

Je nach Status der Simulation, sorgt der Thread für unterschiedliche Arbeiten: Zu Beginn initialisiert er die Simulation (siehe 4.3), innerhalb der einzelnen Durchläufe führt er die Simulationsaktualisierungen durch, nach Beendigung eines Durchlaufs bereitet er den nächsten vor und führt die Regeln der Sensitivitätsanalyse durch. Ist der letzte Durchlauf beendet, wird der Thread ebenfalls durch das SimView-Objekt gestoppt. Auch der SimViewThread kann verschiedene Zustände haben: Direkt nach seiner Erstellung befindet er sich im `[READY]`-Zustand. In ihm versucht er die Simulation endgültig zu initialisieren, so fern dies nicht bereits früher geschehen ist; es kann der Fall eintreten, dass die Anwendung noch vor dem ersten Update der Simulation inaktiv und wieder reaktiviert wurde. Direkt aus diesem Zustand heraus, setzt er sich selbst auf `[PAUSE]`. Auch durch den Benutzer, der im Benutzerinterface auf „Pause“ drückt, kann der Thread in diesen Zustand gelangen. In ihm wird die Simulation nicht weiter vorangetrieben. Der Thread kümmert sich aber weiterhin um die Anzeige der Grafiken, so dass ein Wechsel der Stockwerke auch im Pause-Modus möglich bleibt. Ist der Thread im `[RUNNING]`-Zustand, wird die Simulation so lange vorangetrieben, bis der aktuelle Durchlauf zu Ende ist, oder bis durch den benutzerbedingten Abbruch oder die Pausierung der Thread zerstört oder wieder pausiert wird. Listing 4.8 zeigt einen Ausschnitt der `SimViewThread.run`-Funktion, die so lange ausgeführt wird, bis der Thread gestoppt wird.

4.4.2 Simulationsschritt

Ist der SimViewThread im Status `[RUNNING]`, führt er in einer Endlosschleife, die Simulationsrunden durch. Eine Simulationsrunde besteht in meiner Implementierung, wie auch im F.A.S.T.-Modell aus drei Teilen: Zuerst entscheidet sich jeder Agenten für einen Ausgang, danach wählt jeder Agent aus der Menge seiner gültigen Zielzellen eine Zelle aus, die er am Ende der Runde besetzen möchte und zum Schluss werden die entsprechenden Bewegungen der Agenten durchgeführt. All dies geschieht in der `Simulation.update`-Funktion, die vom SimViewThread aufgerufen wird. Zusätzlich werden in ihr die Graphen und Bilder aktualisiert, die später Teile der Ergebnisse der Simulation beinhalten sollen. Dazu gehört auch, dass zum Ende der Runde überprüft wird, ob die Simulation bzw. der einzelne Durchlauf beendet ist und eventuell ein neuer gestartet werden soll. Die folgenden Abschnitte widmen sich den einzelnen Teilen dieser Update-Funktion.

Ausgangssuche der Agenten

Im ersten Teil der Runde werden die Fernziele der Agenten aktualisiert. Je nachdem, wie der Prozess der Ausgangssuche eines Agenten endet, bleibt dieser bei seiner letzten Wahl, oder entscheidet sich für einen anderen Ausgang. Die Suche gliedert sich dabei in zwei Teile: Das Simulationsobjekt aktualisiert in seiner Update-Funktion einen Agenten nach dem anderem. Es ruft für jeden die agenteneigene Funktion auf, die dann die Ausgangssuche durchführt. Listing 4.9 zeigt diese Funktion in einer verkürzten Version. Folgendes ist bei der Ausgangswahl zu beachten: Ausgänge können aus mehr als einer Ausgangszelle bestehen. Es reicht daher nicht nur einen Ausgang zu wählen, sondern es muss auch eine Ausgangs*zelle* gewählt werden. Ansonsten kann nicht eindeutig eine Distanz zum *Ausgang* berechnet werden, was Folgen für

Listing 4.8: *run*-Funktion des SimViewThreads (Ausschnitt)

```
public void run()
{  // (...)
   while (running)
   {
      if (mode == STATE_READY)
      {
         if (simulation.isTotallyFinished())
         {
            // send message "Simulation finished"
            // (...)
            running = false; // stop thread by leaving endless-loop
         }
         else
         {
            simulation.Init();
            simulation.prepareNewRun();
            if (graphicalmode)
               updateBackgroundImage();
            mode = STATE_PAUSED;
         }
         // (...)
      }
      else
      {  // (...)
         if (mode == STATE_RUNNING)
            updatePhysics();
         doDraw(c);
      }
      if(graphicalmode)
      {
         // if update cycle was shorter than 500 ms then sleep for the difference
             time
         Thread.sleep(sleepTime);
      }
   }
}
```

den Teileinfluss des statischen Floor-Fields hätte (siehe 2.3.2). Die Wahrscheinlichkeitsformel (2.1) für die Ausgangssuche habe ich somit für ihre Implementierung leicht abgeändert:

$$p_e^A = N \cdot \frac{(1 + \delta_{AE}\kappa_E)}{S(A,e)^2}. \tag{4.1}$$

Dabei steht e nun für eine Ausgangszelle anstatt eines Ausgangs. Für δ_{AE} gilt: $\delta_{AE} = 1$, wenn e zum Ausgang E gehört und dieser in der letzten Runde gewählt wurde, auch wenn der Agent zuvor eine andere Zelle dieses Ausgangs als Fernziel hatte.
Es gibt zwei assoziative Arrays: `HashMap<Exit, Double> pe` und `HashMap<Exit, Cell_Exit> ec`. Im ersten werden die Ausgänge mit einem Wahrscheinlichkeitswert verknüpft, im zweiten mit einer Ausgangszelle. Verknüpft wird immer diejenige Zelle des Ausgangs, deren Wahrscheinlichkeit am größten ist. Nachdem alle Einzelwahrscheinlichkeiten berechnet wurden, enthält

Listing 4.9: Agentenfunktion zum Wählen eines Ausgangs

```
public void chooseNewExit()
{
   HashMap<Exit, Double>    pe = new HashMap<Exit, Double>();
   HashMap<Exit, Cell_Exit> ec = new HashMap<Exit, Cell_Exit>();
   for (Map.Entry<Cell, Double> e : curCell.exitDists.entrySet())
   {
      Cell_Exit eCell = (Cell_Exit)e.getKey();
      Exit exitRef    = eCell.getExitReference();
      int delta_exit  = 0;
      if (lastExit == exitRef)
         delta_exit = 1;
      double probability = (1.0 + (delta_exit * params[Constants.K_E])) /
          Math.scalb(e.getValue(), 2);
      if (!pe.containsKey(exitRef) || pe.get(exitRef) < probability)
      {
         pe.put(exitRef, probability);
         ec.put(exitRef, eCell);
      }
   }
   // (...) normalization of the values
   Exit choice  = chooseProbability(pe);
   lastExit     = nextExit;
   nextExit     = choice;
   nextExitCell = ec.get(choice);
}
```

ec Verknüpfungen zwischen allen Ausgängen und ihren jeweiligen Ausgangszellen, die unter allen anderen Zellen des Ausgangs am wahrscheinlichsten sind. Aus diesen wird in einem letzten Schritt der tatsächliche Ausgang und damit die Ausgangszelle gewählt.

Zielzellensuche der Agenten

Die Wahl einer Zielzelle ist, genauso wie die Wahl eines Ausgangs, ein Prozess, der unabhängig von anderen Agenten durchgeführt werden kann. Die Wahl der Zielzelle findet in meinem Programm deshalb im gleichen Iterationsschritt über die Agenten statt, wie die Ausgangssuche.
Der Status der Zelle, die im Moment vom Agenten besetzt wird, gilt während der Zielzellensuche als „unbesetzt“. Dadurch wird diese Zelle ebenfalls in die Menge der gültigen Zielzellen aufgenommen. Bevor die Suche nach den Zielzellen gestartet wird, muss die Geschwindigkeit des Agenten aktualisiert werden. Zuerst wird sie um 1 erhöht, außer es wurde bereits die maximale Geschwindigkeit (v_{max}) erreicht. Danach wird geprüft, ob sich der Agent auf einer Treppenzelle befindet. In diesem Fall wird die neue Geschwindigkeit mit dem Faktor der Treppenzelle multipliziert. Dabei wird sichergestellt, dass sie nicht kleiner als 1 wird. Ansonsten würde der Agent permanent auf der Treppenzelle verharren und somit nie eine Ausgangszelle erreichen. Das wiederum würde dazu führen, dass die Simulation nicht enden kann, weil nicht alle Agenten das Simulationsgebiet verlassen haben (vgl. 3.1).

