Zellularautomat zur Simulation von Personenbewegungen

im Rahmen der 3. Modularbeit zur Veranstaltung Modellbildung und Simulation im Sommersemester 2020 an der Hochschule München

Benedikt Beil[1], Benjamin Eder[2], Konstantin Schlosser[3] und Andreas Stiglmeier[4]

[1] *Hochschule München, Master Informatik, SWE* bbeil@hm.edu
[2] *Hochschule München, Master Informatik, SWE* beder@hm.edu
[3] *Hochschule München, Master Informatik, SWE* k.schlosser@hm.edu
[4] *Hochschule München, Master Informatik, SWE* stiglmeier.andreas@hm.edu

Zusammenfassung

Im Zuge dieser Arbeit wurde ein Personenstromsimulator entwickelt, welcher auf einem Zellularautomaten mit einer event-driven Architektur basiert. Der Ziel des Simulators ist dabei das Bewegen von Personen um Hindernisse zu einem Ziel, wobei Personen einen gewünschten Mindestabstand von einander halten wollen. Zu diesem Zweck wurden unterschiedliche Bewegungs-Strategien (Euklidisch, Dijkstra & Fast-Marching) für die Personen implementiert. Anschließend wurde der Simulator mit standardisierten Testfällen des RiMEA-Vereins validiert. Daraus wurde ersichtlich, dass der Simulator, bis auf kleinere Unreinheiten beim Fundamentaldiagramm, den gegebenen Erwartungen entspricht.

Abstract

In the course of this work, a people flow simulator was developed. The simulator is based on a cellular automaton with an event-driven architecture. The goal of the simulator is the movement of people around obstacles to a target, whereby people want to keep a desired minimum distance from each other. For that purpose, different movement strategies (Euclidean, Dijkstra & Fast-Marching) were implemented for the people. The simulator was then validated using standardized test cases from the RiMEA association. This showed that the simulator, except for minor impurities in the fundamental diagram, met the given expectations.

Modularbeit Modellbildung und Simulation (2020)

Veranstaltung:
Modellbildung und Simulation

Professorin:
Prof. Dr. Gerta Köster

Inhaltsverzeichnis

1 Einleitung (Eder)

Die Simulation von Menschenmengen hat zahlreiche Anwendungsgebiete, wie zur Minimierung der gesundheitlichen Risiken von Personen in oft vollen öffentlichen Einrichtungen bei einer möglichen Evakuierung (Sarmady, Haron und Talib 2010; Fu *u. a.* 2014; Klüpfel 2003). Weitere Anwendungen umfassen die Minimierung von Wartezeiten oder Steuerung von Personenbewegungen (Klüpfel 2003, S. 1-2).

1.1 Zellularautomaten

Zellularautomaten sind ein weitverbreitetes diskretes mathematisches Modell, welches unter anderen im Sozialbereich, Umwelt und Wirtschaft verwendet wird (Zhou 2009, S. 1). Konkret wird das Modell bereits zur Simulation von Personen- oder Verkehrsbewegungen eingesetzt (S. 1). Neben gewissen Schwächen, wie oftmals der Vernachlässigung von Unterschieden zwischen Individuen, gelten Zellularautomaten als praktisches, effizientes und einfach zu implementierendes Simulationsmodell (Fu *u. a.* 2014, S. 1).

1.2 Hintergrund

Im Zuge der Veranstaltung *„Modellbildung und Simulation“* an der *Hochschule München für angewandte Wissenschaften* wollen wir einen solchen Zellularautomaten zur Simulation von Personenbewegungen implementieren und anwenden. Die Anwendung findet anhand ausgewählter Testfälle statt, welche angelehnt an die Tests des “RiMEA e.V.” (2020) sind und uns zur Modellvalidierung dienen.

1.3 Aufbau

Um dem Leser einen groben Überblick zu verschaffen, umreißen wir den Aufbau der Arbeit kurz.

Auf die Einleitung folgt eine Vorstellung unserer verwendeten Methoden und Werkzeuge. Diese umfassen die verwendeten Technologien und einer Beschreibung des Modells mitsamt den Regeln, welchen die Simulation folgen soll.

Nachfolgend stellen wir im Ergebniskapitel die implementierten Werkzeuge vor. Wir nehmen vorweg, dass es sich neben dem Simulator selbst, um eine grafische Benutzeroberfläche zur Erstellung von Simulationsszenarien sowie ein Command Line Interface zur Ausführung Ersterer handelt. Des Weiteren präsentieren wir die bereits erwähnten Testfälle, werten diese mit dem entstandenen Simulator aus und interpretieren die Ergebnisse.

Abschließend fassen wir die erhaltenen Erkenntnisse zusammen und bewerten diese, gefolgt von weiteren Untersuchungsvorschlägen.

2 Methoden

In diesem Kapitel präsentieren wir die verwendeten Werkzeuge und Methoden. Dazu gehört vor allem die Beschreibung des Modells, welches mit einem Zellularautomaten realisiert wird, sowie die dazugehörigen Simulationsregeln.

2.1 Verwendete Werkzeuge (Schlosser)

Um in weiteren Kapiteln unser Modell und den Aufbau des Programms beschreiben zu können, werden zuerst die verwendeten Werkzeuge, Bibliotheken und die verwendete Programmiersprache vorgestellt. Diese wurden ausgewählt, um größtmögliche Kompatibilität mit verschiedenen Betriebssystemen zu gewährleisten und den Compilierungsprozess zu vereinfachen. Folgende Tools werden hauptsächlich von uns verwendet:

- Java
- Gradle
- JavaFX
- Apache Commons Math
- Apache Commons CLI

Die genaue Funktionen und die Gründe für die speziellen Entscheidungen sind in den folgenden Abschnitten erläutert.

Java

Wir haben uns als Team für die Programmiersprache Java entschieden, da wir mit dieser mehr Erfahrung haben und komfortabler im Umgang mit ihr sind. Weiterhin ist Java weit verbreitet, wodurch es viele Bibliotheken und Tools gibt, die für unser Projekt sehr hilfreich sind. Java erlaubt es uns, das fertiggestellte Projekt für verschiedene Betriebssysteme bereitzustellen und sogar als eine, ausführbare Datei zu exportieren.

Wir haben uns weiterhin entschieden, Java Modules zu verwenden. Das erlaubt es uns, unser Programm besser zu unterteilen und Abhängigkeiten zu trennen. Dies erfordert aber auch die Unterstützung von Java Modules aller Bibliotheken, was ein Problem bei älteren Bibliotheken darstellen kann. Über dieses Problem und dessen Lösung werden wir in den folgenden Abschnitten ebenfalls berichten.

Verschiedene Test-, Build- und UI-Frameworks ermöglichen es uns, zwischen ihnen auszuwählen und die Tools zu verwenden, die für unser Projekt uns am geeignetsten erscheinen.

Auch diese Tools und die verwendeten Bibliotheken werden in diesem Kapitel erläutert.

Gradle

Gradle ist ein Buildtool, welches es uns erlaubt komplexe Buildprozesse zu automatisieren und reproduzierbar zu halten. Es speichert Zwischenergebnisse, wenn möglich, um zukünftige Erstellungsprozesse zu verschnellern. Weiterhin automatisiert es das Herunterladen und Bauen eventueller Anhängigkeiten und hält diese aktuell. Unsere verschiedenen Programme können unterschiedliche Abhängigkeiten haben, sodass wir die tatsächlich erstellte Datei so klein wie möglich halten können.

Es erlaubt uns auch, Java Module aufzusetzen, die wir schon im vorhergehenden Kapitel angesprochen haben. Allerdings sind nicht alle Bibliotheken aktuell genug, um die benötigte Unterstützung durch die Autoren erhalten zu haben. Daher haben wir ein Plugin konfiguriert, um diese Module automatisch zu bauen und bereitzustellen.

JavaFX

Um die Oberfläche unseres Programms zu erstellen, benutzen wir das Framework JavaFX. Ein Vorteil dieses Frameworks ist, dass es viele verschiendene, bereits programmierte UI-Elemente gibt, die wir flexibel einsetzbar sind. Dazu gehören nicht nur Elemente wie Knöpfe, sondern auch verschiene Typen von Graphen.

Mit JavaFX ist es uns möglich, verschiedenste Oberflächen programmatisch und deklarativ zu erstellen. Wir haben uns entschieden, die Oberflächen programmatisch zu erstellen. Das erlaubt es uns, die Nutzeroberfläche dynamisch zu erstellen und verschiedene Ereignisse direkt im Programmcode zu behandeln. So müssen wir auch keine zusätzlichen Ressourcen-Dateien berücksichtigen und die Oberfläche wird in einer Datei erstellt. Ebenfalls bekommen wir die Möglichkeit, durch diese Bibliothek die erstellten Oberflächen mittels Themen anzupassen.

Leider hat JavaFX einen Nachteil gegenüber ähnlichen UI-Bibliotheken für andere Programmiersprachen, wie die sehr bekannte Python-Bibliothek matplotlib. Es ist nicht möglich, hochauflösende Diagramme, beispielsweise für Publikationen, als SVG-Datei zu exportieren. Wir umgehen diesen Punkt, indem wir wichtige Datenpunkte über verschiedene Logger exportieren und dann extern plotten.

Diese Bibliothek wird für den Nutzer unseres Programms automatisch durch Gradle heruntergeladen und konfiguriert, sodass keine manuelle Installation von Abhängigkeiten erfolgen muss.

Apache Commons Math

Die Bibliotheken von Apache Commons sind unter den wahrscheinlich am häufigsten verwendeten Bibliotheken für Java. Sie werden in vielen Projekten eingesetzt, da sie nur wenig zusätzlichen Speicher brauchen und wichtige Tools und Funktionalitäten implementieren.

Daher haben wir uns entschieden, die Mathematikbibliothek aus dieser Gruppe zu verwenden. Das erlaubt es uns, bereits getesteten Code zu verwenden und ebenso unseren Code übersichtlicher zu halten.

Daher nutzt der Simulator sie extensiv um verschiedene, statistische Methoden und Verteilungen zur Verwendung zu haben.

Auch diese Bibliothek wird durch Gradle bereitgestellt. Allerdings muss noch zusätzlich konfiguriert werden, welche Module durch sie exportiert werden, da diese Funktionalität noch nicht nativ unterstützt wird.

Pico CLI

Um die Bereitstellung unseres Kommandozeileninterfaces zu vereinfachen, nutzen wir die Bibliothek Pico CLI. Sie erlaubt es uns, durch Annotationen einfach und übersichtlich Argumente zu definieren, deren Position festzulegen oder, im Falle von optionalen Argumenten, die Optionen für dieses Argument zu erstellen. Durch diese Bibliothek können wir das Interface in nur einer übersichtlichen Klasse definieren.

Auch hier ist Gradle für die Bereitstellung der Abhängigkeiten verantwortlich.

JUnit

Um unseren Code zu testen und wichtige Metriken einsehen zu können, wie die erreichte Codeabdeckung, nutzen wir das bekannte Testframework JUnit 5.

Dieses Framework ermöglicht eine einfache Erstellung von Testmethoden, -klassen und Testläufen. So können wir einzelne Methoden, Klassen oder auch ganze Pakete testen und die Ergebnisse übersichtlich darstellen.

Unsere Codeabdeckung kann, gegebenenfalls, mit einem CI-Tool automatisch berechnet werden.

Auch hier wird Gradle von uns verwendet, um die Bibliothek bereitzustellen.

2.2 Modellbeschreibung (Beil Benedikt)

Zellularautomat

Wie aus der Überschrift zu entnehmen ist, verwenden wir ein Zellularautomaten für unser Modell. Der Zellularautomat besitzt eine rechteckige Form und besteht aus Quadraten. Die Maße der Quadrate sind wie die Anzahl der Quadrate konfigurierbar. Eine beispielhafte Belegung des Zellularautomates ist in der Abbildung 1 zu sehen. Weil die Simulation noch nicht begonnen hat, befinden sich noch keine Personen auf den Feld.

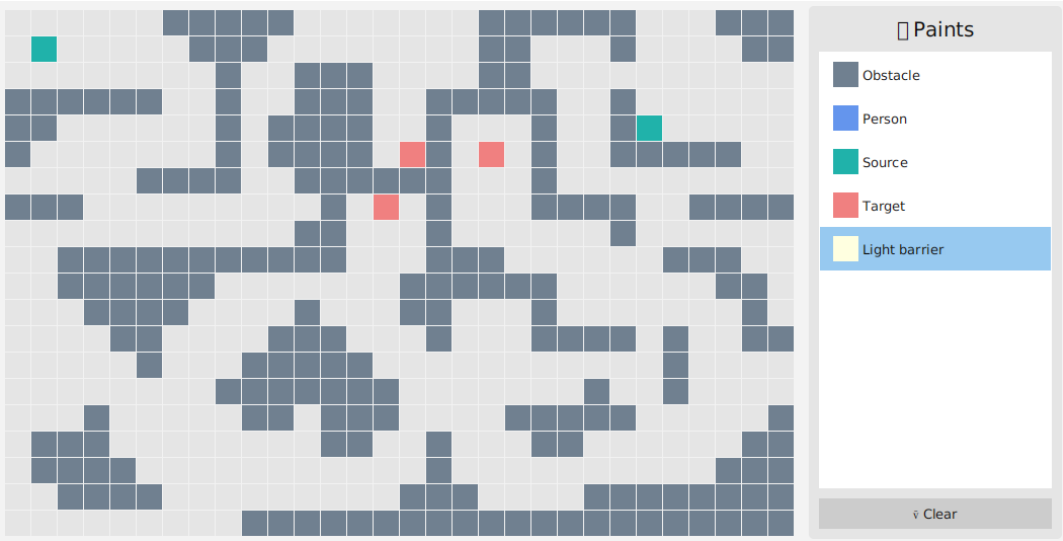


Abbildung 1. Beispielhafte Belegung eines Zellularautomaten

Zellobjekte

Eine Zelle ist entweder von einem Zellobjekt belegt, oder ist leer. Eine Zelle kann nicht von mehreren Objekten zeitgleich belegt sein. Das Objekt belegt die komplette Zelle. Es gibt folgende Zellobjekte mit ihren Eigenschaften.

Hindernis Ein Hindernis kann nicht betreten werden. Es repräsentiert jedes belibiges Hindernis. Zum Beispiel eine Wand oder Möbel.

Person Eine Person wird von einer Quelle hervorgebracht. Jeder Person hat ein Ziel, welches er mit seiner Wunschgeschwindigkeit erreichen will. Dazu springt er von Zelle zu Zelle. Eine Person kann dazu alle 8 Nachbarzellen (Moore-Umgebung) betreten. Personen halten Abstand voneinander. Mehr dazu im Abschnitt 2.2.

Quelle Eine Quelle bringt Personen in einen vorgegebene Zeitabstand hervor. Dieser Zeitabstand kann entweder ein fester Wert sein, oder eine Poisson-Verteilung. Die Personen erscheinen auf eine zufällige gewählte Nachbarzelle. Die Anzahl an zu erzeugenden Personen kann begrentzt werden.

Ziel Wird eine Person hervorgebracht, wird ein zufälliges Ziel gewählt, welches er erreichen will. Je nach dem, wie das Ziel konfiguriert ist, wird die Person beim Erreichen entfernt, oder durch eine zufällige Quelle erneut hervorgebracht.

Lichtschranke Die Lichtschranke ist ein einzigartiges Objekt. Es ist das einzige Objekt, welches betreten werden kann. Es hat indirekt einen Einfluss auf die Simulation und dient rein zur Messung des Personenflusses. Dies wird für das Fundamentaldigarmm benötigt. Mehr dazu im Abschnitt 3.4.