Im nächsten Schritt findet nun die Suche nach den erreichbaren Zellen statt. Den Aufbau des Algorithmus stellt 4.8 dar. In der Ergebnismenge C sind alle Zellen enthalten, die theoretisch in dieser Runde vom Agenten von seiner jetzigen Position und mit seiner derzeitigen Geschwindigkeit aus erreichbar sind. Neben der Menge C, die bei mir eine einfache `ArrayList<Cell>` ist, gibt es noch die Menge M und U. In M werden alle Zellen gespeichert, die bereits untersucht wurden. Da hier sehr oft Mengenzugehörigkeiten geprüft werden müssen, habe ich mich für eine `HashSet<Cell>` entschieden. Mit dieser Menge will ich verhindern, dass Zellen mehrfach besucht werden und somit eventuelle Endlosschleifen entstehen. Die dritte Menge U beinhaltet alle Zellen, die noch untersucht werden müssen. In meiner Implementierung ist U eine Warteschlange (`Queue<Cell>`). Neue Elemente werden am Ende der Liste angehängt. Das älteste Element steht somit ganz vorne in der Liste und wird als erstes bearbeitet.
Ausgehend von der Startzelle, auf der sich der Agent befindet, breitet sich die Suche nach Zielzellen im Gebiet aus. Alle Nachbarzellen der Startzelle werden in U aufgenommen. Von jeder dieser Zellen werden wieder alle Nachbarzellen in U aufgenommen und so weiter. Das Kriterium, ob eine Nachbarzelle aufgenommen wird, ist die ihre Distanz zur Startzelle. Ist sie größer als der Geschwindigkeitsbetrag des Agenten, scheidet sie als erreichbare Zielzelle aus. Da das Simulationsgebiet so aufgebaut sein kann, dass potentielle Zielzellen keine direkte Verbindung zur Startzelle haben (d. h. „nicht sichtbar“ für die Startzelle sind), müssen ihre Distanzen durch Teildistanzen aufsummiert werden. Hierfür gibt es die `lastDest`-Variable, die jede Zelle besitzt. In ihr wird die Referenz auf die letzte sichtbare Zelle auf dem Weg zur Startzelle hinterlegt. Gibt es keine direkte Verbindung zwischen einer Zelle und der Startzelle, berechnet sich die Distanz wie folgt: $||Zelle - Startzelle|| = ||Zelle - lastDest|| + ||lastDest - Startzelle||$. Der zweite Summand ist bekannt, denn die `lastDest`-Zelle wurde vom Suchalgorithmus bereits zuvor besucht (weiter vorne in U gelegen). Abbildung 4.2 zeigt nochmals ein Beispiel für den Einsatz dieser Hilfsvariable. Die Ergebnismenge ist in keinem Fall leer, da sie zumindest die Startzelle enthält. Sollte die Ergebnismenge nach Beendigung der Suche lediglich diese eine Zelle enthalten, tritt ein Sonderfall ein. Das bedeutet, dass sich der Agent zu keiner anderen Zelle bewegen kann. Eine Auswahl der Zielzelle ist damit bereits getroffen. Die Geschwindigkeit des Agenten wird auf 0 gesetzt, die Zelle wird als „besetzt“ markiert und es wird mit dem nächsten Agenten fortgefahren. Enthält die Ergebnismenge mehr als eine Zelle, muss nun eine Auswahl getroffen werden. Dazu wird der entsprechenden Agentenfunktion die Menge der potenziellen Zielzellen übergeben. Diese arbeitet ähnlich, wie die Funktion zur Ausgangssuche: Eine For-Schleife iteriert über die Zellen der Menge. Für jede Zelle wird ein Wahrscheinlichkeitswert nach Formel (2.2) berechnet. Eine zusätzliche Funktion wählt aus den normalisierten Wahrscheinlichkeiten eine aus, die für den Agenten dann als neue Zielzelle gilt. Zum Abschluss wird die vom Agenten besetzte Zelle, die während der Zielzellensuche als „unbesetzt“ galt, wieder als „besetzt“ markiert.

Bewegen der Agenten

Zu aller erst wird eine Liste (`ArrayList<Agent> pleaseRemove`) erstellt. In ihr werden die Referenzen auf diejenigen Agenten gespeichert, die durch ihre folgende Bewegung eine Ausgangszelle erreichen und damit erfolgreich aus der Simulation ausscheiden. Die Liste dient am Ende der Runde dafür, die entsprechenden Agenten aus ihrer Datenstruktur zu entfernen.

Algorithmus 4.8 Finde gültige Zielzellen

```
function DetermineDestCells(a, cells)
    C, M, U ← ∅
    v ← current speed of a
    if (a stands on Cell_Stair) then
        v ← max(1, ⌈v · stairCell.factor⌉)
    end if
    start ← cell, a stands on
    start.lastDest ← start
    U ← U ∪ start,  M ← M ∪ start
    while (U ≠ ∅) do
        u ← first element from U
        C ← C ∪ u (if not blocked)
        N ← neighbor cells of u
        if (u type of Cell_Teleport) then
            add u.partnerCell to N
        end if
        for all (n ∈ N ∧ n ∉ M) do
            if (u.lastDest is visible for n) then
                dist ← ||n − u.lastDest|| + ||u.lastDest − a||
                n.lastDest ← u.lastDest
            else
                dist ← ||n − u|| + ||u − a||
                n.lastDest ← u
            end if
            if (dist ≤ v) then
                U ← U ∪ n
                ||n − a|| = dist
            end if
            M ← M ∪ n
        end for
    end while
    return C
end function
```

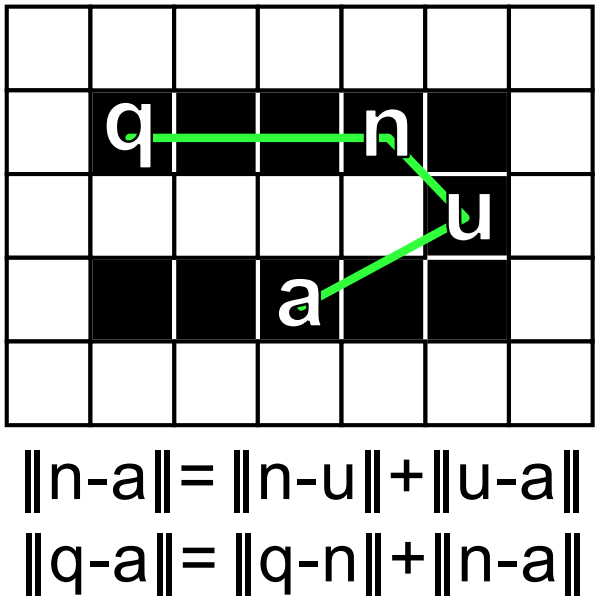


Abbildung 4.2: Distanzberechnung bei der Zielzellensuche. In diesem Beispiel wird die Distanz der Zellen n und q zu a gesucht. Die Berechnung mit Hilfe von Teildistanzen ist unter dem Bild dargestellt. Entscheidend ist die Hilfe der *lastDest*-Variablen (siehe 4.4.2)