Wunschgeschwindigkeit

Wie im Abschnitt 2.2 beschrieben, wollen Personen ihr Ziel mit Ihrer Wunschgeschwindigkeit erreichen. Diese wird aus einer Normalverteilung mit $\mu = 1.34\frac{m}{s}$ und $\sigma = 0.26\frac{m}{s}$ gewählt. Um negativ und positv unendlich große Geschwindigkeiten zu vermeiden, wurde beide Enden gekappt. Der Minimalwert beträgt $1.08\frac{m}{s} = \mu - \sigma$. Der Maximalwert beträgt $1.6\frac{m}{s} = \mu + \sigma$. Wir haben die ein σ Umgebung gewält, da hier die Varianz zwischen den Simulationen verhältnismäßig gering ist. Dadurch werden Simulationen vergleichbarer, da die Ausreißer nicht so extrem sind. Die Verteilung ist in der Abbildung 2 dargestellt. Der Event-Driven-Scheduler sorgt dafür, dass die Personen sich in Ihrer Wunschgeschwindigkeit vortbewegen. Personen mit einer höheren Wunschgeschwindikeit werden häufiger zum Springen aufgefordert, wie langsamere Personen. Mehr zum Scheduler im Abschnitt 2.3.

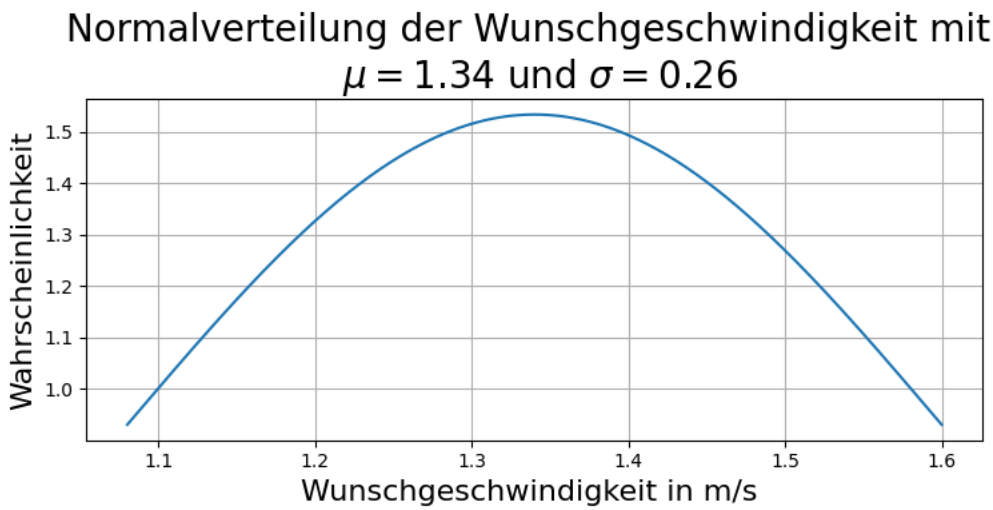


Abbildung 2. Verteilung der Wungschgeschwindigkeiten

Wahl des nächsten Schrittes

Eine Person kann alle 8 Nachbarzellen betreten um sein Ziel näher zu kommen. Die Wahl, welche dieser 8 Nachbarzellen betreten werden soll, wird über das Potential bestimmt. Jede Zelle besitzt ein Potential größer 0, außer die Ziele. Ziele haben immer ein Potential von 0. Umso weiter die Zelle vom Ziel entfernt ist, umso größer ist das Potential. Eine Person wählt die Zelle mit den größten Potentialabstieg. Gibt es in einer Simulation nur ein Ziel, so wird das Potential aller freien Zellen graphisch dargestellt, wenn die Simulaiton pausiert wird. Umso heller die Zelle dargestellt wird, umso geringer ist ihr Potential. Mehr dazu im Abschnitt 3.1.In diesem Modell kann zwischen drei Methoden gewählt werden, um das Potential der Zellen zu bestimmen.

Euklidischer Abstand Der euklidescher Abstand ist die einfachste Methode zum Bestimmen der Potentiale. Das Potential entspricht der Entfernung zwischen einer Zelle und dem Ziel. Dabei werden Hindernisse auf den Weg nicht beachtet. Die Abbildung 3 zeigt das Ergebnis der Berechnung. Umso näher eine Zelle dem Ziel ist, umso geringer ist ihr Potential. Dabei wird die Wand aus Hindernissen nicht berücksichtigt.

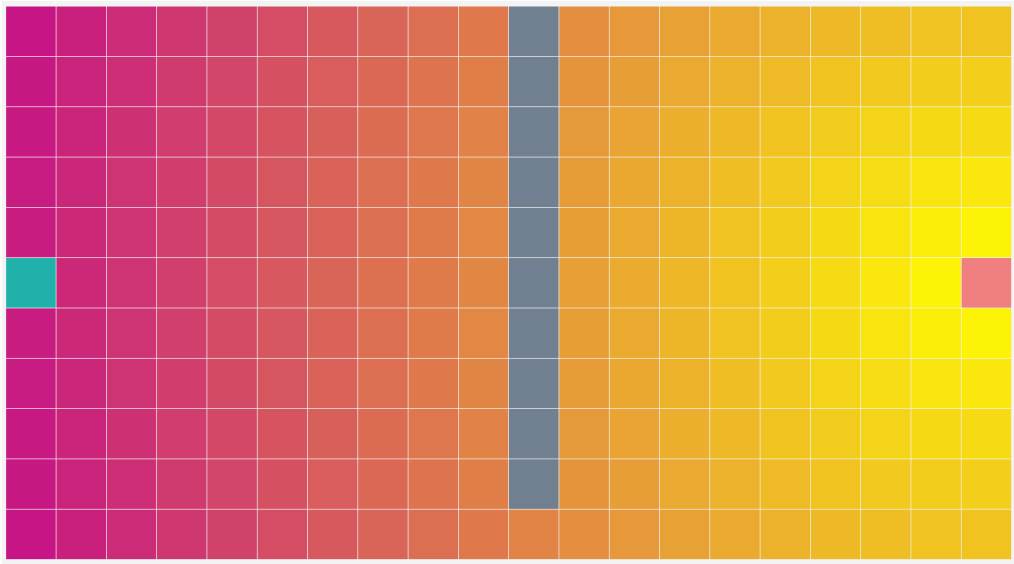


Abbildung 3. Potenitale bestimmt durch den euklidischen Abstand

Dijkstra-Algorithmus Bei dem Dijkstra Algorithmus wird jede frei Zelle durch einen Knoten repräsentiert. Jeder Knoten wird mit seinen bis zu 8 benachbarten Konten verbunden. Dabei wird sich die Distanz zu diesem Knoten gemerkt. Das Potential einer Zelle ergibt sich aus der kürzesten Summe der Verbindungsdistanzen zu dem Ziel. Da belegte Zellen wie zum Beispiel Hindernisse nicht als Knoten repräsentiert werden, werden Hindernisse beim Dijkstra-Algorithmus berücksichtigt. Daher ist bei der Abbildung 4 der obere Bereich links der Mauer dunkler wie bei der Bestimmung durch den euklidischen Abstand.

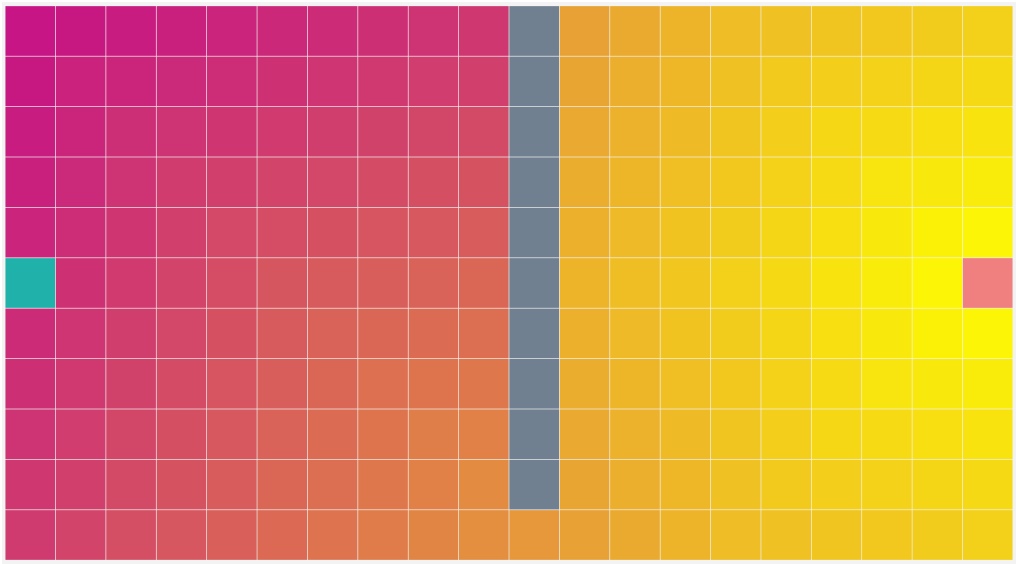


Abbildung 4. Potenitale bestimmt durch den Dijkstra Algorithmus

Fast marching method **(Eder)** Neben der euklidischen Distanzmetrik und dem Dijkstra Algorithmus verwenden wir die *Fast marching method* zur Berechnung der Potenzialmatrix. Der implementierte Algorithmus basiert auf der Beschreibung aus Rasch und Satzger (2008), während die Berechnung der Updatewerte mit der zweidimensionalen Approximation der Eikonalgleichung nach "Eikonal equation - Wikipedia" (2020) erfolgt. Nachfolgend wollen wir einen groben Überblick über die Funktionsweise geben.

1. **Initialisierung**: Die Ergebnismatrix R wird erstellt. Dabei initialisieren wir die Zelle, auf der sich das Ziel befindet, mit dem Wert 0. Zellen, welche Hindernisse enthalten werden vorab bereits mit dem Wert `Double.MAX_VALUE` belegt, um auszudrücken, dass diese nicht in realistischer Zeit erreicht werden können. Der Rest der Zellen wird mit ∞ belegt. Diese werden später während des Algorithmus berechnet.
2. Eine `PriorityQueue` *considered* wird neben R als grundlegende Datenstruktur verwendet. Diese enthält zum Startzustand lediglich die Zelle mit dem Updatewert 0 in R. Von dieser Zelle soll sich eine Welle ausbreiten.
3. **Algorithmus-Schritt** (Solange *considered* noch Zellen enthält):
   (a) Nehme die nächste Zelle mit dem niedrigsten Wert aus *considered*. Im folgenden C genannt.
   (b) Der Updatewert v von C wird an seiner Position in R eingefügt.
   (c) Für alle direkten Nachbarn von C (Moore-Nachbarschaft), welche noch den Wert ∞ in R besitzen, berechnen wir die Updatewerte v mithilfe der zweidimensionalen Approximation der Eikonalgleichung und R. Dabei nehmen für alle Zellen dieselbe Geschwindigkeit von $1,0$ an. Existiert der Nachbar noch nicht in *considered* oder ist v geringer als ein bestehender Wert desselben Nachbarn, wird die Nachbarzelle entweder zu *considered* hinzugefügt, oder der Updatewert in der `PriorityQueue` aktualisiert.

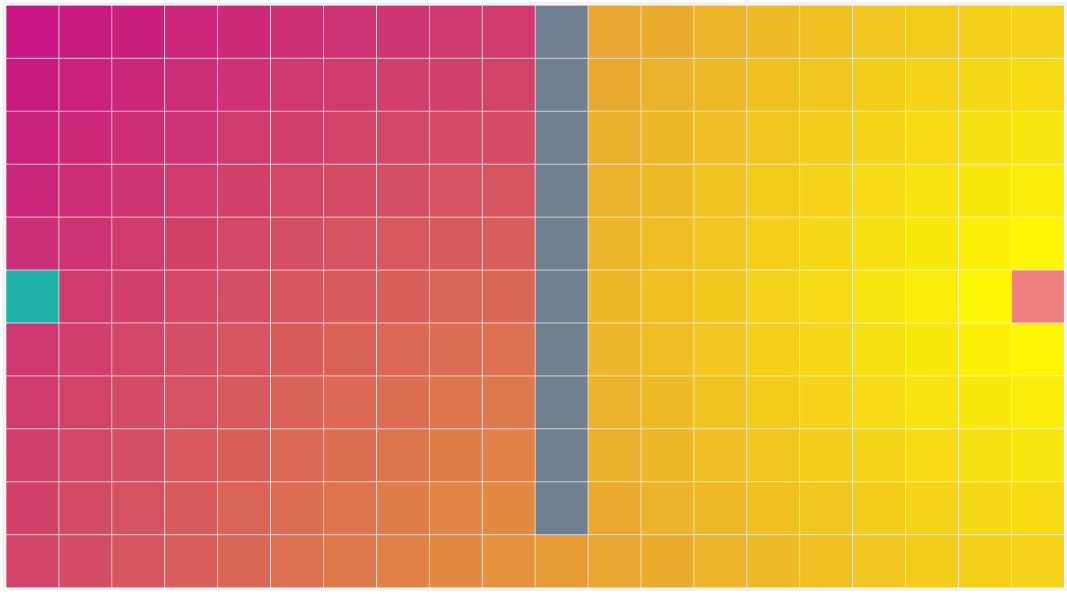


Abbildung 5. Potenziale bestimmt durch die Fast marching method

Personen halten Abstand

Die Wahl des nächsten Schrittes wird über Potentiale bestimmt. Dabei haben andere Personen auch einen Einfluss auf die Berechnung der Potentialmatrx. Jede Person trägt ein Abstoßungspotential mit sich herum. Dieses Abstoßungspotential wird ermittelt durch den Glättungskern von Friedrichs. Dieser wird durch folgende Funktion beschrieben.

$$u_p(d) = \begin{cases} -h\, exp(\frac{1}{(d/w)^2-1}) & \text{if } |d| < w \\ 0 & \text{else} \end{cases} \tag{1}$$

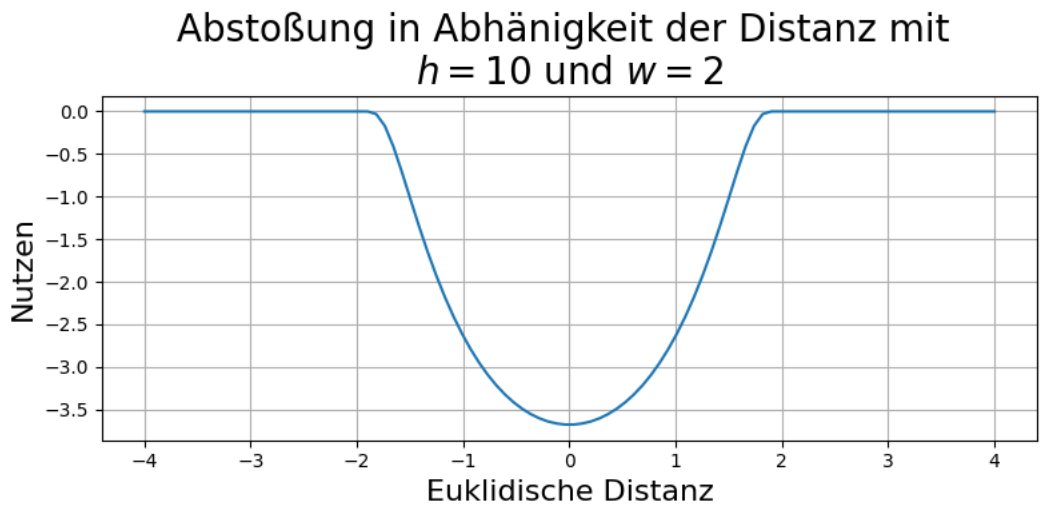


Abbildung 6. Verlauf des Abstoßungspotentials in Abhänigkeit der Distanz

Dabei ist d der euklidischer Abstand zwischen den Personen, h die konfigurierbare Abstoßungsstärke und w der konfigurierbare Einflussradius. Die Abbildung 7 demonstriert den Einfluss des Abstoßungspotentials einer Person für $w = 2$ und $h = 10$. Durch das Abstoßungspotental der Person, steigt das Potential der benachbarten Zellen, wodurch diese dunkler dargestellt werden.