In meiner Implementierung habe ich den Vorschlag des *direkten Hinspringens* zur Zielzelle umgesetzt (siehe 2.3.3). Es gibt aber eine Änderung, was die auftretenden Konflikte angeht: Bei mir bewegen sich die Agenten zu ihrer Zielzelle, sobald sie an der Reihe sind und dies möglich ist. Ist sie durch jemand anderen bereits besetzt, bleibt der Agent auf seiner aktuellen Zelle stehen. Ein Konflikt wird aufgelöst, indem immer der Agent gewinnt, der als erster die gemeinsame Zielzelle besetzt. Da die Reihenfolge der Agenten für jede Runde zufällig gesetzt wird, realisiert mein Programm in gewisser Weise das direkte Hinspringen mit $\mu = 0$; denn ein Konflikt wird hier immer aufgelöst. Bevor die Agentenbewegungen stattfinden können, muss noch die zufällige Reihenfolge für die Agenten festgelegt werden. Dazu füge ich einem Array (`ArrayList<Agent> agentOrder`) alle vorhandenen Agenten hinzu und wende auf diese Menge eine Misch-Funktion an, die für die zufällige Anordnung der Elemente im Array sorgt. Java stellt eine solche Funktion durch die `Collections`-Klasse zur Verfügung: `Collections.shuffle(agentOrder)`. Sie erreicht die zufällige Durchmischung der Elemente in linearer Zeit, d. h. dieser Schritt ist somit effizient durchführbar.
Danach wird über die Agenten in der eben festgelegten Reihenfolge iteriert. Unabhängig von einer grafische Ausgabe, wird das Bild der Ebene, auf der sich der Agent befindet, erneuert. Der Bildpunkt, der bisher die Farbe des Agenten hatte, soll die ursprüngliche Farbe der belegten Zelle erhalten. Beim Erstellen der Zellenobjekte (während der Initialisierung), wurde sie als Eigenschaft für jede Zelle gespeichert. An dieser Stelle wird darauf nun zurückgegriffen. Die eigentliche Bewegung des Agenten erfolgt durch einen Aufruf einer agenteninternen Funktion. Dort wird zuerst überprüft, ob die Zielzelle noch unbelegt ist. Ist sie das nicht, kann

sich der Agent nicht bewegen und seine Geschwindigkeit wird auf 0 gesetzt. Andernfalls werden folgende Schritte ausgeführt: Die bisher besetzte Zelle wird freigegeben (`NOT_BLOCKED`), die Position des Agenten wird angepasst, der Bewegungsvektor wird auf das Feld des dynamischen Floor-Fields addiert, die Referenz auf die besetzte Zelle wird geändert und es wird geprüft, ob der Agent nun eine Ausgangszelle besetzt. Ist der letzte Vergleich negativ, muss die neue Zelle nun als besetzt gelten. Außerdem wird die Geschwindigkeit des Agenten entsprechend seiner eben ausgeführten Bewegung gesetzt; v_{max} gilt dabei wieder als Obergrenze. Schließlich erhält der Bildpunkt an der Stelle der neu besetzten Zelle, die Farbe des Agenten. Wenn der Agent erfolgreich eine Ausgangszelle erreicht hat und damit „geflüchtet“ ist, wird zum einen der Zähler des Ausgangs inkrementiert und zum anderen der Agent zu der am Anfang beschriebenen Liste (`pleaseRemove`) hinzugefügt.
Sind alle Agentenbewegungen durchgeführt, ist bekannt, welche Agenten in dieser Runde entkommen sind. Ihre Referenzen stehen in der „Entfernen“-Liste. Mit Hilfe der von Java zur Verfügung gestellten Mengenoperationen, können die betreffenden Agenten aus der Gesamtagentenmenge leicht entfernt werden (`.removeAll(pleaseRemove)`). In Listing 4.10 ist eine verkürzte Version des Abschnitts der Agentenbewegungen zu sehen.

Listing 4.10: Reihenfolge aufstellen und Bewegungen durchführen

```
ArrayList<Agent> pleaseRemove = new ArrayList<Agent>();
ArrayList<Agent> agentOrder = new ArrayList<Agent>();
for (int i = 0; i < agentsById.size(); i++)
   agentOrder.addAll(agentsById.get(i));
Collections.shuffle(agentOrder);
for (Agent a : agentOrder)
{
   // reset pixel color
   a.moveToDest(); // internal agent movement function
   if (a.hasEscaped())
      pleaseRemove.add(a);
   else
      // set new pixel color
}
for (int i = 0; i < agentsById.size(); i++)
{
   ArrayList<Agent> temp = agentsById.get(i);
   temp.removeAll(pleaseRemove);
   remain += temp.size();
}
```

Ende der Runde

Das Ende der Runde besteht aus einer Vielzahl von Aktualisierungsvorgängen. So wird als erstes das dynamische Floor-Field bearbeitet. Je nach Größe der Simulationsparameter α (für Verwischung) und δ (für Verblassen) kommt es zu einer Diffusion unter den Komponenten bzw. einer Abschwächung der absoluten Beträge der einzelnen Vektorfelder (siehe dazu auch Kapitel 2.2.2). Zusätzlich werden alle Graphen aktualisiert: Der Evakuierungsgraph, der den

Verlauf der entkommenen Agenten dokumentiert, der Graph für den durchschnittlichen Fluss der Agenten und auch die Ausgangsstatistiken (die nicht als Graph, sondern textuell ausgegeben werden). In Abschnitt 4.5 gibt es weitere Details zum Aufbau und zur Speicherung von Graphen in meinem Framework. Die Bilder (z. B. die aktuelle Dichte der Agenten) werden hier nicht aktualisiert. Dies geschieht erst, wenn die „Ergebnis"-Aktivität besucht wird. Dies würde sonst unnötig Zeit vergeuden.
Daran anschließend folgt eine Überprüfung der Anzahl der verbliebenen Agenten in der Simulation. Hier gibt es verschiedene Kriterien, ab wann zusätzliche Schritte eingeleitet werden: Hat die Zahl die 95%-Marke erreicht bzw. überschritten, müssen dafür die entsprechenden Statistiken für die Evakuierungszeiten aufgenommen werden. Liegt die Anzahl bei weniger als 4, werden die Namen der Agentengruppen, zu denen die verbliebenen Agenten gehören, gesichert, so dass sie am Ende der Simulation in den Ergebnissen dokumentiert werden können. Sollten alle restlichen Agenten in dieser Runde das Simulationsgebiet verlassen haben, geht der aktuelle Durchlauf mit dieser Runde zu Ende. In diesem Fall wird eine Markierung in der Simulation gesetzt (`finished = true`), auf die anschließend vom SimViewThread reagiert wird. Sollte mit dieser Runde sogar der letzte Durchlauf beendet werden, wird zusätzlich die Variable `overallFinished` auf `true` gesetzt. Denn das Ende der gesamten Simulation zieht weitere Maßnahmen für den Thread und die umgebende Aktivität nach sich (siehe 4.4.1).

4.5 Ergebnisspeicherung und -anzeige

Während der Simulation werden fortlaufend Daten über die Fortschritte gesammelt. Zur späteren Analyse sind diese Daten entscheidend und deshalb habe ich auch bei meinem Programm auf eine Speicherung und Ausgabe dieser Statistiken geachtet. Ich habe einige der in [Kre06] vorgeschlagenen und in Kapitel 2.4 beschriebenen Ausgaben implementiert. Sie können allesamt in der „Ergebnis"-Aktivität und in einer separater Textdatei betrachtet werden. Die Speicherung von fortlaufenden Daten als Graph habe ich mittels eigener generischer Klassen realisiert, während eine von Androids „ImageView" abgeleitete Klasse (`GraphView`) sich um die korrekte Anzeige dieser Graphen auf dem Bildschirm kümmert. Im Folgenden will ich genauer auf diese beiden Klassen eingehen.

4.5.1 Graph- und GraphView-Klasse

Die Anzahl der geflüchteten Agenten, die pro Runde in einem Graphen gespeichert wird, ist eine Ganzzahl (Integer), während der durchschnittliche Fluss der Agenten eine Fließkommazahl ist (Double). Ich wollte deshalb eine Klasse erstellen, die es möglich macht, jegliche Arten von numerischen Datentypen aufzunehmen und in Form eines Graphen zu speichern. Herausgekommen ist die `Graph`-Klasse. Sie ist generisch, d. h. eine Instanziierung dieser Klasse erfordert die Angabe eines Datentyps. Da sich nicht jedes Objekt dafür eignet, beschränkt sich die Wahl des Datentyps auf Objekte, die von Javas `Number`-Klasse abgeleitet sind und das Interface `Comparable` implementieren. Dazu zählen unter anderem die Standarddatentypen **`Integer`**, **`Float`**, **`Double`**, **`Long`**. Damit ist es möglich, die Klasse unverändert zu lassen, auch

wenn zu einem späteren Zeitpunkt ein neuer Datentyp für eine Speicherung als Graph wichtig wird.
Die Klasse speichert einen Namen zur Identifizierung des Graphen und die beiden Achsenbeschriftungen als `String`-Typ. Daneben gibt es mehrere assoziative Arrays, welche die gespeicherten Werte verwalten:

- **`SparseArray`**`<T> data_min`: Dieses Array speichert die minimalen Werte an den entsprechenden Indizes. Der Index ist die Position des Wertes auf der Abszisse. Aus diesem Grund habe ich darauf verzichtet eine normale `List<T>` oder ein simples Array (`T[]`) zu verwenden, da sonst der Index immer bei 0 beginnt, die Abszissenwerte aber durchaus negativ sein können. Dies hätte zu weiteren Variablen geführt und Umrechnungen, die bei meiner Variante wegfallen.
- **`SparseArray`**`<T> data_max`: Analog zu `data_min` speichert dieses Array die maximalen Werte der entsprechenden Abszissenpositionen.
- **`SparseIntArray`** `data_numbers`: Hierin wird gespeichert, wie viele Werte für einen bestimmten Wert auf der Abszisse vorhanden sind. Es ist ein Hilfsarray, um später die Durchschnittswerte besser und schneller berechnen zu können.
- **`SparseArray`**`<`**`Double`**`> data_sum`: Ein weiteres Hilfsarray, das die Werte an den entsprechenden Positionen der Abszisse des Graphs aufsummiert. Hier habe ich den Datentyp **`Double`** fest vergeben. Da die Durchschnittswerte ebenfalls **`Double`**-Werte sind, spare ich spätere Typkonvertierungen ein.
- **`SparseArray`**`<`**`Double`**`> data_avg`: Dieses Array speichert schließlich die Durchschnittswerte des Graphen, die sich mittels `data_sum` und `data_numbers` berechnen lassen.

Außerdem wird der insgesamt kleinste und größte Wert gesondert gespeichert, so dass die Abfrage dieser beiden Werte schnell erfolgen kann. Neben verschiedenen Funktionen zur Rückgabe einzelner oder aller Werte des Graph-Objekts, gibt es auch eine Funktion, mit der man einen neuen Wert hinzufügen kann. Listing 4.11 zeigt eine verkürzte Version von ihr. Bei der Speicherung des Wertes wird folgendermaßen vorgegangen: Es wird verglichen, ob der Wert kleiner als der aktuell kleinste Wert an dieser Position ist. Ist dies der Fall, wird das Minimum angepasst. Dasselbe geschieht für den aktuell größten Wert. Zusätzlich wird der Zähler inkrementiert, wie oft an der entsprechenden Position ein Wert aufgenommen wurde. Zum Schluss wird der Wert auf die Summe aller Werte für diese Position addiert.
Eine effiziente Lösung habe ich für die Speicherung der Durchschnittswerte entwickelt: Da sich bei jedem Hinzufügen eines neuen Wertes das arithmetische Mittel ändert, müsste man es auch jedes Mal neu berechnen. Um diese Zeit zu sparen, wird beim Hinzufügen eines neuen Wertes lediglich eine boolsche Variable gesetzt, die anzeigt, ob die aktuellen Durchschnittswerte aktuell sind oder nicht. Wird nun das arithmetische Mittel von außerhalb des Graph-Objekts angefordert, überprüft die Funktion diese Variable und gibt entweder sofort die Durchschnittszahlen zurück (Variable ist „falsch“), oder berechnet alle Durchschnittswerte, setzt die Variable auf „falsch“ und gibt danach die Werte zurück (Variable war „wahr“).

Listing 4.11: Neuen Wert einem Graph hinzufügen

```
public void addData(int where, T data)
{
   data_numbers.put(where, data_numbers.get(where) + 1);
   T cur = data_min.get(where);
   if (cur == null || cur.compareTo(data) > 0)
   {
      data_min.put(where, data);
      if (minimum == null || minimum.compareTo(data) > 0)
         minimum = data;
   }
   // analog for data_max (...)
   Double value = data_sum.get(where);
   if (value == null)
      value = data.doubleValue();
   else
      value += data.doubleValue();
   data_sum.put(where, value);
   noUpdate = false;
}
```

Die `GraphView`-Klasse ist von `ImageView` abgeleitet. Diese Komponente dient normalerweise dem Laden und Anzeigen eines Bildes, sowie dessen Einfärbung und Skalierung. Meine Klasse implementiert davon die Funktion neu, die für das Zeichnen des Bildes zuständig ist: `onDraw`. Daneben gibt es eine Reihe weiterer Funktionen, die das Hinzufügen, Löschen und Anzeigen der Graphen und Bilder regeln.
In der GraphView-Klasse wird je eine Liste dieser Graphen (**`ArrayList`**`<Graph<T>>`) und Bilder (**`ArrayList`**`<Bitmap>`) gespeichert. Das Simulationsobjekt fügt sie der Instanz hinzu, sobald sie initialisiert wird. Die Namen der gespeicherten Graphen und Bilder erscheinen in einer Auswahlliste des Benutzerinterfaces („Ergebnis“-Aktivität). Die dort getroffene Auswahl wird an das GraphView-Objekt weitergegeben, das dann die Ausgabe einleitet. Eine `Bitmap` dient als Leinwand, auf die der Graph oder das Bild gezeichnet wird. Damit ist es möglich auch zusätzliche Dinge auszugeben, wie z. B. die Achsenbeschriftungen oder eine Markierung am aktuell angewählten Abszissenwert, wenn der Benutzer auf den Bildschirm klickt. Das Grafikobjekt wird so skaliert, dass es den Bildschirm optimal ausfüllt. Angezeigt wird bei einem Graph jeweils der Verlauf der Minimal- und Maximalwerte (in roter bzw. grüner Farbe), sowie der Durchschnittswerte (in gelber Farbe). Die Bilder, z. B. der aktuelle Status des dynamischen Floor-Fields, geben ihre Darstellung selbst vor und müssen deshalb lediglich skaliert werden. Das Seitenverhältnis der Bilder wird dabei beachtet.

4.5.2 Speicherung in Textdatei

Nicht alle Statistiken können in Form von Graphen und Bildern angezeigt werden. Entweder, weil es einfach nicht geht (z. B. Namen von Agentengruppen), oder die Übersichtlichkeit darunter leiden würde. Dies gilt besonders im Hinblick auf die Parameterauswahl bei der

Sensitivitätsanalyse: Es müsste für jeden Durchlauf einen Evakuierungsgraphen geben, der mit der entsprechenden Regel bezeichnet wäre. Bei vielen Durchläufen würde dabei die Übersichtlichkeit so sehr leiden, dass eine Analyse der Ergebnisse nicht mehr gut durchführbar wäre. Auch aus diesem Grund wird, parallel zur Erstellung der Graphen und Bilder, eine Textdatei mit Ergebnissen gefüllt. Vor jedem Durchlauf werden die Parameter mit ihren neuen Werten aufgelistet, die auf Grund von Regeln der Sensitivitätsanalyse angepasst wurden. Ebenfalls für jeden Durchlauf werden die Rundenanzahlen dokumentiert, zu denen 95% bzw. 100% der Agenten geflüchtet sind. Zum Abschluss der Simulation werden Ergebnisse dokumentiert, die über alle Durchläufe hinweg ermittelt wurden: Es wird gespeichert, wie viele Runden es mindestens und höchstens gedauert hat, bis die beiden Agenten-Grenzen erreicht wurden. Zusätzlich wird der Durchschnittswert sowie die Standardabweichung für beide Fälle dokumentiert. Den Abschluss der Datei bildet eine Information über die jeweils zuletzt entkommenen Agenten: Es werden die Namen der Agentengruppen aufgelistet, aus denen mindestens ein Agent zu den letzten drei verbliebenen eines Durchlaufs gezählt hat. Zusätzlich gibt eine Zahl dahinter an, wie oft diese Gruppe bei allen Durchläufen dieses Kriterium erfüllt hat. Zur späteren Identifizierung wird die Datei mit dem aktuellen Datum und Uhrzeit zu Simulationsbeginn benannt. Dies soll auch verhindern, dass eine Datei eine ältere mit demselben Namen überschreibt.