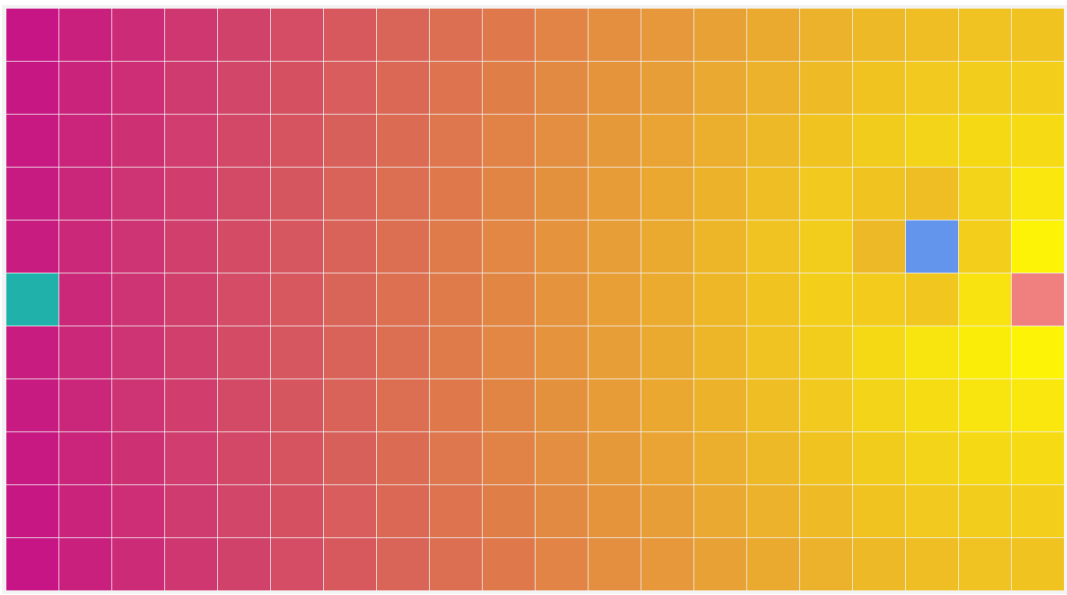


Abbildung 7. Potenitale zur darstellung des Abstoßungspotentials

Ein alternatives Modell für die Abstoßung ist der Modell „personal space“ von Hall. Es beschreibt, wie Nah wir Personen an uns ran lassen in Abhängigkeit davon, wie gut wir sie kennen. Geliebte, Familie und enge Freunde werden dem Bereich von Berühren bis zu ca. 46 cm zugelassen. Der persöhnliche Bereich besitzt eine Radius von ca. 122 cm was ungefähr die Länge eines ausgesteckten Armes entspricht Hall 1966. Das Modell ist zu genau für unser grob aufgelösten Zellularautomaten. Daher ist dieses Abstandsmodell für unsere Simulation ungeeignet.

Bezug des Modells „personal space“ von Hall auf Corona Das BZgA empfiehlt in Zeiten von Corona eine Mindestabstand von 1,5 Metern “Verhaltensregeln und -empfehlungen zum Schutz vor dem Coronavirus im Alltag und im Miteinander” 2020. Menschen versuchen laut Hall einen Abstand von einer Handlänge (ca. 1,22 cm) zu fremden Personen zu halten Hall 1966. Gibt man Menschen die Möglichkeit ausreichend Distanze zu halten, so wollen diese von sich aus schon eine wirksamme Distanz einhalten. Es gilt Situalionen zu verbieten, wo Abstand zu halten nicht möglich ist.

Ungeduld **(Eder)**
Die bisherigen Regeln beschreiben die Bewegung von Personen unzureichend. Es stellt sich die Frage, was Personen machen, wenn sie keinen verbessernden Bewegungsschritt machen, sich also nur weiter vom Ziel entfernen können. Das kann in unserem Simulator passieren, wenn sich beispielsweise bereits eine andere Person in Wunschrichtung befindet oder die kumulative Abstoßung von Personen in der nahen Umgebung, dem tatsächlichen Nutzen in der nächsten Zielzelle überwiegt.

Zwei Verhaltens Extrema sind denkbar:
1. **Stehen bleiben**, bis der Wunschweg wieder freigegeben wird.
2. **Um jeden Preis bewegen**, auch wenn das eine Verschlechterung (Entfernung von Ziel) bedeutet.

Beide sind nicht unbedingt wünschenswert: Beim ersten Verhalten laufen wir Gefahr die Simulation *einzufrieren*, wenn sich kein Individuum mehr bewegen möchte. Im Gegensatz dazu bewegen sich die Personen mit dem zweiten Verhalten ständig umher, vergleichbar mit einem Läufer, der *möglichst in Bewegung bleiben* will, auch wenn dadurch die Strecke länger wird. Ein durchschnittlicher Mensch wird jedoch nicht sofort einen anderen Weg suchen, sondern erst einige Zeit abwarten, ob der Weg vor ihm wieder freigegeben wird.

In unserem Simulationsmodell haben wir uns dafür entschieden, die **Ungeduld** von Menschen zu modellieren, um einen Kompromiss zwischen den beiden Verhaltensextremen zu finden. Muss ein Individuum warten, weil sich keine Wegverbesserungsmöglichkeit bietet, belastet das seine Geduld. Jede Person erhält dafür in der Simulation einen Wert für die Geduld. Reicht die Geduld nicht mehr, trifft das Individuum auch suboptimale Entscheidungen.

Konkret ist die Geduld eine ganze Zahl, die falls sich die Person nicht verbessernd bewegen kann, heruntergezählt wird. Erreicht diese Zahl das Minimum 0 oder wird der Wunschweg freigegeben, bewegt sich das Individuum wieder und setzt seine Geduld auf den ursprünglichen Wert zurück.

In Fu *u. a.* (2014) auf Seite 1 wird angeführt, dass Zellularautomaten zur Simulation von Personenbewegungen oft das Problem haben, dass sich die Agenten zu wenig unterscheiden. Daher

halten wir es für sinnvoll zur Berechnung der Geduld wieder eine **Normalverteilung** heranzuziehen. Die Verteilung wird von den beiden Parametern Erwartungswert μ und „maxDeviation“, also der maximalen Abweichung, bestimmt. Somit wird auch der minimale ($\mu - maxDeviation$) sowie maximale ($\mu + maxDeviation$) Wert, welche aus der Verteilung gezogen werden können, begrenzt. Theoretisch sind mit einer Normalverteilung auch kleinere und größere Werte möglich, diese schneiden wir jedoch ab. Die Standardabweichung σ berechnen wir, indem wir die gegebene „maxDeviation“ durch 3 teilen. Damit erreichen wir wie in Abbildung 8 zu sehen ist, eine große Abdeckung des Wertebereichs der Normalverteilung, ohne zu viel abzuschneiden.

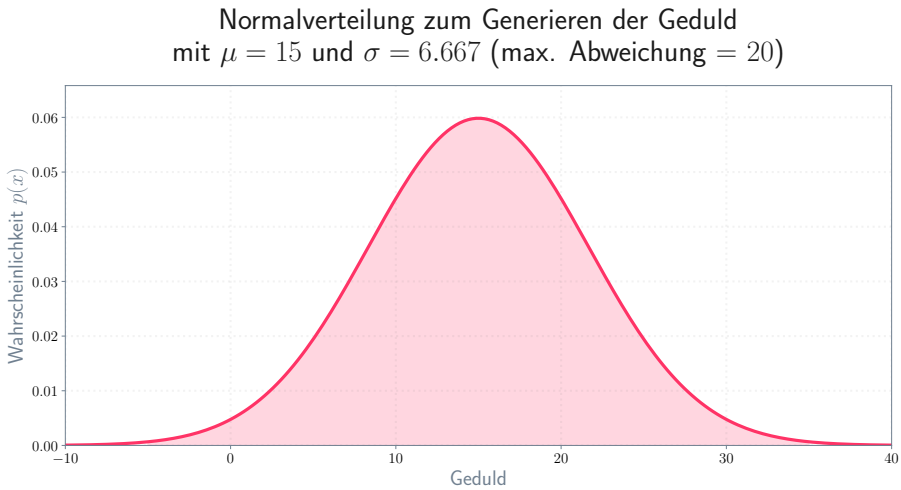


Abbildung 8. Normalverteilung, welche zur Generierung der Geduld verwendet wird. Mit exemplarischen Parametern.

2.3 Architektur (Schlosser)

Unsere Anwendung wurde, wie in vorhergehenden Kapiteln (siehe, u.A. 2.1) beschrieben, in mehrere seperate Projekte eingeteilt.

Diese Einteilung wurde aufgrund mehrerer Gründe vorgenommen. Ein grundlegendes Muster der Softwarearchitektur ist das sogenannte *KISS*-Prinzip. Durch die Verwendung des ”keep it simple, stuipid”Grundsatzes ist es für andere Entwickler einfacher zu verstehen, was ein Stück Software macht und welche Aufgaben es erfüllt. Ein weiteres Muster ist *Separation of Concerns*. Dies bedeutet, dass ein Teilstück einer Software, wie ein Modul oder Paket, nur für eine Aufgabe zuständig sein soll. Goll (2019)

Geleitet von diesen Prinzipien haben wir die Software in mehrere große Gruppen eingeteilt: Die Simulationslogik, das Userinterface und das Kommandozeileninterface.

Die Anwendung der angesprochenen Prinzipien erlaubt es uns, die Teilprojekte klein zu halten, sowie uns und eventuellen zukünfigen Entwicklern die Arbeit zu erleichtern. Der Code wird verständlicher, da Abschnitte, die das gleiche Ziel haben, nicht nur im gleichen Ordner, sondern auch im gleichen Projekt liegen.

Weiterhin könnten so einfach Technologien ausgetauscht werden, wenn beispielweise eine andere Benutzeroberfläche gewünscht wird.

Die drei seperaten Projekte unseres Programms lauten somit `sim`, `ui` und `cli`.

Im Projekt `sim` ist unsere Simulationslogik enthalten. Dieses Projekt ist unabhängig von den beiden anderen und kann auch einzeln gebaut werden. Hier ist unsere gesamte ”Business-Logikënthalten und der eigentliche Simulator - ohne eigene Oberfläche - implementiert. Es stellt einige Logger bereit, die Daten der Simulation in Dateien schreiben können, um externe Interpretation zuzulassen.

Um eine Nutzeroberfläche bieten zu können, wurde das Projekt `ui` erstellt. Es bietet sowohl die Möglichkeit der Erstellung von Szenarien, und des Durchführens von einzelnen Simulationen.

Dieses Teilprojekt braucht unsere grundlegende Logik in `sim` als Abhängigkeit. Diese wird genutzt, um die eigentliche Simulation durchzuführen, während der Code in diesem Projekt die Sicht auf die Daten und die Kontrolle der auszuführenden Operationen übernimmt. Diese Funktionalität kann als ein getrenntes MVC-Muster verstanden werden. Voorhees (2020)

Das Durchführen und Mitteln über mehrere Läufe erleichtert unser Projekt `cli`. Dieses beöntigt ebenfalls die Simulationslogik als Abhängigkeit und ermöglicht ein flexibles und mächtiges Konfigurieren von Durchläufen.

Auf die letzten beiden Module wird in späteren Kapiteln näher eingegangen, da diese dynamisch aus unserem Entwicklungszyklus entstanden sind. Daher wurde die Architektur im Voraus nicht explizit festgelegt, um sie unseren Anforderungen anpassen zu können.

Diese Vorhergehensweise wäre für die Implementierung der Simulationslogik jedoch fatal gewesen, da diese weitaus komplexer als die anderen beiden Teilprojekte ist. Daher soll sie in dem folgenden Abschnitt näher beschrieben werden.

Beschreibung der Architektur der Simulationslogik

Wir haben uns für einen Event-driven Ansatz des Simulators entschieden. Es ermöglicht uns, Klassen einfach auszutauschen und einzubauen sowie unsere Architektur zu entkoppeln.

Der zentrale Baustein der Simulationslogik ist daher unser Scheduler. Er ist dafür verantwortlich, dass die Schritte der Fußgänger ausgeführt und neue Fußgänger entstehen.

Dazu nutzt er Events, die jeweils einen EventHandler haben. Diese Events können durch unseren Scheduler in eine Reihe angehängt werden, die dann nacheinander, sortiert nach ihrem Ausführungszeitpunkt, abgearbeitet werden. Die Bezeihungen der Klassen untereinander sind grob im Diagramm 9 dargestellt.

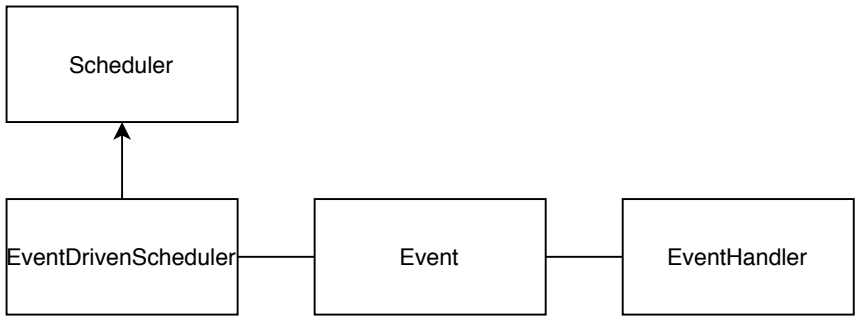


Abbildung 9. Beziehungen der Klassen des Schedulers untereinander.

Ein EventHandler kann, durch die Deklaration als funktionelles Interface, fast jede Methode einer Klasse der Simulationslogik sein. So ist beispielsweise die Methode der Quelle, die neue Personen erschafft ein EventHandler ebenso wie die Methode der Person, die das Laufen kontrolliert.

Diese berechnen bei einer Ausführung jeweils den gewünschten, nächsten Zeitpunkt der Ausführung und reihen sich wieder in die Liste des Schedulers ein. Der Scheduler sortiert diese Liste aufsteigend nach dem Ausführungszeitpunkt und wählt das nächste, auszuführende Event aus. Dadurch schreitet die Simulationslogik nicht konstant in der Zeit voran, sondern ßpringt"von Event zu Event.

Unser Simulator arbeitet mit verschiedenen Objekten, wie Personen, Quellen und Zielen. Diese stammen in unserer Implementierung einer gemeinsamen Oberklasse ab, um die Abstraktion der verwendenden Klassen gewährleisten zu können. Ein Überblick der verfügbaren Objekte ist in Bild 10 zu finden.

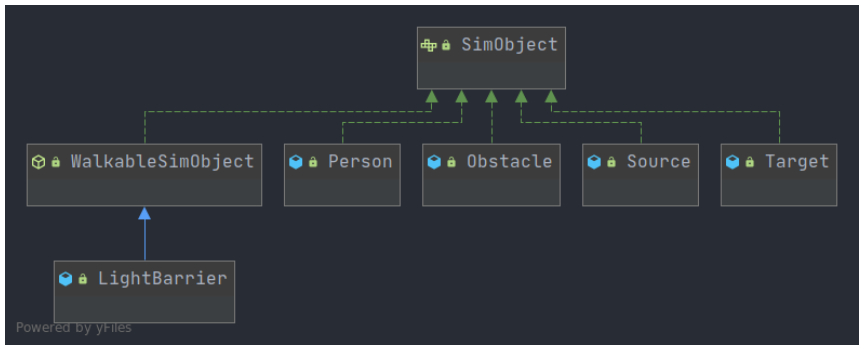


Abbildung 10. Verschiedene Objekte der Simulation und deren Beziehungen untereinander.