5 Ergebnisse und Analyse

In diesem Kapitel will ich die Ergebnisse einiger Testläufe des Programms vorstellen. Sie sollen zeigen, welchen Einfluss die Größe des Simulationsgebiets und die Anzahl der Agenten auf die Simulationszeiten und den Speicherverbrauch haben. Während der Entwicklung habe ich das Programm nur mit relativ kleinen Szenarien getestet, hauptsächlich um bestimmte Situationen nachzustellen. Der Test mit großen Szenarien hat allerdings zwei große Probleme offenbart, deren Lösung bis zur Veröffentlichung dieser Arbeit nicht komplett behoben werden konnten:

1. Standardmäßig erlaubt Android auf dem Entwicklungsgerät „Samsung Galaxy Note 10.1“ jeder Benutzeranwendung maximal 64 MB vom Hauptspeicher zu nutzen. Das führt bereits bei mäßig großen Szenarien dazu, dass diese Grenze erreicht wird und das Programm mit Speicherfehlern beendet werden muss.
2. Die Größe des Programm-Heaps wird so klein wie möglich gehalten. Jede Speicherallokationen löst einen Aufruf des nebenläufigen „Garbage-Collectors“ aus, der versucht Platz zu schaffen. Bis er dies gemacht hat, bzw. bis die Heap-Größe erweitert wurde, vergehen unnötige Sekundenbruchteile, da der Haupt-Thread derweil warten muss. Je mehr Speicher bereits allokiert ist, desto länger werden die Wartezeiten. Dieses Problem fällt vor allem bei der Initialisierung auf, in der die großen Datenstrukturen mit Objekt-Instanzen gefüllt werden.

Eine erste Lösung für Problem 1 konnte ich schnell finden: Ein zusätzlicher Eintrag in der Manifest-Datei der Applikation (siehe Listing 5.1) sorgt für eine Anhebung der Grenze von 64 MB auf 256 MB. Wie die Ergebnisse zeigen, ist dies allerdings nicht ausreichend, um wirklich große Gebiete im Speicher zu halten. Das zweite Problem konnte ich bis zum Zeitpunkt dieser Veröffentlichung nicht lösen oder eindämmen. Vor allem jenes Problem spiegelt sich in den Ergebnissen wider.

Listing 5.1: Erweiterung des maximal nutzbaren Hauptspeicheranteils

```
<manifest ...>
<!-- (...) -->
<application
        android:largeHeap="true"
      <!-- (...) -->
<!-- (...) -->
```

5.1 Ergebnisse

Neben der Messung des Speicherverbrauchs wurden verschiedene Zeiten erfasst: Die Initialisierungszeit, die durchschnittliche Zeit pro Durchlauf, sowie die durchschnittliche Zeit pro Runde, bis zu dem Zeitpunkt, zu dem der erste Agent geflüchtet ist. Ich habe verschieden große Gebiete simuliert: Angefangen bei einem sehr kleinen Gebiet von 40×20 Zellen (16×8 Meter^2), bis hin zu einem Gebiet mit 160×80 Zellen (64×32 Meter^2). Jedes Areal ist vertikal in der Mitte in zwei gleich große Stockwerke aufgeteilt, die über eine Treppe am unteren Rand mittels Teleportzellen verbunden sind. Für jedes Gebiet gab es ein Szenario mit 30, 60 und 90 Agenten, welche alle relativ homogen über das Gebiet verteilt waren. Bei der Erhöhung der Agentenanzahl wurden die zusätzlichen Agenten jeweils neben den bereits vorhandenen platziert. Jedes Szenario wurde 5 Mal simuliert, mit 50 Durchläufen pro Simulation, so dass ein guter Mittelwert für die Durchschnittszeiten entstand. Die Agentenkonstanten und auch die maximale Geschwindigkeit der Agenten waren für alle Tests dieselben: $v_{max} = 4, \kappa_E = 5.0, \kappa_s = 1.5, \kappa_d = 0.5, \kappa_i = \kappa_w = \kappa_p = 0$. Alle Ergebnisse sind in Tabelle 5.1 zu finden, Abbildung 5.1 zeigt die Simulationsgebiete.
Ein weiterer Test soll verdeutlichen, wo die Grenze des simulierbaren auf dem Entwicklungsgerät liegt. Ein erster Versuch mit einem Szenario der Größe $200 \times 200 Zellen$ führte zu einer Überschreitung der 256 MB Speichergrenze. Die Ergebnisse des Tests mit einem Szenario der Größe 177×185 Zellen ($\approx 71 \times 74$ Meter^2) sind in Tabelle 5.2 dargestellt. Auf Grund der sehr langen Durchlaufzeiten, beschränkte ich die Anzahl der Durchläufe auf 5 pro Szenario.
Ein letzter Test wurde mit einer Sensitivitätsanalyse durchgeführt. Dazu habe ich das mittlere Szenario mit 30 Agenten gewählt. Die Regel zur Analyse lautete: „Verteile die Maximalgeschwindigkeit aller Agenten gleich zwischen 1 und 7“. Die Simulation bestand aus 50 Durchläufen, vor deren Start jeweils ein neuer, gleichverteilter Wert für v_{max} gewählt und allen Agenten zugewiesen wurde. Der resultierende Evakuierungsgraph ist in Abbildung 5.2 dargestellt.

5.2 Analyse

Einfluss der Gebietsgröße und Agentenanzahl

In den Ergebnissen des ersten Tests (5.1) ist deutlich zu erkennen, dass die Anzahl der Agenten nur sehr geringen Einfluss auf die Initialisierungszeit und den Speicherplatzverbrauch hat. Das liegt vor allem daran, dass zur Erstellung eines Agentenobjekts keine großen Operationen, wie z. B. Distanzberechnungen oder Zellverknüpfungen, nötig sind. Da die Agenten kleinere Datenmengen in sich speichern müssen, tragen sie auch wenig zum Speicherverbrauch bei bzw. erhöhen diesen bei steigender Anzahl nur geringfügig.
Die Erhöhung der Agentenanzahl spiegelt sich allerdings in einem Anstieg der Durchlaufzeiten wider. Hier kann man eine lineare Zunahme von ihnen erkennen. Man muss allerdings bedenken, dass hier bereits das Problem des Garbage-Collectors mit den Wartezeiten zum Tragen kommt. Besonders deutlich ist dies beim Anstieg der Initialisierungszeiten zu sehen: Die meiste Zeit während der Initialisierung wird für die Verknüpfung der Zellen und

	kleines Szenario			mittleres Szenario			großes Szeanrio		
Gebietsgröße	20x40 Zellen			40x80 Zellen			80x160 Zellen		
Anzahl Agenten	30	60	90	30	60	90	30	60	90
Zeit zur Initialisierung	0,40 s	0,42 s	0,42 s	1,43 s	1,41 s	1,47 s	10,74 s	10,72 s	10,74 s
∅ Zeit pro Durchlauf	1,40 s	3,21 s	6,52 s	3,12 s	6,43 s	10,13 s	10,70 s	20,03 s	30,81 s
∅ Zeit pro Runde, bis 1. Agent entkommen ist	66,5 ms	119,5 ms	159,9 ms	91,2 ms	167,0 ms	223,9 ms	165,8 ms	297,3 ms	393,6 ms
Speicherplatzverbrauch	9,58 MB	9,61 MB	9,68 MB	10,44 MB	10,51 MB	10,63 MB	24,47 MB	24,67 MB	24,69 MB

Tabelle 5.1: Ergebnisse der Simulations-Tests. Einfluss von Gebietsgröße und Agentenanzahl auf Zeiten und Speicherverbrauch.

	sehr großes Szeanrio	
Gebietsgröße	177x185 Zellen	
Anzahl Agenten	1315	2747
Zeit zur Initialisierung	184,3 s	189,8 s
∅ Zeit pro Durchlauf	1350 s	2490 s
∅ Zeit pro Runde, bis 1. Agent entkommen ist	13,44 s	16,49 s
Speicherplatzverbrauch	111,6 MB	114,6 MB

Tabelle 5.2: Ergebnisse der Simulations-Tests. Grenze für Simulationen auf aktuellem Entwicklungsgerät.

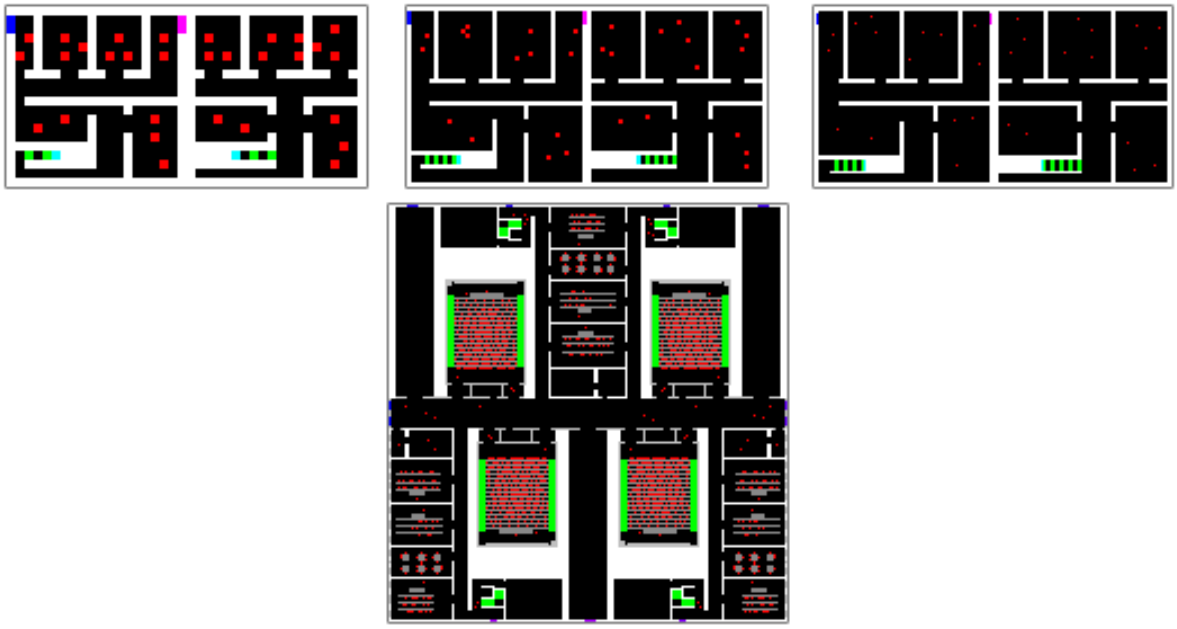


Abbildung 5.1: Verwendete Szenarien (von links nach recht und oben nach unten): kleines, mittleres, großes und sehr großes Szenario

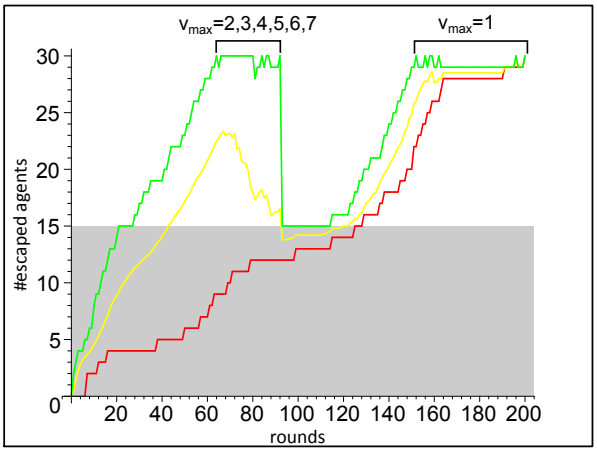


Abbildung 5.2: Evakuierungsgraph mit Sensitivitätsanalyse. Grauer Bereich: Die ersten 15 entkommenen Agenten sind diejenigen, die im 1. Stock starteten.

Knotenzellen, sowie für die Distanzberechnung zwischen den Ausgangszellen und den Knotenzellen benötigt. Alle Gebiete besitzen die gleiche Anordnung der Wandzellen. Die Anzahl der Knotenzellen unterscheidet sich für jedes Szenario somit nur geringfügig. Die Anzahl der Nicht-Wandzellen verfünffacht sich vom kleinen zum mittleren Gebiet und vervierfacht sich vom mittleren zum großen Gebiet. Während sich die Initialisierungszeit vom kleinen zum mittleren Gebiet ebenfalls fast verfünffacht (etwas weniger), liegt sie beim großen Gebiet beim 7,5-fachen der mittleren Zeit. Dieser ungewöhnliche Anstieg der Zeit ist durch das oben beschriebene Problem zu erklären. Der Test mit dem sehr großen Gebiet verdeutlicht diese Annahme nochmals: Auch wenn das Gebiet anders aufgebaut ist und damit die Anzahl der Knotenzellen stärker variiert, nimmt die Zeit zur Initialisierung um der 17-fache zu, während die Anzahl der Nicht-Wandzellen nur verdoppelt wird.
Um zu gewährleisten, dass die Berechnung der durchschnittlichen Rundenzeiten für alle Szenarien fair ist, habe ich den Durchschnitt nur von Rundenzeiten genommen, bis der erste Agent entkommen ist. Die Zunahme der Zeiten bei einer steigenden Anzahl Agenten ist nachvollziehbar: Mit der Anzahl der Agenten steigen auch die Zielzellensuchen pro Runde. Die Zielzellensuche hat den größten Einfluss auf die Rundenzeit. Von 30 auf 60 Agenten liegt die Zeitzunahme bei allen Gebieten zwischen dem 1,79- und 1,83-fachen, während sie von 60 auf 90 Agenten zwischen dem 1,32- und 1,34-fachen liegt. Hier hat die Größe des Gebiets keinen Einfluss. Allerdings nehmen die Zeiten bei gleicher Agentenzahl und größerem Gebiet zu: Die Rundenzeiten aller Agentenzahlen des mittleren Gebiets liegen zwischen dem 1,37- und 1,4-fachen über denen des kleinen Gebiets. Beim großen Gebiet liegt die Zunahme der Zeit im Vergleich zum mittleren Gebiet zwischen dem 1,76- und 1,82-fachen. Obwohl hier ebenfalls eine Linearität zu erkennen ist, erschließt sich der Grund für die Zunahme nicht direkt. Ein Vergleich der Anzahl an Zielzellen, zwischen denen sich alle Agenten während den ersten Runden entscheiden müssen, gibt hier einen Hinweis: Die Zahl liegt beim mittleren Gebiet circa 1,5-mal über der des kleinen Gebiets und circa 1,7-mal unter der des großen Gebiets. Diese Verhältnisse ähneln denen der Rundenzeiten. Die unterschiedliche Anzahl potenzieller Zielzellen bei gleicher Anzahl Agenten rührt vom Aufbau der Gebiete her: Beim jeweils kleineren Gebiet liegen die Wände dichter an den Agenten, so dass sich die Zielzellen auf einer kleineren Fläche verteilen. Da die Geschwindigkeit in jedem Test gleich groß war, änderte sich die Ausdehnung der Zielzellensuche in das Gebiet nicht. Ein weiteres Argument für die Zeitzunahme kann das Problem des Speicherallokierens sein, das sich bei größerem Gesamtspeicherverbrauch auch stärker auswirkt. Der Speicherplatzverbrauch ist beim großen Gebiet deutlich größer als bei den anderen beiden Gebieten.