Hier setzen wir auf die gemeinsame Oberklasse, die die Verwendung ohne Kentnisse der genauen Art der Objekte, beispielsweise im Scheduler, ermöglicht.

Um die Implementierung der verschiedenen Möglichkeiten der Personenbewegung, -erstellung und ähnlicher, konfigurierbaren Funktionen des Simulators zu ermöglichen, setzen wir auf das Strategy-Muster.

Dieses Muster erlaubt es uns, mehrere konkrete Implementierungen für, zum Beispiel, die Strategie der Personenerstellung zur Verfügung zu stellen. Diese implementieren ein gemeinsames Interface und erweitern eine spezifische Bassisklasse, die grundlegende Funktionalität zur Verfügung stellt.

Da dieses Muster an vielen Stellen in unserer Anwendung vorkommt, soll nur ein Beispiel genauer betrachtet werden.

Ausgewählt wurden dazu die Bewegungsstrategien, da diese die meisten konkreten Implementierungen bieten und somit das Muster am besten darstellen. Ein Klassendiagramm ist in 11 zu sehen anhand diesem die Erklärungen erfolgen werden.

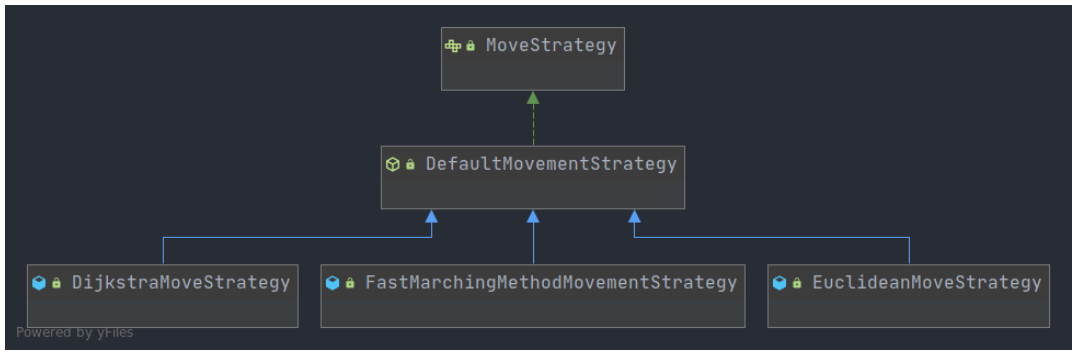


Abbildung 11. Beziehungen der Bewegungsstrategien untereinander. Zu sehen ist in absteigender Reihenfolge das gemeinsame Interface, die Bassisklasse und die konkreten Implementierungen.

Im Diagramm ist die zu sehen, dass die Klassen alle das Interface `MoveStrategy` implementieren. Dieses legt den Grundstein für die gemeinsame Funktionalität und legt Methoden fest, die alle Implementierungen zur Verfügung stellen müssen. Dadurch kann dieses Interface unabhängig der konkreten Implmentierung in unserem Scheduler und anderen Klassen im Modell genutzt werden. So bekommt jede Person beim Erstellen durch die Senke eine Strategie zugewiesen, die die Bewegung dieser Instanz regelt.

Weiterhin wird durch die abstrakte Basisklasse `DefaultMovementStrategy` grundlegende Funktionalität sichergestellt. So werden hier die Ziele aus einem Simulationszustand herausgesucht, das Feld mit dem größten Zuwachs des Nutzens als Ziel der Bewegung auswählt, die Person tatsächlich dahin bewegt und das Vorgehen der Personen bei nicht möglicher Bewegung festgelegt.

Durch dieses Vorgehen muss eine konkrete Implementierung einer Bewegungsstrategie ”nurëine Methode zur Verfügung stellen, die eine Potentialmatrix für die konkrete Person und den konkreten Simulationsschritt berechnet.

Es wird somit soweit wie möglich eine Abstraktion dieser Funktionalität sichergestellt.

Wie also in den vorhergehenden Abschnitten erwähnt, bieten die konkreten Implementierungen, in diesem Fall `EuclidianMoveStrategy`, `DjikstraMoveStrategy` und `FastMarchingMoveStrategy` eine unterschiedliche Methode `calculateBasePotential` an.

Wir denken, dass dies die größtmögliche Abstraktion für diese Logik ist.

Ähnlich diesem Vorgehen wurden auch Strategien für das Erstellen, das Entfernen, die Geschwindigkeit und den Geduldsfaktor von Personen festgelegt. Diese sind jeweils in der Quelle bzw. des Ziels der Personen konfiguriert und in der Szenariodatei auch abgelegt.

3 Ergebnisse

Nachfolgend wollen wir den implementierten Simulator im Zusammenhang mit dem Command Line Interface und der grafischen Benutzeroberfläche vorstellen. Des Weiteren stellen wir die ausgewählten Testfälle zur Validierung vor, führen diese durch und interpretieren die Ergebnisse.

3.1 Grafische Benutzeroberfläche (Eder)

In unserem Simulator von Personenbewegungen sind zahlreiche Szenarien denkbar. Diese manuell zu erstellen ist vor allem für große Simulationswelten aufwendig. Daher haben wir eine grafische Benutzeroberfläche erstellt, welche die Konfiguration eines Simulationsszenarios erleichtert.

Startansicht Die Startansicht der Anwendung ist in Abbildung 12 zu sehen. Auf dieser werden Funktionen zum Laden und Erstellen von Szenarien angeboten.

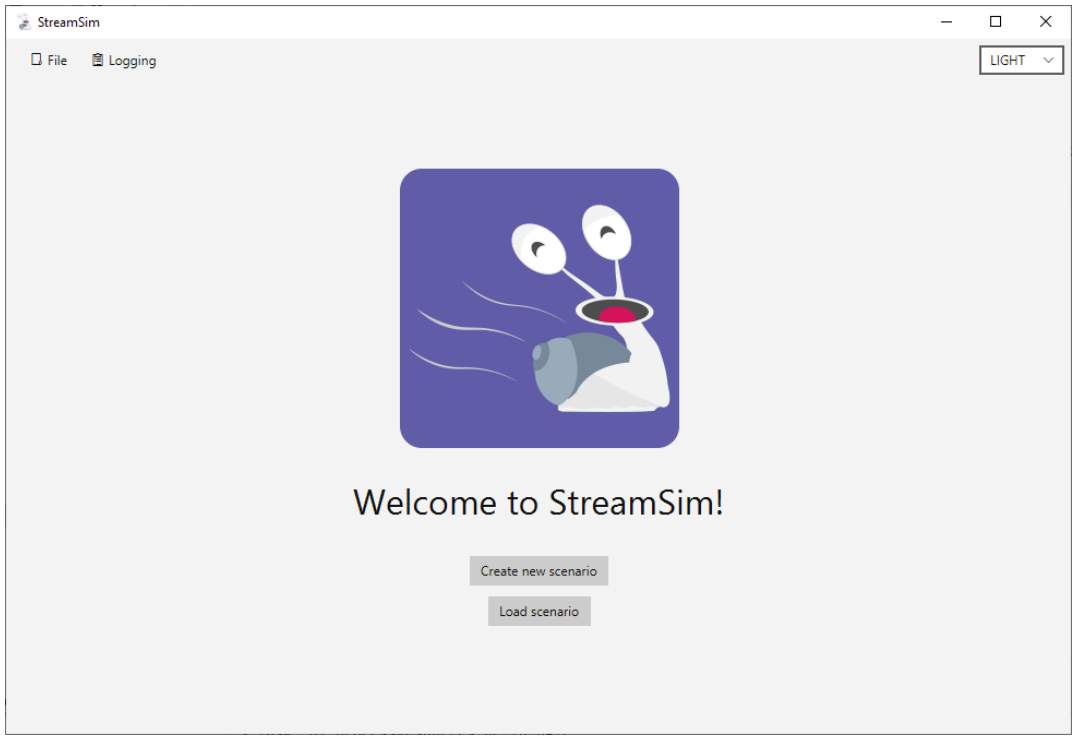


Abbildung 12. Startansicht der grafischen Benutzeroberfläche

Wir gehen erst später genauer auf das Laden von Szenarien ein, im Folgenden wollen wir zuerst die Neuerstellung präsentieren. Bei der Erstellung öffnet sich ein Dialog, wie er in Darstellung 13 zu sehen ist. Dieser erlaubt die Konfiguration der Höhe und Breite des Zellularautomaten.

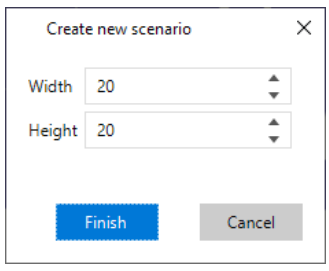


Abbildung 13. Dialog zum Erstellen eines neuen Szenarios

Hauptansicht Daraufhin öffnet sich die Hauptansicht des Programms, in welcher die Simulationswelt visualisiert wird. Hier gibt es wiederum drei verschiedene Modi: **Malen**, **Konfigurieren** und die **Simulation** an sich.

Malmodus Im **Malmodus** können wie in Screenshot 14 zu sehen ist die einzelnen „Bausteine“ der Simulationswelt auf dem Raster des Zellularautomaten gesetzt werden. Der Vorgang ähnelt Malen, woher der Modus seinen Namen hat.

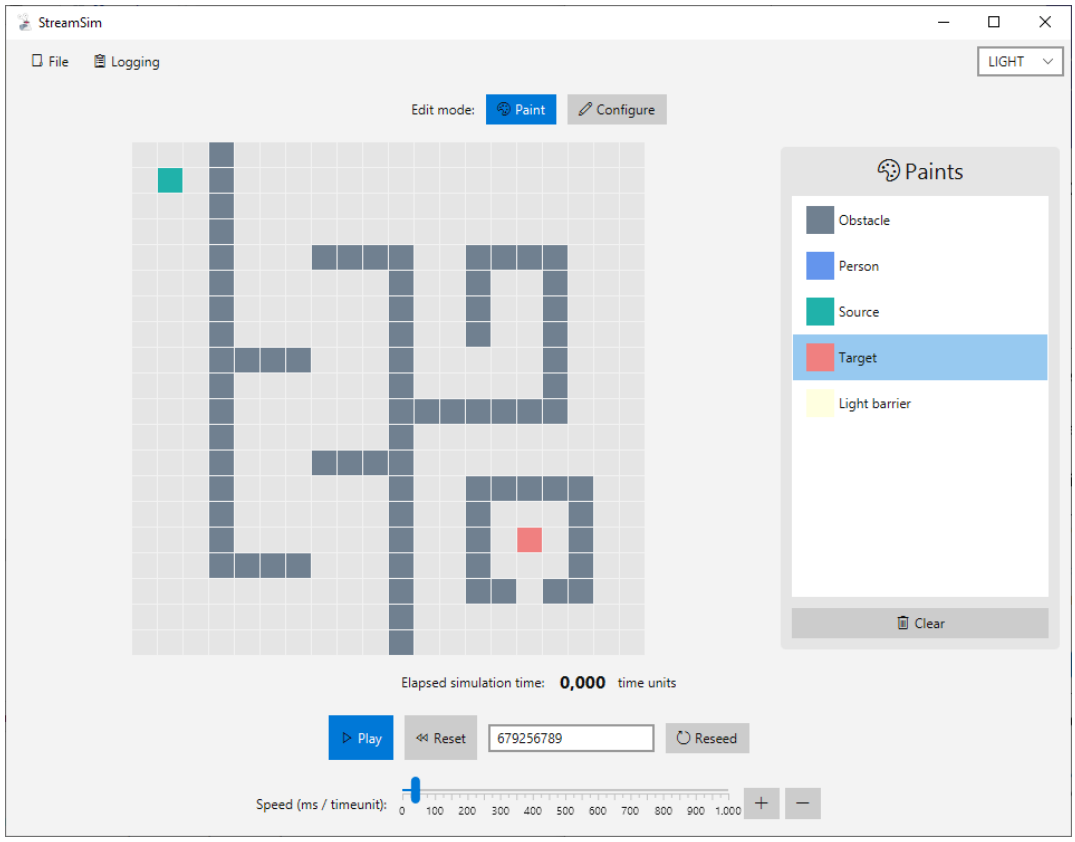


Abbildung 14. „Malen“ der Simulationswelt

Konfigurationsmodus Im Kontrast zum Malen Modus kümmert sich der **Konfigurationsmodus** um die Konfiguration der einzelnen Bausteine, welche zuvor „gemalt“ wurden. Jeder Baustein ist einzeln selektierbar, woraufhin im Abschnitt neben der Visualisierung der Simulationswelt die Konfigurationsmöglichkeiten erscheinen. Aktuell können lediglich Quellen und Ziele konfiguriert werden. Auf den Zielobjekten kann die sogenannte ConsumeStrategy festgelegt werden, also wie sich das Ziel verhält, wenn eine Person darauf läuft. Bei Quellobjekten sind alle anderen Möglichkeiten vorhanden, von der Häufigkeit der „Geburten“ neuer Personen neben der Quelle über die Definition der Verteilung der Personengeschwindigkeiten, bis zur Festlegung der zu verwendenden Bewegungsstrategie. In Abbildung 15 sind die Optionen für ein selektiertes Quellobjekt dargestellt.

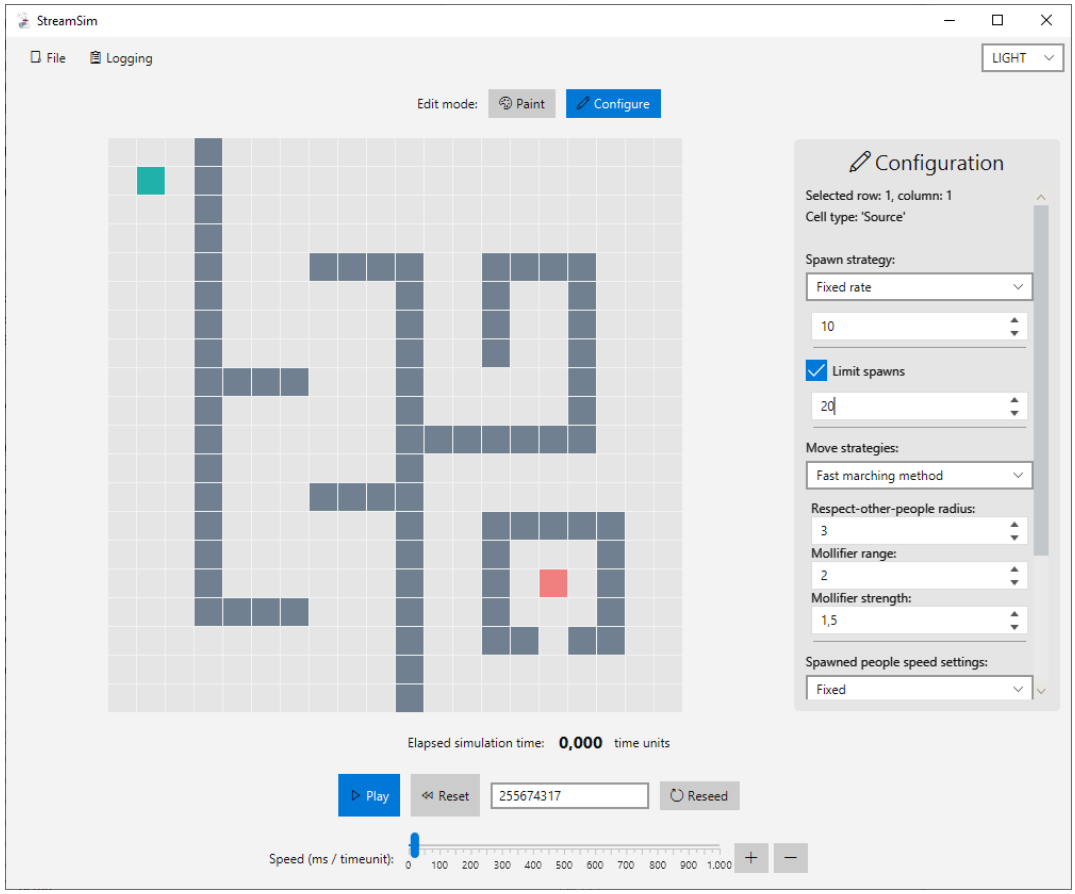


Abbildung 15. Konfigurationsmöglichkeiten werden für ein selektiertes Quellobjekt dargestellt

Simulationsmodus Der letzte Modus der Hauptansicht ist der **Simulationsmodus**. Dieser wird gestartet, wenn im unteren Teil der Ansicht auf den „Play“-Knopf gedrückt wird. Daraufhin startet die Simulation, welche die Visualisierung stetig aktualisiert. So kann der Verlauf live nachvollzogen werden.

Des Weiteren öffnen sich im rechten Abschnitt neben der Visualisierung verschiedene Statistiken, wie die aktuelle *Personenanzahl*, mittlere *Personengeschwindigkeit* und den *Fluss* in Abhängigkeit von der *Dichte*. Da unser Simulator intern nicht auf dem metrischen System aufbaut, sondern alles in „Zellbreiten“ misst, können wir die mittlere Geschwindigkeit in $\frac{m}{s}$, wie auch die Dichte in $\frac{1}{m^2}$ nicht direkt messen und brauchen eine Übersetzung von Zellbreiten in Meter - also wie viele Zellen in einen Meter passen. Dieser Wert kann in einem Eingabefeld unter den Diagrammen eingestellt werden. Geht man von einer Personenbreite von $40cm$ aus (Sarmady, Haron und Talib 2010, S. 1), passen in einen Meter $\frac{1}{0{,}4} = 2{,}5$ Zellen, was als Standardwert in der Applikation hinterlegt ist.

Neben der einstellbaren Zellbreite muss ebenso ein Fenster definiert werden, welches angibt wie viele der zuletzt gespeicherten Geschwindigkeiten von Personen zur Berechnung der mittleren Geschwindigkeit herangezogen werden. Als Standardwert ist hier 5 definiert, was bedeutet, dass nur die letzten 5 aufgenommenen Geschwindigkeiten einer Person in die Berechnung mit einfließen.

Außerdem kann die Simulation jederzeit pausiert werden. Ein Nebeneffekt der Pausierung ist, dass die aktuelle Potenzial- bzw. Nutzenfunktion als Heatmap direkt in der Visualisierung der Simulationswelt angezeigt wird. Ein Screenshot des pausierten Simulationsmodus ist in Abbildung 16 zu finden.

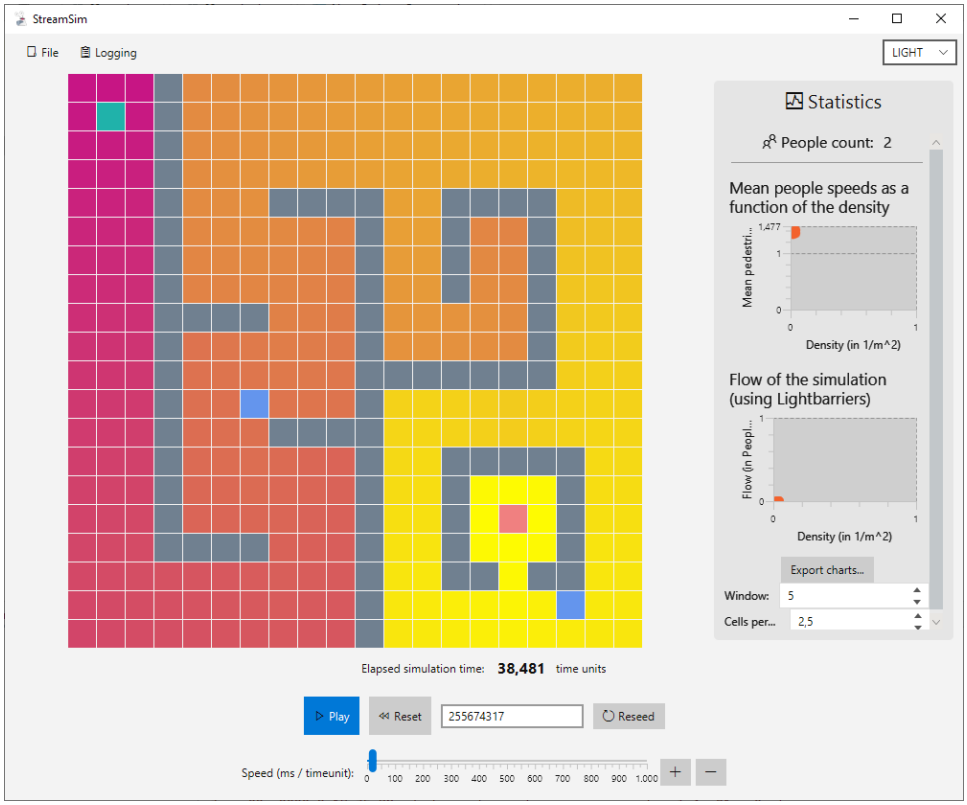


Abbildung 16. Screenshot des pausierten Simulationsmodus mit der Fast Marching Method als Bewegungsstrategie (siehe Heatmap)

Logging Das Logging der Personenbewegungen sowie der berechneten Statistiken wird über das Menü „Logging“ aktiviert. Ebenso können wir dort den Speicherort der Logdateien definieren.

Speichern und Laden von Szenarien Die Simulation mit der grafischen Benutzeroberfläche ist durch die Visualisierung deutlich langsamer als über das Command Line Interface. Daher bietet es sich an, das soeben erstelle Szenario zu speichern und später über das CLI zu laden. Dazu bieten wir eine Speicher- und Ladefunktion an, welche wie auch das Logging über das Menü aufgerufen wird. Die Datei wird im JSON Format abgelegt. Ein Ausschnitt einer solchen Datei ist in Listing 1 zu sehen.

```
{
  ...
  "R1C1": {
    "typeID": 3,
    "location": {
      "row": 1,
      "column": 1
    },
    "configuration": {
      "@type": "Source",
      "spawnStrategy": {
        "@type": "Fixed rate",
        "fixedRate": 10
      },
      "moveStrategy": {
        "@type": "Fast marching method",
        "mollifierConfiguration": {
          "range": 2,
          "strength": 1.5
        },
        "radius": 3
      },
      "maxSpawns": 20,
      "speedGenerator": {
        "@type": "Fixed",
        "speed": 3.34,
        "name": "Fixed"
      },
      "patienceGenerator": {
        "@type": "Norm",
        "mean": 20,
        "standardDeviation": 10
      }
    }
  },
  "R13C13": {
    "typeID": 2,
    "location": {
      "row": 13,
      "column": 13
    }
  },
  ...
}
```

Listing 1. Auszug aus einer gespeicherten Szenariodatei im JSON Forrmat

3.2 Command Line Interface (Stiglmeier)

Das Command Line Interface (CLI) dient der Ausführung von Simulationen ohne eine Darstellung des rechen-aufwändigen User Interfaces. So können z.B. wie im vorherigen Kapitel erwähnt Konfigurationen mit dem User Interface erstellt und gespeichert werden, um darauf eine Simulation mit der Konfiguration zu starten. Der Befehl zum Starten des CLI aus der Entwicklungsumgebung lautet:

linux: *gradlew :cli:run –args=«PATH TO CONFIGURATION>"*
windows: *gradlew.bat :cli:run –args=«PATH TO CONFIGURATION>"*

Dabei erwartet das CLI als erstes und einziges Argument den Pfad zur Konfigurationsdatei. Zusätzlich können dem Befehl noch weitere Optionen mitgegeben werden. Eine vollständige Übersicht findet sich unter Tabelle 1.

Während der Großteil der Optionen dazu dient, die in der Konfiguration angegebenen Parameter zu überschreiben, gibt es vier besondere Optionen, denn diese Funktionalitäten sind ausschließlich im CLI vorhanden.

Die Option *-r* dient der wiederholten Ausführung der Simulation. Damit können also Batch-Jobs realisiert werden.

Die Optionen *- -max-simulation-time*, *- -max-density* und *- -log-simulation-time-change-delay* dienen der Erstellung so genannter Early-Exit Strategien. Wenn eine Bedingung der genannten Optionen zutrifft, dann wird die Simulation an diesem Punkt beendet.

Name	Description
-s, - -seed	The seed to use instead of the one in the configuration
-as, - -auto-seed	Automatically choose a random seed other than the seed in the configuration
-r, - -runs	Specify the number of times to run the simulation
-d, - -delay	Delay the simulation: Specify how long a time unit should take (in milliseconds)
-l, - -log	Whether writing the simulation logs to the file system is enabled
-lf, - -log-folder	Folder to write simulation logs to
-ln, - -log-file-name	File name prefix for simulation logs written to the file system
- -statistics-logging-debounce-delay	Time units to wait before logging statistics again
- -cells-per-meter	How many cells fit in a meter. Used to calculate density. Value of 2.5 means a person needs 40cm.
- -mean-speed-window-size	Size of the window to use to calculate the mean speed of people
- -max-simulation-time	The maximum simulation time to early exit the simulation
- -max-density	The maximum density to early exit the simulation
- -max-people	The maximum amount of people to early exit the simulation
- -log-simulation-time-change-delay	The delay specifies how much time is waited until logging the current simulation time again

Tabelle 1. Übersicht der möglichen Optionen beim Aufruf des CLI

3.3 Verifikation (Stiglmeier)

Zur Überprüfung der Korrektheit des Simulators wurden Unit-Tests erstellt. Dabei wird eine Code-Abdeckung von 80 % erreicht. Nach Cornett 2013 stellt dies einen ausreichenden Grad an Abdeckung dar, da ein höherer Wert meist ressourcen-ineffizient ist. Die Menge an Tests lässt sich nach Zweck in zwei Kategorien einteilen. Tests zur Validierung des Modells und Tests zur Verifikation der Software. Die Validierung wird im nachfolgenden Kapitel 3.4 behandelt. Um den erstellten Code zu verifizieren, wurden die folgenden Bereiche getestet:

Scheduler

Nachdem der Simulator einer Event-driven Architektur zugrunde liegt (Vgl. Kapitel 2.3), ist es wichtig den Scheduler für die Abarbeitung der Events zu testen. Dabei wurde Wert gelegt, den Normalfall sowie Sonderverhalten wie das falsche Einreihen von Events in der Vergangenheit oder dem Sicherstellen des deterministischen Verhalten zu überprüfen.

Logging

Eine Funktionalität des Simulators ist das Loggen der Events in Dateien, welches besonders für Batch-Jobs relevant ist. Um sicherzustellen, dass dies korrekt funktioniert, wurde auch dies überprüft.

Objektverhalten

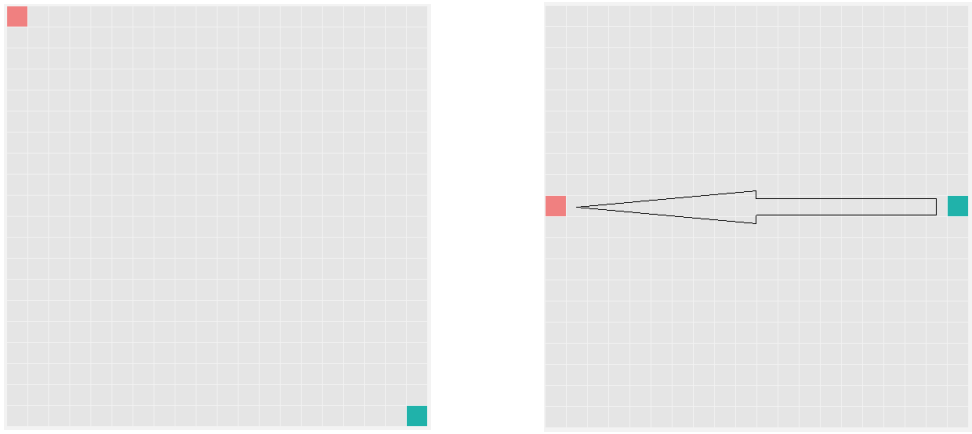
Eine weitere Kernfunktionalität ist das Verhalten der einzelnen Objekte auf dem Feld. Bei Zielen wurde überprüft, ob Personen beim Erreichen ihres Ziels richtig entfernt oder wieder auf der anderen Seite eingereiht werden. Beim Erstellen von Personen wurde überprüft, ob diese zeitlich korrekt erstellt werden, also z.B. kontinuierlich oder gemäßt einer Verteilung. Beim Bewegen der Personen wurde ebenfalls überprüft, ob die drei implementierten Bewegungsmuster (Euklidisch, Fast-Marching und Dijkstra) die erwartete Potentialmatrix für das Feld korrekt berechnen. Die Potentialmatrix ist elementar für die richtige Bewegung.

3.4 Validierung

Zur Validerung des Simulators wird eine Reihe von Tests ausgewählt, welche vom RiMEA-Verein (Vgl. “RiMEA e.V.” 2020) als standardisierte Testfälle für Personenstromsimulatoren entworfen worden sind. Im Folgenden wird dabei für die einzelnen Testfälle jeweils der Aufbau sowie die Resultate des implementierten Simulators dargestellt.

Freier Fluss (Stiglmeier)

Beim Test des freien Flusses geht es darum zu überprüfen, ob die Personen, beim freien Bewegen ohne Hindernisse, direkt zum Ziel laufen und dabei ihre Wunschgeschwindigkeit einhalten. Hierfür wurden 4 Testfälle für jede Bewegungsstrategie auswählt (Vgl. Abbildung 17). Eine Person geht dabei diagonal, horizontal, vertikal und halb-diagonal vom Startpunkt zum Ziel. Für den Testaufbau sind zwei Felder mit der Größe 50x50 und 10x10 Zellen sowie einer Breite von 40cm pro Zelle aufgebaut. Die Personen laufen dabei mit einer Geschwindigkeit von 3.34 Zellen pro Zeiteinheit.



(a) Aufbau bei diagonaler Bewegung

(b) Aufbau bei horizontaler Bewegung

(c) Aufbau bei vertikaler Bewegung

(d) Aufbau mit halb-diagonaler Bewegung

Abbildung 17. Übericht der einzelnen Testfälle im Test zum freien Fluss (grün: Personengenerator, rot: Ziel)

Die Ergebnisse der Durchführung des Tests sind in den Tabellen 2, 3 und 4 zu sehen.

Das Verhalten der euklidischen (Vgl. Tabelle 2) und der Fast-Marching (Vgl. Tabelle 3) Bewegungs-Strategie ist beim freien Fluss in dieser Implementierung identisch. Bei kleiner Feldgröße lässt die Genauigkeit nach, da die Bewegung der Personen von der geraden Bewegung auf die Zellen normiert werden muss und dabei die Geschwindigkeit am Ende einen kleinen Fehler aufweist. Bei steigender Feldgröße wird dieser Fehler dann relativ zur Größe kleiner, weshalb bei der Feldgröße von 50x50 auch die Abweichung um etwa den gleichen Faktor von 5 kleiner wird. In der halb-diagonalen Richtung ist die Abweichung jedoch, anders als bei den anderen Richtungen, bei steigender Feldgröße gestiegen. Das liegt daran, dass die Bewegungs-Strategien versuchen, die Person der geraden Linie entlang vom Start bis zum Ziel entlang zu bewegen. Da dies durch die Zelleinteilung nicht immer optimal möglich ist, muss die Person gelegentlich korrigierende Schritte vollziehen. Dies ist exemplarisch in Abbildung 18 zu sehen. Bei großer Feldgröße sind das mehr Schritte als bei kleiner Feldgröße, weshalb die Abweichung steigt. Das heißt also, dass sich die Personen mit Wunschgeschwindigkeit bewegen und die Abweichung lediglich durch das Zellgitter entsteht.