Test der Systemgrenzen

Die Ergebnisse des Tests mit dem sehr großen Szenario zeigen die momentanen Grenzen. Die Initialisierung der Simulation liegt bei mehr als 3 Minuten, die Zeit für einen Durchlauf zwischen 20 und 40 Minuten, je nachdem, wie viele Agenten simuliert werden. Die gemittelte Zeit pro Runde, bis der erste Agent das Gebiet verlassen hat, liegt bei 13,4 bzw. 16,5 Sekunden. Hier wirkt sich das Problem mit den Wartezeiten für die Speicherbereinigung gleich doppelt aus: Zum einen in der Initialisierungsphase und zum anderen in jeder Runde für die Zielzellensuche der tausenden Agenten. Die Speichergrenze ließe noch mehr Agenten

zu, allerdings würde damit die Zeit eines einzigen Durchlaufs schnell über 60 Minuten klettern.

Test der Sensitivitätsanalyse

Die Abbildung 5.2 zeigt das Ergebnis der durchgeführten Sensitivitätsanalyse in Form des Evakuierungsgraphen. Auf den ersten Blick fällt sofort der große Abstand zwischen den Maximal- und Minimalwerten und der Einbruch der Maximalwerte bei Runde 90 (circa) auf. In jedem Stockwerk starteten 15 Agenten. Alle Ausgangszellen lagen im 1. Stockwerk. Aus dem Graph kann man schließen, dass trotz der Variierung der maximalen Agentengeschwindigkeit ($v_{max} \in [1, 7]$), es alle Agenten des unteren Stockwerks als erstes schafften zu entkommen, bevor ihre Kollegen aus dem zweiten Stockwerk eine Ausgangszelle erreichten. Die Auswertung der Textdatei hilft bei der weiteren Bewertung des Evakuierungsgraphen. Für jeden der 50 Durchläufe wurde der gewählte Parameterwert für v_{max}, sowie die Anzahl der Runden des gesamten Durchlaufs dokumentiert. Man kann beide Werte miteinander verknüpfen. Für den Fall, dass $v_{max} \geq 2$ war, schafften es sogar alle 30 Agenten bis maximal zur 93. Runde das Gebiet zu verlassen. In den Durchläufen, in denen $\leq v_{max} = 1$ war, schafften es bis zu diesem Zeitpunkt maximal die Hälfte der Agenten, denn im Graph ist zu erkennen, dass die Maximalwerte kurz nach Runde 93 bei genau 15 entkommenen Agenten liegen. Diese Sensitivitätsanalyse zeigt damit deutlich, welchen Einfluss die Wahl des Zeitschritts einer Simulation auf deren Ergebnisse hat. Wählt man die simulierte Zeit pro Runde sehr gering (Millisekunden), muss sich auch der Wert von v_{max} reduzieren, wenn die reale Geschwindigkeit in der Umrechnung gleich bleiben soll. Wählt man den Zeitschritt zu klein, kann dies zu vollkommen anderen Ergebnissen führen, wie man in diesem Beispiel gut sehen kann. Für einen Zeitschritt, der zu Werten von $v_{max} \geq 2$ führt, sind die Ergebnisse dichter beisammen. Der Grund, warum bei $v_{max} = 1$ die Ergebnisse so weit von den anderen entfernt liegen, kann verschiedene Gründe haben: Geringere Geschwindigkeiten führen generell auch zu einer kleineren Menge potenzieller Zielzellen, aus denen der Agent wählen kann. Bei ungünstiger Wahl der Agentenkonstanten können die Wahrscheinlichkeiten der einzelnen Zellen dichter beisammen liegen. Somit kann es passieren, dass die Agenten sehr oft ihre Richtung ändern oder Situationen entstehen, in denen sie im Kreis oder sogar vom Ausgang weglaufen. Vor allem, wenn - wie im Beispiel - der Einfluss der Trägheit keine Rolle spielt und Richtungsänderungen damit alle gleich wahrscheinlich sind, kann so etwas passieren. Auch wenn der Einfluss des statischen Floor-Fields im Vergleich zu den anderen Agentenkonstanten zu gering ist, kann ein unkontrolliertes Verhalten der Agenten beobachtet werden. Die Sensitivitätsanalyse kann deshalb auch dabei helfen, die Wahl dieser Parameter zu verbessern. Ändert man in diesem Beispiel den Wert von κ_S von 1.5 auf 5.0, nähern sich die Ergebnisse für $v_{max} = 1$ denen der größeren Geschwindigkeitswerte bereits sichtbar an.

6 Zusammenfassung und Ausblick

F.A.S.T. ist ein mächtiges Modell für Fußgängersimulationen. Gebäude-, Schiffs-, oder Flugzeugevakuierungen lassen sich problemlos entwerfen und realitätsnah simulieren. Optimierungen von bestehenden Fluchtwegen oder Planungen und Konzepte neuer Gebäude sind ebenso möglich. Mit meiner Arbeit will ich nicht nur eine Implementierung dieses Modells beschreiben, sondern vor allem dessen Einsatz auf mobilen Computern vorstellen. Die Idee dahinter war zum einen, die Simulation und Analyse aus den Planungsbüros zum Ort des Geschehens zu transportieren, so dass Experten und Verantwortliche sich vor Ort die Simulationen ansehen und daraus Schlüsse ziehen können. Eine andere Richtung kann der Einsatz dieser mobilen Software in Universitäten und Schulen sein. Junge Menschen können für die Thematik und die verwendete Technik und Mathematik motiviert werden. Die grafische Ausgabe der Simulationen trägt hier viel zu Demonstrationszwecken bei.
Um mein Framework stichhaltig erklären zu können, diente das zweite Kapitel der Einführung in das F.A.S.T.-Modell. Neben den Grundprinzipien, wie dem Aufbau einer Simulationsrunde und den Floor-Fields, ging es auch um Details: Die Formeln zur Wahl eines Ausgangs und einer Zielzelle, die Bewegungsmöglichkeiten der Agenten und die Varianten zur Erstellung der Agentenreihenfolge wurden von mir thematisiert. Die verschiedenen Arten der Ergebnisdarstellung habe ich im letzten Abschnitt des Kapitels angesprochen.
Obwohl das Modell bereits sehr umfänglich ist, wollte ich für meine Implementierung zusätzliche Simulationskomponenten einbauen. Zu den im Modell bereits vorgestellten Erweiterungen zählen die zusätzlichen Stockwerke und die Treppenzellen. Letztere habe ich mit einem dynamischen Parameter versehen, so dass unterschiedliche Arten von Treppen entstehen können, die sich verschieden stark auf die Geschwindigkeit der Agenten auswirken können. Nach demselben Prinzip erweiterte ich die Wandzellen um einen Parameter, der die Zellen, die vor einer Wandzelle liegen, mehr oder weniger attraktiv für Agenten machen. Damit sind verschiedene Typen von Wänden realisierbar. Das Erweitern des Gebiets um zusätzliche Stockwerksebenen, brachte einen weiteren Zellentyp hervor: Teleportzellen. Sie sind nötig, um Agenten zwischen den Ebenen hin- und her transportieren zu können, denn auf Grund der Zweidimensionalität der Gebiete, liegen die Stockwerke nicht über- sondern nebeneinander. Damit mussten aber auch fast alle Distanzberechnungen und die Zielzellensuche der Agenten modifiziert werden, um diesen veränderten Bewegungen Rechnung zu tragen. Eine weitere wichtige Erweiterung ist die Möglichkeit eine Sensitivitätsanalyse mit dem Programm durchzuführen. So ist es möglich zu Simulationsbeginn Regeln zu erstellen, die vor jedem Durchlauf ausgeführt werden. Sie weisen den angegebenen Parametern neue Werte zu, die durch eine Vielzahl verschiedener Wahrscheinlichkeitsverteilungen gewählt werden können.
Der Fokus des vierten Kapitels lag auf der Beschreibung der Umsetzung des Modells und der Erweiterungen. Neben dem grundlegenden Aufbau des Programms und seiner einzelnen