Die Dijkstra Bewegungs-Strategie (Vgl. Tabelle 4) weist bei horizontaler Bewegung eine nahezu perfekte Genauigkeit auf und ist zusätzlich in der halb-diagonalen Bewegung, anders als bei den anderen Bewegungen, bei steigender Feldgröße genauer geworden.

Feldgröße	horizontal	vertikal	diagonal	halb-diagonal
10x10	4.60 %	11.11 %	6.50 %	1.94 %
50x50	0.84 %	2.04 %	1.19 %	6.19 %

Tabelle 2. Übersicht der Abweichungen des Tests im freien Fluss mit euklidischer Bewegung

Feldgröße	horizontal	vertikal	diagonal	halb-diagonal
10x10	4.60 %	11.11 %	6.50 %	1.94 %
50x50	0.84 %	2.04 %	1.19 %	6.19 %

Tabelle 3. Übersicht der Abweichungen des Tests im freien Fluss mit Fast-Marching Bewegung

Feldgröße	horizontal	vertikal	diagonal	halb-diagonal
10x10	1.15E-13 %	11.11 %	3.25 %	6.15 %
50x50	1.21E-13 %	2.04 %	0.59 %	5.43 %

Tabelle 4. Übersicht der Abweichungen des Tests im freien Fluss mit Dijkstra Bewegung

Bei Betrachtung dieser Werte scheint die Implementierung der Dijkstra-Bewegungstrategie am Gelungensten. Leider ist der Dijkstra Algorithmus sehr rechenintensiv und verursacht schon bei einer Feldgröße von 150x150 einen Stackoverflow.

Im Gesamten ist eine Abweichung bei der Wunschgeschwindigkeit von wenigen Prozent bei einer Normierung der Bewegung auf feste Zellen ein ausreichend genaues Ergebnis und damit validiert.

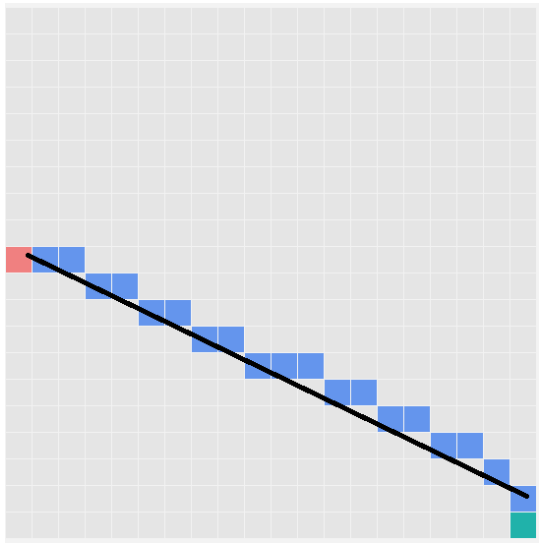


Abbildung 18. Bewegung einer Person mit euklidischer Strategie in halb-diagonaler Richtung (grün: Personengenerator, rot: Ziel, blau: Person)

Hühnertest (Schlosser)

Der Hühnertest ist eine Methode zum Überprüfen, ob Bewegungsalgorithmen mit Hindernissen umgehen, die zwischen Ziel und Quelle liegen und das Potential haben, Personen einzusperren, sofern sie versuchen, nur auf kürzestem Weg ins Ziel zu gelangen.

Dazu wurde ein Aufbau erstellt, in dem ein *U*-förmiges Hindernis den kürzesten Weg von Quelle zu Senke blockiert. Ein Testaufbau ist in Abbildung 19 dargestellt.

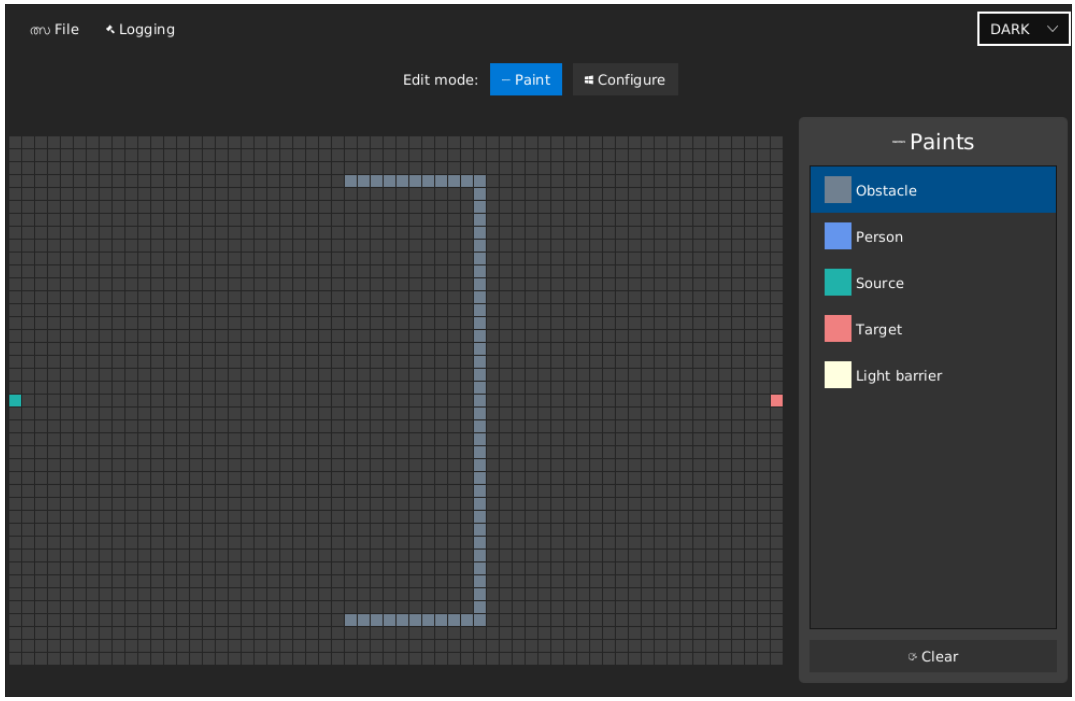


Abbildung 19. Szenario des Hühnertest mit dem Hindernis im Weg von der Quelle (grün) zum Ziel (rot)

Wir erwarten, dass Algorithmen, die Hindernisse und Wände in die Berechnung der Wege einbeziehen, mit dieser Situation umgehen können. Für einfache Algorithmen hingegen, die nur darauf abzielen, den Abstand zum Ziel zu minimieren, würden wir erwarten, dass die erstellten Personen in dem Hindernis ”hängen”bleiben.

Dieser Test wird automatisiert mit allen unseren Bewegungsalgorithmen (Euklidische Distanz, Djikstra, Fast Marching) durchgeführt.

Für die Euklidische Bewegungsmethode, welche nur die Distanz von Personen zum Ziel minimiert gilt der Test als erfolgreich, wenn keine Person das Ziel betritt. Diese zielt, wie oben angesprochen, nur darauf ab, den Abstand zu minimieren und kann daher mit dieser Situation nicht korrekt umgehen.

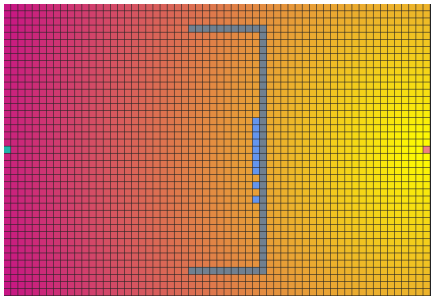
Für die beiden anderen hingegen gilt der Test nur dann als erfolgreich, wenn alle erstellten Personen das Ziel auch erreichen.

Diese Tests laufen im Rahmen unserer Testfälle bei Änderungen im Code und informieren uns so, ob die beiden komplexeren Algorithmen noch korrekt funktionieren und nicht unerwartet abgeändert wurden. Diese Tests sind bei unseren Algorithmen daher erfolgreich.

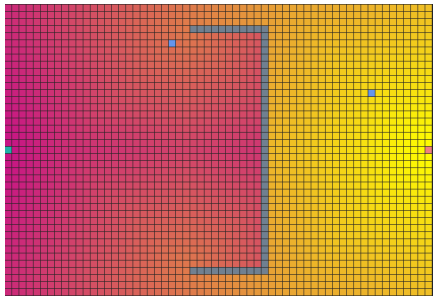
In Bild 20 folgt eine Darstellung der einzelnen Testszenarien. Gut zu sehen ist der Abfall des Potentials im Hindernis beim Djikstraalgorithmus und der Fast-Marching Methode.

Dieser wird durch die rötere Färbung der Felder ersichtlich.

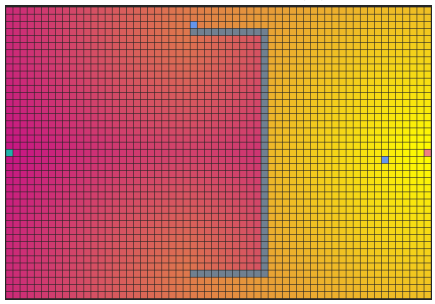
Beim Euklidischen Algorithmus ist dieser Abfall nicht zu beobachten, da dort die Hindernisse nicht mit in die Berechnung der Bewegungen aufgenommen werden.



(a) Hühnertest mit der Euklidischen Bewegungsmethode



(b) Hühnertest mit dem Djikstraalgorithmus



(c) Hühnertest mit der Fast-Marching Methode

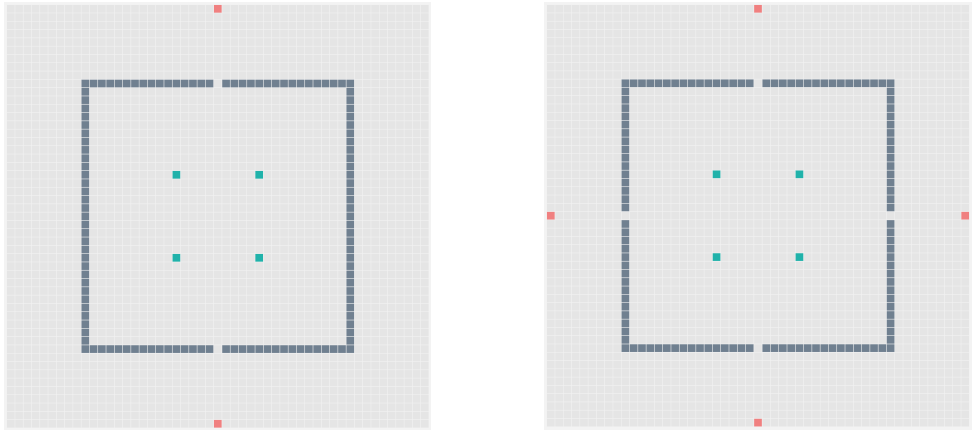
Abbildung 20. Übericht der Testfälle für den Hühnertest (grün: Personengenerator, rot: Ziel)

Evakuierung eines Raums **(Beil Benedikt)**

Bei diesem Test wird das Evakuierungsverhalten überprüft. Dazu werden zwei fast identische Räume erstellt. Ein Raum besitzt zwei Türen. Der andere Raum besitzt 4 Türen. Für jede Tür gibt es ein Ziel. Unser Simulationsmodell bietet nicht die Möglichkeit eine vollen Raum zu erschaffen. Dazu wäre zum Beispiel eine Schrank nötigt, die sich öffnet, sobal der Raum gefüllt ist. Alternativ wäre ein Simulationsmodell erforderlich, welches die Möglichkeit bietet, aktive Personen auf den Feld zu platzieren, bovor die Simulation beginnt. Da keiner dieser Funktionen in unseren Simulationsmodell vorhanden ist, versuchen wir einen vollen Raum zu erschaffen, indem wir mit mehreren Quellen möglichst viele Personen erzeugen. Dazu verwenden wir folgende Quellenkonfiguration:

Spawnstrategie	Fester Abstand von 0,1 Zeiteinheiten. Die Quelle erzeugt jede 0,1 Zeiteinheit eine Persone, fals eine angrenzende Zelle unbelegt ist. Dieser Wert wurde bewusst so gering gewählt, um den Raum mit Personen zu fluten. Jede Quelle erzeugt insgesammt 250 Personen.
Bewegungsstrategie	Euclidische Distanz mit der Basiskonfiguration für die Personenabstoßung. Die Euclidische Distanz wurde gewählt um die Personen bestmöglichst im Raum zu streuen. Bei den alternativen Bewegunngsstrategien, würde die Personen sich nie in die Raumecken bewegen, da die Personen wissen, dass ein Schritt in die Ecke sie nicht näher zu ihren Ziel bringt. Bei einen gefüllten Raum sind die Ecken aber besetzt.

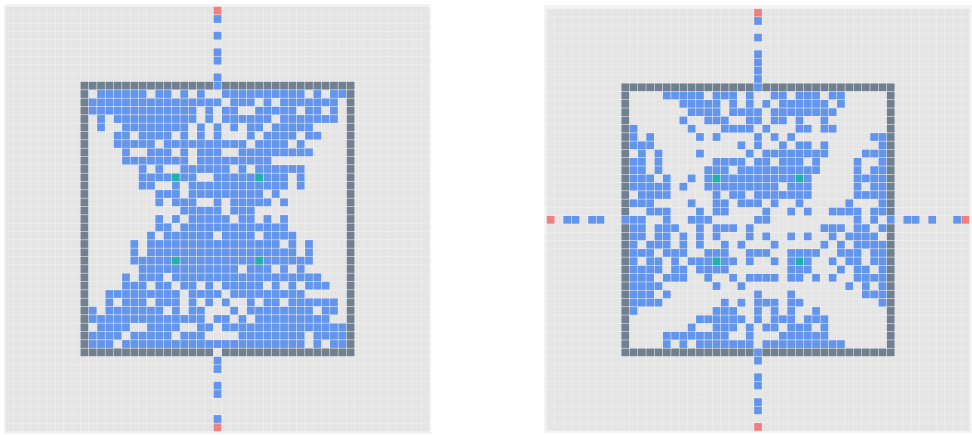
Beide Modelle verwenden eine Seed von 985085411. Für die restlichen Konfigurationen wurde die Basiskonfiguration verwendet. Der Startzustand der Zellularautomaten kann der Abbildunge 21 entnommen werden.



(a) Startzustand des Modells mit zwei Türen (b) Startzustand des Modells mit 4 Türen

Abbildung 21. Startzustand der Modelle, welche in diesem Test verglichen werden

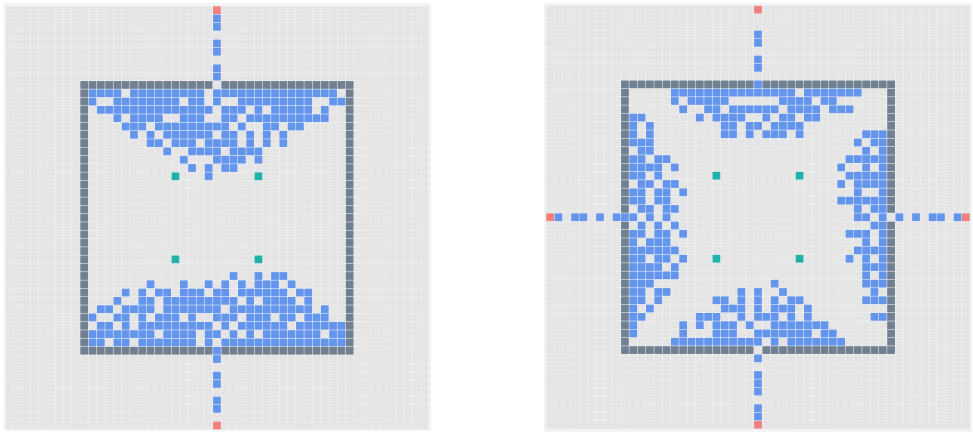
Der Raum besitzt $31x31 = 961$ Zellen, wovon 4 Zellen immer durch die 4 Quellen belegt sind. Es wurden mehrer Quellen verwendet, um schneller den Raum zu füllen. Der Raum wird schneller gefüllt als bei einer Quelle, da 4 Quellen mehr benachbarte Zellen zum Erzeugen von Personen zur verfügung steht. Diese wurden auch bewusst mit Abstand platziert, um das gegenseitige Verstopfen zu verhindern. Jede Quelle erzeugt 250 Personen wodurch insgesamt 1000 Personen erzeugt werden. Dieser Wert wurde bewusst großer wie die Zellenanzahl des Raumes gewählt, da Personen abhanden gehen, bis der Raum gefüllt ist. Der Zellularautomat im gefüllten Zustand ist in der Abbildung 22 zu entnehmen. Der Raum mit zwei Türen ist dabei besser gefüllt, da der Personenabfluss im vergleich zu den Raum mit vier Türen geringer ist. Dadurch staut es die Personen im Raum mit zwei Türen stärker an.



(a) Gefüllter Zustand des Modells mit zwei Türen (b) Gefüllter Zustand des Modells mit 4 Türen

Abbildung 22. Gefüllter Zustand der Modelle, welche in diesem Test verglichen werden

Zum Ende der Simulationen ergibt sich eine pilzartiges Muster vor den Türen. Da nun keine neuen Personen erzeugt werden, befinden sich in der Mitte keine Personen mehr. Da die Euclidische Distanz als Bewegungsgstrategie verwendet wird, laufen alle Personen in die Richtung ihrer jeweiligen Tür und verteilen sich darum. Dieses Verhalten ist Abbildung 23 zu sehen.



(a) Zustand des Modells mit zwei Türen nahe dem Ende der Simulation

(b) Zustand des Modells mit 4 Türen nahe dem Ende der Simulation

Abbildung 23. Zustand der Modelle, welche in diesem Test verglichen werden, nahe dem Ende der Simulation

Ergebnis des Vergleiches In der Simulation mit 4 Türen wurden die Personen mit dem **Faktor 1,82** schnell evakuiert im Vergleich zu der Simulation mit 1 Türen. Erwartet wurde ein Wert von 2 bzw. eine Wert nahe 2, da die doppelte Anzahl an Türen einen doppelten Personenfluss ermöglichen. Ein Wert von 2 bzw. ein Wert von sehr nahe an 2 kann durch dieses Simulationmodell nicht erreicht werden, da die Simulation nicht mit einen gefüllten Raum beginnen kann. Dieser muss zuerst durch die Quellen gefüllt werden.

	Simulationsdauer [Zeiteinheiten]
2 Türen	255,716
4 Türen	140,513

Fundamentaldiagramm (Eder)
Bei diesem Test lassen wir virtuelle Personen einen Gang entlanglaufen. Dieser soll 65 Meter lang sowie 12 Meter breit sein. Gehen wir wie in Sarmady, Haron und Talib (2010, S. 1) erwähnt von einer Zellbreite b von $40cm = 0,4m$ aus, muss die Simulationswelt $\frac{65m}{0,4m} \approx 163$ Zellen lang und $\frac{12m}{0,4m} = 30$ Zellen breit sein.

Zur Variation der Dichte mit fortschreitender Simulationszeit speisen wir Personen, welche am rechten Ende des Ganges ankommen sofort wieder auf der linken Quellseite ein. Zusätzlich erzeugen Quellen zur linken Seite Poisson-verteilt neue Personen. Die Personenanzahl steigt so stetig.

Das **Fundamentaldiagramm** gibt die mittlere Geschwindigkeit aller Personen (in $\frac{m}{s}$) in Abhängigkeit von der Dichte (in Personen pro Quadratmeter $\frac{1}{m^2}$) wieder und dient uns als Vergleichsmerkmal zur Validierung mit Beobachtungen aus der realen Welt. Wir verwenden Diagramme von Weidmann ("Weidmann - Fundamentaldiagramm" 2020) und Hankin & Wright ("Hankin and Wright - Fundamentaldiagramm" 2020) zum Vergleichen. Das erlaubt uns eine **Kalibrierung** der einstellbaren Parameter des Simulators durch **Ausprobieren**. In Abbildung 24 sind die beiden erwähnten Referenz-Fundamentaldiagramme zu sehen.

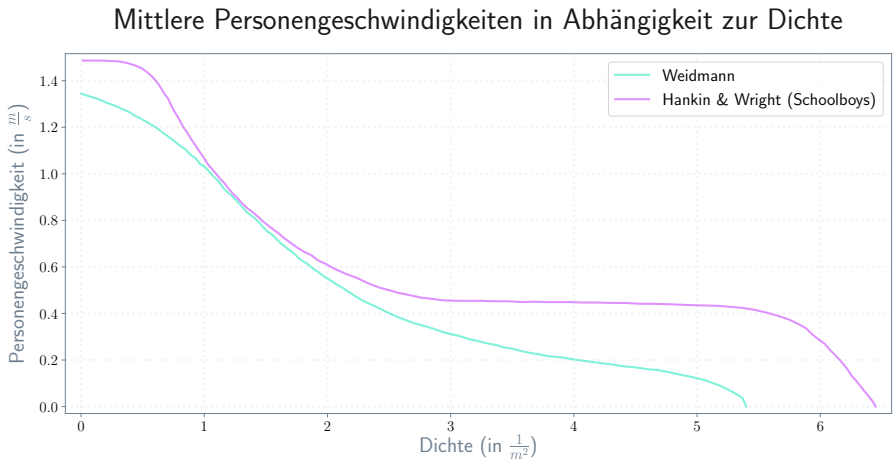


Abbildung 24. Die Referenz-Fundamentaldiagramme von Weidmann und Hankin & Wright

Mittlere Geschwindigkeit Die mittlere Geschwindigkeit $\overline{s}$ wird berechnet, indem wir die w letzten Geschwindigkeitsaufzeichnungen eines jeden Individuums mitteln. Dabei berechnet sich eine Geschwindigkeitsaufzeichnung $s(p, x)$ einer Person p zur Bewegung x mit Hilfe der zurückgelegten Distanz d (Diagonal $\sqrt{2}$, sonst 1) sowie der vergangenen Zeit zur letzten Bewegung Δt_b.

$$s(p, x) = \frac{d}{\Delta t_b} \frac{Zellen}{s}$$

Bei n Personen folgt die Bildung des Durchschnitts mit der folgenden Formel, wobei $l(p)$ die Anzahl an verfügbaren Geschwindigkeitsaufzeichnungen für die Person p darstellt:

$$\overline{s} = \frac{b}{n} \sum_{k=0}^{n-1} \frac{1}{min(w, l(k))} \sum_{i=0}^{min(w,l(k))-1} s(k, l(k) - i)$$

Die Normalisierung von $\frac{Zellen}{s}$ auf $\frac{m}{s}$ erfolgt über die Zellbreite b.

Dichte Im Gegensatz dazu berechnet sich die Dichte ρ global über die Größe der Simulationswelt. Genauer über die Anzahl der betretbaren Zellen c_w, die Zellbreite b und der aktuellen Anzahl an Personen c_p.

$$\rho = \frac{c_p}{c_w * b^2}$$

Beispielsweise in einer Welt, wie dem vorgeschlagenen Gang mit $163 * 30 = 4890$ Zellen, davon $c_w = 161 * 30 = 4830$ betretbaren (60 Zellen werden von nicht-betretbaren Quell- und Zielobjekten belegt). Wenn $c_p = 1500$ beträgt ergibt sich eine Dichte von $\rho = \frac{1500}{4830 * 0{,}4^2} \approx 1,94$.

Fluss Zusätzlich messen wir den Personenfluss τ (in $\frac{Personen}{m*s}$) durch den Gang mit Hilfe einer virtuellen Lichtschranke. Diese ist in Abbildung 25 in den gelben Zellen dargestellt. Ihre Funktionsweise ist einfach: Wann immer eine Person diese Zelle betritt, wird ein Zähler z hochgezählt, welcher bei jedem Logeintrag in die Statistiklogdatei ausgelesen und zurückgesetzt wird. Dabei spielt die Zeitdifferenz t_d zwischen der aktuellen Simulationszeit und der Zeit der letzten Auswertung sowie die Anzahl an Lichtschrankenzellen c_L eine Rolle (Zur Normalisierung und damit Vergleichbarkeit).

$$\tau = \frac{z}{t_d * \frac{c_L}{b}}$$

Probleme Bei den ersten Tests ist schnell aufgefallen, dass der Gang irgendwann nicht mehr voller wird. Der Grund lag darin, dass unsere Quellen nach einem erfolglosen Spawnversuch einer Person erst wieder nach einer Poisson-verteilt gezogenen Zeit, einen erneuten Versuch starten. Wenn die Dichte schon sehr hoch ist, ist die Wahrscheinlichkeit gering, dass wieder ein Platz neben der Quelle zum „spawnen“ frei ist. Die Lösung besteht darin, dass die Quelle nach einem erfolglosen Versuch direkt nach dem nächsten Ereignis in unserem Event-Driven-Scheduler es erneut versucht. Dadurch ist garantiert, dass sobald ein Platz neben der Quelle frei wird, dieser sofort von der neuen Person besetzt wird.

Szenario Das entstehende Szenario besteht, wie bereits erwähnt aus einer $163x30$ Zellen großen Welt. In der ersten Spalte befinden sich Quellen, in der letzten die Ziele. In der Mitte sind die Lichtschranken zur Messung des Flusses zu finden. Eine Darstellung ist in Abbildung 25 zu finden.

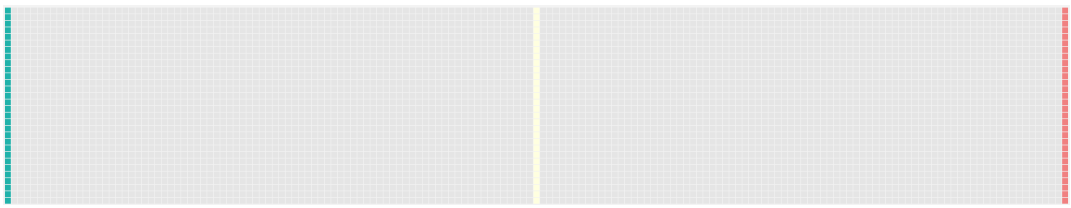


Abbildung 25. Screenshot des Simulationsszenarios für den Test. Ein Gang mit 163 Zellen Länge und 30 Zellen Breite.

Für die folgenden Tests wollen wir die exakte Konfiguration auflisten.

Allgemeine Konfiguration

Zeilen	30
Spalten	163
Seed	331810044 (Willkürlich gewählt, dient der Nachvollziehbarkeit)
Mittelungsfenster	Das Fenster zur Mittelung der Geschwindigkeit wird auf 5 gesetzt.
Zellen in einem Meter	2,5 Zellen sollen in einen Meter passen. Anders gesagt eine Zelle ist $40cm$ breit.

Quellenkonfiguration

Spawnstrategie	Poisson-verteilt mit $\lambda = 10$. Quelle versucht also im Mittel alle 10 Sekunden eine neue Person zu „spawnen“. Dieser Wert wurde gewählt, um die Menschenanzahl im Gang nicht schlagartig zu erhöhen, sondern langsam, jedoch schnell genug, dass die Simulation nicht ewig dauert.
Bewegungsstrategie	Fast marching method, wobei der Glättungskern die Reichweite 3 und Stärke $1,5$ erhält. Die Abstoßung von Personen wird noch in einem Radius von 4 betrachtet, damit die Reichweite des Glättungskerns ausgenutzt wird. In Hall (1966, S. 126) wird die Distanz des „personal space“ mit 4 Fuß $\approx 1,22m$ angegeben. Bei einer Zellbreite von $40cm$ ergibt sich die Reichweite 3. Die Stärke ist willkürlich festgelegt, scheint jedoch in kleineren Tests bei Vergleichen mit den Referenzfundamentaldiagrammen zu guten Ergebnissen zu führen.
Wunschgeschwindigkeiten	Über eine Normalverteilung mit $\mu = 3,34\frac{Zellen}{s}$ (Entspricht $1,34\frac{m}{s}$ bei einer Zellbreite von $0,4m$) und $\sigma = 0,65\frac{Zellen}{s}$ (Entspricht $0,26\frac{m}{s}$).
Geduld	Über eine Normalverteilung mit $\mu = 10$ und $maxDeviation = 5$. Die Werte wurden als Kompromiss zwischen zur starker und zu schwacher Bewegungsbereitschaft im Falle einer verschlechternden Bewegung gewählt.

Zielkonfiguration Des Weiteren erhalten alle 30 Zielobjekte die Strategie zum Teleportieren der Personen an eine zufällige Quelle bei Ankunft.

Basisversion Wir wollen die Simulation in der genannten Konfiguration, der „Basisversion“ durchführen und das Fundamentaldiagramm sowie den Fluss visualisieren. Das Ergebnis ist in Darstellung 26 und 27 abgebildet. Die Simulation ist $\approx 2,3h$ in Simulationszeit gelaufen.

Wenn man das Fundamentaldiagramm betrachtet fällt auf, dass die „kalibrierte“ Basisversion (Sicher noch zu verbessern) qualitativ zu den beiden Referenzkurven passt, jedoch eher zu der von Hankin & Wright. Ebenso beobachten wir einen schlagartigen Abfall der mittleren Geschwindigkeit um die Dichte 1. Möglicherweise handelt es sich hier um ein Problem, welches durch die Verwendung des Zellularautomaten oder zu ähnlichem Verhalten aller Personen entsteht. Des Weiteren sind zu Beginn der Kurve wenige Datenpunkte vorhanden, weil die Personenanzahl am Anfang deutlich schneller steigt, als im späteren Verlauf der Simulation. Grund hierfür sind die Quellen, welche bei höheren Dichten oftmals keine Möglichkeit finden (keine freien Nachbarzellen), auf denen eine neue Person „gespawned“ werden kann. Die Dichte bleibt also längere Zeit gleich, wodurch mehr Datenpunkte auf höheren Dichten entstehen.

Bei dem Flussdiagramm können wir beobachten, dass der Fluss zu Beginn mit zunehmender Dichte stark ansteigt. Ab einer Dichte von ca. 1 fangen die Personen an sich durch Abstoßung zu beeinflussen, wodurch der Fluss sinkt. Je dichter die Welt mit Personen besetzt ist, desto mehr Personen können allerdings trotz Abstoßung sich bewegen, was bei fast konstanter mittleren Geschwindigkeit (Siehe Fundamentaldiagramm) zu einer Flusserhöhung führt. Bei sehr hohen Dichten > 6 ist die Welt sehr dicht besetzt, sodass immer mehr Personen keine Möglichkeit mehr haben sich zu bewegen, wodurch die mittlere Geschwindigkeit langsam gegen 0 konvergiert.

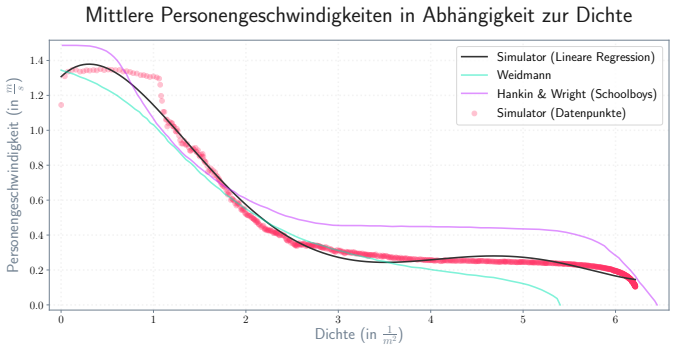


Abbildung 26. Fundamentaldiagramm zur Basisversion der Konfiguration zum Fundamentaldiagramm Test.