Komponenten, galt es auch die drei Hauptteile der Simulation detailliert zu beschreiben: Initialisierung, Durchführung und Ergebnisspeicherung. Bei der Initialisierung lag der Fokus vor allem auf den Konfigurationsdateien, welche die Szenarien definieren. Die Erklärung ihres Aufbaus und ihrer internen Verarbeitung wurden von zusätzlichen Codeausschnitten und Beispielen unterstützt. Aber auch die Algorithmen für die vielen Distanzberechnungen wurden von mir ausführlich thematisiert. Der Abschnitt über die Simulationsdurchführung widmete sich der Umsetzung der Ausgangs- und Zielzellensuche, dem Finden einer Agentenreihenfolge für die Bewegungen und den Aktualisierungsvorgängen der Statistiken am Ende der Runde. Die gesammelten Resultate waren der Mittelpunkt des letzten Abschnitts. Dort wurde die interne Speicherung und die Möglichkeiten ihrer Darstellung in Form von Graphen, Bildern und Text vorgestellt.
Abschließend zeigte das fünfte Kapitel Ergebnisse von verschiedenen Simulationstests. Die sollten zum einen aufzeigen, welchen Einfluss die Größe der Gebiete und die Anzahl der eingesetzten Agenten auf die Durchführungszeiten und den verbrauchten Speicherplatz haben. Ein zusätzlicher Test offenbarte die Grenze des derzeit simulierbaren mit dem vorliegenden System. Der abschließenden Test beinhaltete eine Sensitivitätsanalyse. Mit ihr konnte man die Auswirkung der Änderung der Maximalgeschwindigkeit von Agenten auf die Ergebnisse einer Simulation sehr gut sehen.

Ausblick

Dieses Framework soll nicht als abschließende Lösung für die Umsetzung des F.A.S.T.-Modells auf mobilen Rechnern gesehen werden. Ganz im Gegenteil: Der Aufbau der Konfigurationsdateien und ihrer Verarbeitung, die Art, wie die Wahrscheinlichkeiten für eine Zielzelle berechnet werden und wie die Agenten sich bewegen ist absichtlich offen von mir implementiert worden. Meine Arbeit soll Ausgangspunkt für zusätzliche Erweiterungen und Anpassungen des Modells sein. Vor allem was die grafische Darstellung von Ergebnissen betrifft, ist in meinem Programm Ausbaumöglichkeit gegeben. In [Kre06, S. 86 f.] sind noch weitere Ausgaben beschrieben, wie z. B. Frustration und Blockierung der Agenten oder die Verteilung der Evakuierungszeiten [Kre06, S. 88]. Auch die Art, wie sich Agenten zur Zeit bewegen, kann modifiziert werden. Das zellenweise Bewegen zur Zielzelle, ist bereits detailliert beschrieben worden ([Kre06, S. 28 ff.]). Eine große Erweiterung wäre der Einbau eines Grafikeditors, mit dessen Hilfe im Programm selbst die Simulationsgebiete bearbeitet werden können. So können schnelle Ideen direkt umgesetzt und getestet werden. Auch der Bau komplett neuer Szenarien kann damit vereinfacht werden, da die Vergabe der Farben für der entsprechenden Zelltypen besser gesteuert und verwaltet werden kann. Programmiertechnisch muss für die in 5.1 angesprochenen Probleme eine Lösung gefunden werden. Vor allem das zweite Problem ist von großer Bedeutung: Die vielen einzelnen Speicherallokationen beim Befüllen der großen Datenstrukturen, wie z. B. dem Zellenarray oder den einzelnen Arrays für die Ausgangs- und Knotenzellendistanzen jeder Zelle sind Ausgangspunkte dieses Problem. Die Erzeugung der vielen `Integer`- oder `Double`-Objekte führen zu den Einsätzen des automatischen Garbage-Collectors, der für die minimalen, aber in der Summe trotzdem langen, Wartezeiten des Hauptprogramms sorgt. Eine Lösung ist sicherlich, die Arrays auf ein Mal zu initialisieren und

erst dann die einzelnen Zell-Objekte zu erstellen. Dies ist aber mit den derzeit verwendeten Datenstrukturen nicht möglich. Hier muss entweder auf andere Datenstrukturen umgestellt werden, oder nativer Code (z. B. in C++) für die Speicherreservierungen eingesetzt werden.

Alles in allem ist dieses Framework für die Simulation kleiner bis mittlerer Szenarien sehr leistungsstark. Auf Grund der Erweiterungen sind die Ergebnisse realitätsnäher geworden und mit der Sensitivitätsanalyse lassen sich die Resultate genauer untersuchen.

Literaturverzeichnis

[An1] What is the NDK? URL http://developer.android.com/tools/sdk/ndk/overview.html. (Zitiert auf Seite 38)

[An2] Developer Tools. URL http://developer.android.com/tools/index.html. (Zitiert auf Seite 37)

[An3] Android Reference Index. URL http://developer.android.com/reference/classes.html. (Zitiert auf Seite 46)

[BKSZ01] C. Burstedde, K. Klauck, A. Schadschneider, J. Zittarz. Simulation of pedestrian dynamics using a 2-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295:507, 2001. doi:10.1016/S0378-4371(01)00141-8. (Zitiert auf den Seiten 9 und 10)

[Cun] S. Cunningham. The Ubiquitous Bresenham Algorithm as a Basis for Graphics Interpolation Processes. California State University Stanislaus. URL http://www.cs.csustan.edu/~rsc/sdsu/Interpolation.pdf. (Zitiert auf den Seiten 33 und 48)

[Dij59] E. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://www.digizeitschriften.de/index.php?id=ppn&PPN=GDZPPN001163248&L=2. (Zitiert auf den Seiten 31, 33 und 51)

[Dre67] H. Dreyfuss. *The Measure of Man - Human Factors in Design*. Whitney Library of Design, ISBN:B-000-7EJK6-O, 1967. (Zitiert auf Seite 9)

[Gat05] P. Gattermann. Grundlagen von Entfluchtungskonzepten. In *Annotation in a talk at the Brandschutz-Fachtagung in Melk*. 2005. (Zitiert auf Seite 9)

[KKN+03] A. Kirchner, H. Klüpfel, K. Nishinari, A. Schadschneider, M. Schreckenberg. Simulation of competitive egress begavior: comparison with aircraft evacuation data. *Physica A: Statistical Mechanics and its Applications*, 324:689–697, 2003. doi:10.1016/S0378-4371(03)00076-1. (Zitiert auf Seite 9)

[Kre06] T. Kretz. *Pedestrian Traffic - Simulation and Experiments*. Dissertation, Universität Duisburg-Essen, 2006. (Zitiert auf den Seiten 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25, 26, 27, 31, 32, 33, 34, 65 und 76)

[KS02] A. Kirchner, A. Schadschneider. Simulation of Evacuation Processes Using a Bionics-inspired Cellular Automaton Model for Pedestrian Dynamics. *Physica A: Statistical Mechanics and its Applications*, 312(1-2):260–276, 2002. doi:10.1016/S0378-4371(02)00857-9. (Zitiert auf Seite 9)

[NKNS04] K. Nishinari, A. Kirchner, A. Namazi, A. Schadschneider. Extended Floor Field CA Model for Evacuation Dynamics. *IEICE Trans. Inf. & Syst.*, E87-D:726–732, 2004. URL `OAI:arXiv.org:cond-mat/0306262`. (Zitiert auf den Seiten 9 und 16)

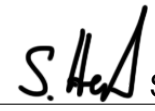
[Sam] Samsung Galaxy Note 10.1 - Technische Daten. URL `http://galaxynote10punkt1.samsung.de/technische-details/`. (Zitiert auf Seite 37)

[Wei92] U. Weidmann. Transporttechnik der Fussgänger, 1992. (Zitiert auf Seite 25)

Alle URLs wurden zuletzt am 12. 03. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Backnang, den 11.03.13  Stephan Herb

Ort, Datum, Unterschift