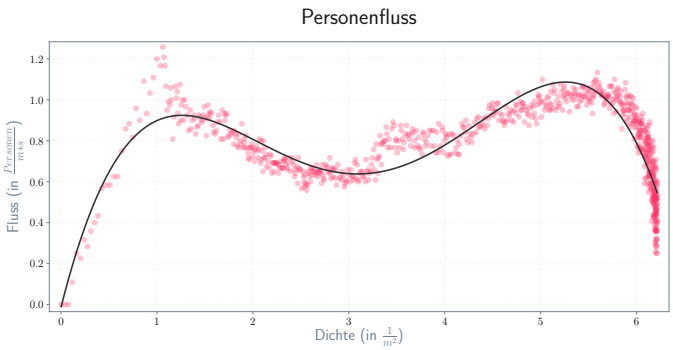


Abbildung 27. Flussdiagramm zur Basisversion der Konfiguration zum Fundamentaldiagramm Test.

Fundamentaldiagramm beeinflussen Es stellt sich die Frage, wie unsere Parameter das Fundamentaldiagramm beeinflussen. Dazu führen wir mehrere Simulationen aus, wobei wir immer ein bisschen von der Basisversion abweichen. Insbesondere sind wir an den Auswirkungen der Geduld, der Glättungskern Reichweite und Stärke sowie dem Beachtungsradius der Abstoßung interessiert.

Angefangen mit Abbildung 28 sehen wir die Auswirkungen von sehr hoher Geduld (999999999) und keiner Geduld (0). Wenn wir keine Ungeduld implementiert hätten, wäre die grün-gelbliche Kurve unser Ergebnis. Die Personen warten nicht, sondern bewegen sich um jeden Preis. Dadurch sinkt die mittlere Geschwindigkeit wenig. Im Gegensatz dazu fangen die Personen bei der mintgrünen Kurve mit sehr viel Geduld bei leichter Beeinflussung durch andere Personen schon an unbegrenzte Zeit zu warten. Eine Art Stau entsteht, die Personen bewegen sich also erst, wenn der Wunschweg vor ihnen freigegeben wird. Hohe Dichten werden gar nicht erst erreicht, da sehr früh ein Zeitpunkt erreicht wird, an dem keine Person mehr einen verbessernden Weg findet und dadurch die Simulation einfriert.

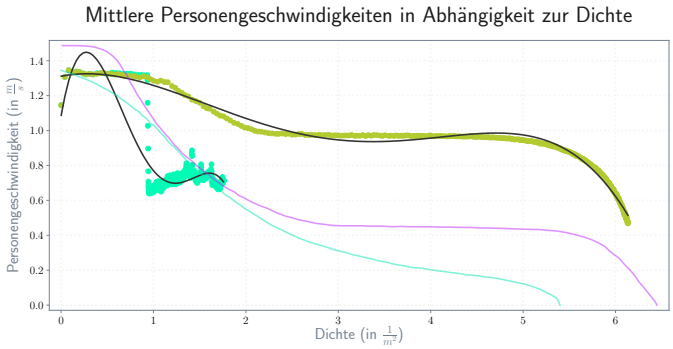


Abbildung 28. Fundamentaldiagramm Test mit sehr hoher Geduld (mintgrüne Kurve) und keiner Geduld (grün-gelbliche Kurve).

Weiter untersuchen wir die Auswirkungen des Diagramms beim Variieren des Beachtungsradius der Abstoßung. Die Variationen umfassen den Radius 0, 2, 10 und 99999 und sind in Abbildung 29 dargestellt. Es fällt sofort auf, dass die Radien 10 und 99999 aufeinander liegen, eine weitere Erhöhung abgesehen vom erhöhten Rechenaufwand keine oder kaum Auswirkungen zu haben scheint. Zusammenfassend kann man jedoch beobachten, dass der Radius durchaus eine Auswirkung auf die Kurvenform hat.

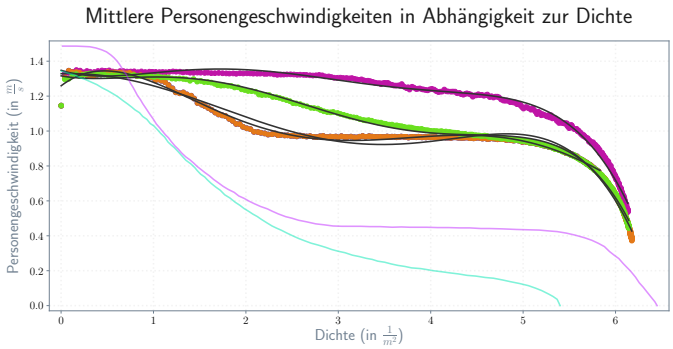


Abbildung 29. Fundamentaldiagramm Test mit Radius 0 (lila), 2 (grün), 10 (orange) und 99999 (orange).

Ähnlich zur lila Kurve beim niedrigen Bewegungsradius gestaltet sich die Auswirkung der Glättungskern Reichweite 1 in Diagramm 30. Außerdem lässt sich anhand der Reichweiten 5 und 99999 beobachten, dass der Abstieg der mittleren Geschwindigkeit maßgeblich beeinflusst wird.

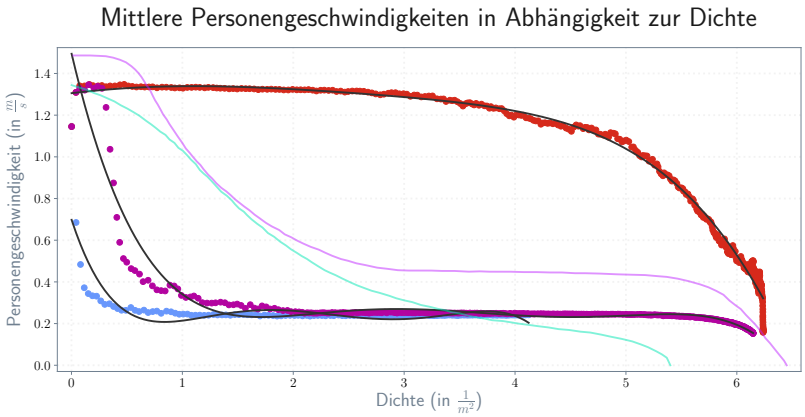


Abbildung 30. Fundamentaldiagramm Test mit den Glättungskern Reichweiten 1 (rot), 5 (lila) und 99999 (blau).

Der letzte Parameter, die Stärke des Glättungskerns, wird ebenfalls auf 0, 5, 5, 0 und 999999 variiert. Das entstandene Fundamentaldiagramm in Abbildung 31 zeigt für höhere Stärken einen schnellen Abstieg der mittleren Geschwindigkeit - schon bei niedrigeren Dichten. Für geringe Werte kann mit der Stärke der Zeitpunkt des Geschwindigkeitsverlusts bestimmt werden, wie die rote Kurve zeigt.

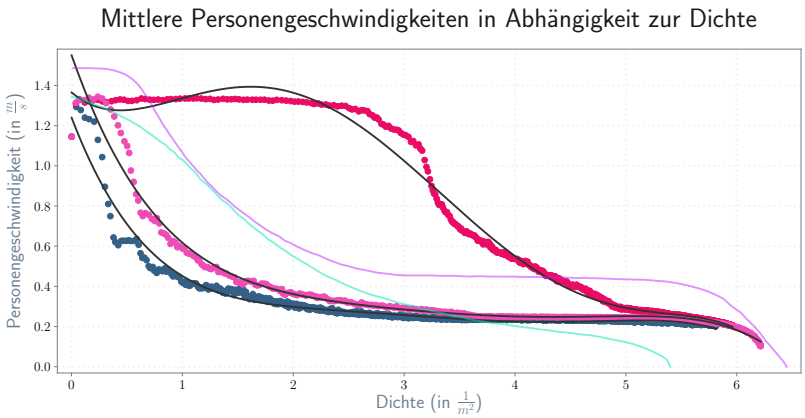


Abbildung 31. Fundamentaldiagramm Test mit den Glättungskern Stärken 0, 5 (rot), 5.0 (lila) und 999999 (türkis).

Anhand dieser Beobachtungen kann der Versuch gestartet werden die Parameter noch deutlich besser zu kalibrieren, als uns das mit der Basisversion gelungen ist. Auch die Anwendung eines Maximum-Likelihood-Schätzers ist denkbar um die bestmögliche Konfiguration zu finden. Das ist jedoch durch den enormen zeitlichen Aufwand in dieser Arbeit nicht mehr möglich. Die durchschnittliche Simulationszeit für das Fundamentaldiagramm betrug $\approx$ 3 Stunden bis Dichten > 6 erreicht wurden.

4 Zusammenfassung und Bewertung (Stiglmeier)

4.1 Fazit

Im Zuge dieser Arbeit wurde ein Personenstromsimulator entwickelt.Dieser basiert auf dem Modell eines Zellularautomaten mit einem event-driven Scheduler. Um eine Simulation möglichst benutzerfreundlich zu erstellen, wurde eine GUI sowie ein Command-Line-Interface dafür implementiert.

In der GUI kann ein beliebig großes Feld ausgewählt und mit Objekten bestückt werden. Zur Auswahl stehen Quellen, Ziele, Personen, Hindernisse und Lichtschranken zur Durchflussmessung. Quellen und Ziele können anschließend nach Wunsch konfiguriert werden. Dabei ist unter anderem möglich, die Geschwindigkeit und Erstellungsrate, die Entfernungstrategie, sowie das Bewegungsmuster (Euklidisch, Dijkstra oder Fast-Marching) von Personen einzustellen. Simuliert wird dabei entweder einmalig über die GUI oder per Einstellung im Command-Line-Interface mit Batch-Jobs

Zur Verifikation des Simulators wurden Unit-Tests für jedes Modul des Simulators geschrieben. Dabei wurde eine Testabdeckung von 80 % erreicht. Zusätzlich wurden zur Validierung mehrere, durch den RiMEA-Verein bereitgestellte, standardisierte Testfälle für Personenstromsimulatoren ausgewählt und gegen den entwickelten Simulator ausgeführt.

Der Test zum freien Fluss zeigte dabei auf, dass die Wunschgeschwindigkeit bei freier Bewegung von Personen auf dem Feld in der Regel eine Genauigkeit von über 95 % aufweist. Der Hühnertest zeigte auf, dass die euklidische Bewegung den Test zwar nicht besteht, aber die Bewegung nach Dijkstra und nach der Fast-Marching Methode den Test jedoch erfolgreich bestehen. Der Test zur Evakuierung eines Raum mit zwei bzw. vier Türen wurde ebenfalls bestanden, da sich der Zeitunterschied zwischen den beiden Simulationen um den erwarteten Faktor von 2 unterscheidet. Beim Test des Fundamentaldiagramms, also der Bewegung von Personen in einem Gang, weist die Implementierung zwar qualitativ das richtige Ergebnis auf, sollte aber noch weiter kalibriert werden, um das erwartete Verhalten exakt zu treffen.

Im Rückblick betrachtet war eine erste, funktionale Implementierung sehr schnell gelungen. Jedoch ist das erfolgreiche Bestehen der Validierungstests kein triviales Unterfangen und erfordert viel Zeitaufwand. Eine gute Performance durch Optimierung der Rechenzeiten leider ebenfalls schwierig zu erreichen. Der hier entwickelte Simulator beschränkt sich auf ein zweidimensionales Feld und kommt bereits ab einer Feldgröße von 150x150 Zellen bei der ein oder anderen Konfiguration an seine Grenzen.

Aus diesem Grund werden im Folgenden einige Möglichkeiten der Verbesserung und Erweiterung des Simulators vorgestellt.

4.2 Weitere Untersuchungsvorschläge

Die Stabilisierung der Ergebnisse und Erhöhung der Performance ist für die langfristige Nutzung elementar. Hierbei sind die folgenden Verbesserungen zu unternehmen:

- Herausfinden warum so ein schlagartiger Geschwindigkeitsabfall im Fundamentaldiagramm bei Dichten um 1 zu beobachten sind
- Maximum Likelihood Klassifikator um den Simulator auf eine beliebige Kurve (z. B. Weidmann) bestmöglich zu kalibrieren
- Neuimplementierung des Dijkstra Algorithmus um bei großen Feldern keinen StackOverflow auszulösen

Um den Simulator sinnvoll zu erweitern, sind während der Implementierung die folgenden Ideen entstanden:

- Erweiterung des Feldes in den dreidimensionalen Bereich
- Personen haben eine maximale Sichtweite
- Personen wollen einer Kette von Zielen entlang laufen

Literaturverzeichnis

Cornett. 2013. “Minimum Acceptable Code Coverage”. Besucht am 8. Juni 2020. https://www.bullseye.com/minimum.html.

“Eikonal equation - Wikipedia”. 2020. [Online; accessed 3. Jun. 2020], Februar. https://en.wikipedia.org/wiki/Eikonal_equation#Numerical_approximation.

Fu, Y., J. Liang, Q. Liu und X. Hu. 2014. “Crowd Simulation for Evacuation Behaviors Based on Multi-agent System and Cellular Automaton”. In *2014 International Conference on Virtual Reality and Visualization,* 103–109.

Goll, J. 2019. “Übersicht über die vorgestellten Entwurfsprinzipien und Konzepte”. In *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik: Strategien für schwach gekoppelte, korrekte und stabile Software,* 155–161. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-25975-4. https://doi.org/10.1007/978-3-658-25975-4_9. https://doi.org/10.1007/978-3-658-25975-4_9.

Hall, E. T. 1966. *The Hidden Dimension.* Doubleday. ISBN: 9780385084765.

“Hankin and Wright - Fundamentaldiagramm”. 2020. [Online; accessed 4. Jun. 2020], Juni. https://www.asim.uni-wuppertal.de/de/datenbank/data-from-literature/fundamental-diagrams/hankin-and-wright.html.

Klüpfel, H. 2003. “A Cellular automaton model for crowd movement and egress simulation” (Juli).

Rasch, C., und T. Satzger. 2008. “Remarks on the implementation of the fast marching method”. *IMA Journal of Numerical Analysis* 29, Nr. 3 (November): 806–813. https://doi.org/10.1093/imanum/drm028. https://doi.org/10.1093/imanum/drm028.

“RiMEA e.V.” 2020. [Online; accessed 31. May 2020], Mai. https://rimea.de.

Sarmady, S., F. Haron und A. Z. Talib. 2010. “Simulating Crowd Movements Using Fine Grid Cellular Automata”. In *2010 12th International Conference on Computer Modelling and Simulation,* 428–433.

“Verhaltensregeln und -empfehlungen zum Schutz vor dem Coronavirus im Alltag und im Miteinander”. 2020, 8. Juni 2020. Besucht am 8. Juni 2020. https://www.infektionsschutz.de/coronavirus/verhaltensregeln.html.

Voorhees, D. P. 2020. “Introduction to Model-View-Controller”. In *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models,* 175–179. Cham: Springer International Publishing. ISBN: 978-3-030-28501-2. https://doi.org/10.1007/978-3-030-28501-2_14. https://doi.org/10.1007/978-3-030-28501-2_14.

“Weidmann - Fundamentaldiagramm”. 2020. [Online; accessed 4. Jun. 2020], Juni. https://www.asim.uni-wuppertal.de/de/datenbank/data-from-literature/fundamental-diagrams/weidman.html.

Zhou, S. 2009. “Study on the Evacuation Simulation Based on Cellular Automata”. In *2009 International Conference on Information Technology and Computer Science,* 2:481–484.