

Fußgängersimulation

Projektarbeit im Fach Scientific Computing

Jakob v. d. Heydt, Sönke Beier, Kevin Meißner

Juni 2019

Inhaltsverzeichnis

1. Aufgabenstellung	3
1.1. Verhalten von Fußgängern	4
2. Funktionsweise der Simulation	5
2.1. Erstellen des statischen Feldes S	5
2.1.1. Mehrere Ziele	6
2.1.2. Spezialfall Korridor	6
2.2. Erstellen des Dynamischen Feldes D	7
2.2.1. Modifizierungen	7
2.3. Bewegungsvorgang	8
2.3.1. Transition Matrix	8
2.3.2. Gewichtung der Felder	10
2.3.3. Updateregeln	11
2.3.4. Konflikte	11
2.3.5. Panik	12
2.4. Zusammenfassung der Funktionsweise	13
3. Eigenschaften des Systems	14
3.1. Statisches Feld und Reibung	14
3.2. Dynamisches Feld und Diffusion	15
3.3. Panikparameter	17
4. Anwendungen der Simulation	19
4.1. Einfluss von Hindernissen	19
4.2. Flaschenhals-Effekt	23
4.3. Korridor	26
4.4. Weitere Situationen	27
5. Probleme	34
6. Fazit und Ausblick	35
7. Quellen	36
A. Programablaufpläne	II
A.1. Erzeugen des statischen Feldes	II
A.2. Ablauen aller Nachbarfelder	III
A.3. Erzeugen des dynamischen Feldes	IV
A.4. Decay des dynamischen Feldes	V
A.5. Diffusion des dynamischen Feldes	VI
A.6. Erstellen der Transition Matrix	VI
A.7. Bewegungsentscheidung einer Person	VII
A.8. Bewegung der Personen	VIII

A.9. Panikschwelle	IX
A.10. Initialisierung aller Objekte	X
A.11. Funktion der Hauptklasse	XI
B. Quellcode	XII
B.1. initial situation.cpp	XIII
B.2. main s.cpp	XVI
B.3. klassen.cpp	XXXIII

1. Aufgabenstellung

In einer Situation, wo viele Leute zu einem oder wenigen Zielen wollen, können besonders viele Interaktionen beobachtet werden. Hierbei kann man in der Realität Panikverhalten beobachten. Dies wollen wir simulieren, um Laufwege und Panikentstehung besser zu verstehen. Für Evakuierungsszenarios können dies entscheidende Daten zum Verhalten von Menschenmassen sein, was zur Verbesserung von Laufwegen, Navigation, Planung der Ausgänge und Ähnlichem genutzt werden kann, um die Evakuierungszeit zu verringern und Verletzungen (aufgrund von Quetschungen o.Ä.) zu verhindern.

Das Ziel unseres Projektes ist, eine möglichst realistische Simulation der Dynamik von Fußgängern zu entwickeln. Es sollen bestenfalls verschiedene Effekte abgebildet werden, die durch menschliches Verhalten, insbesondere durch Interaktion und Panik entstehen. Diese Effekte werden später unter 1.1 vorgestellt. Wir haben hierzu verschiedene Szenarien betrachtet: die Grundrisse unserer Gelände variieren in Komplexität und Größe. Je nach Aufbau wird der ein oder andere Effekt besonders stark sichtbar. Hierbei muss zuerst die Grundlage der Bewegung verstanden werden, insbesondere im Fall von hohen Anzahldichten.

Wir sollen folgende 5 Situationen darstellen:

1. Verlassen eines geschlossenen Raums mit einer Tür
2. Effekt eines Hindernisses vor dem Ausgang
3. entgegengesetzt laufende Menschen vor einem schmalen Durchgang
4. entgegengesetzt laufende Menschen in einem Korridor
5. andere interessante Situationen

Auf diese Situationen kommen wir im Laufe des Berichts zurück.

Auf einer höheren, meta-physischen Ebene können Fußgänger als aktive Partikel aufgefasst werden, die sich zweidimensional bewegen. In der Regel bewegen sie sich zu einem bestimmten Ziel (mit Fluktuationen) oder zumindest in eine bestimmte Richtung. Hier lässt sich eine Analogie zum physikalischen Potential finden - die Fußgänger bewegen sich in Richtung des Ziels entlang des Gradienten. Dabei interagieren sie mit ihrem Umfeld, z.B. indem sie Zusammenstöße versuchen zu vermeiden. Dabei wirken diese Interaktionen als „soziale Kraft“ auf die Fußgänger. Dies ist unsere physikalische Motivation, Fußgänger-Dynamiken zu untersuchen. Wir haben uns in unserer Arbeit an der Bachelorarbeit von Christian Nitzsche zum Thema „Cellular automata modeling for pedestrian dynamics“ orientiert und viele seiner Ideen aufgegriffen oder abgewandelt.

1.1. Verhalten von Fußgängern

Um die Dynamiken von einer Masse an Fußgängern zu verstehen, muss man die Mechanismen und die Motivation des einzelnen Fußgängers verstehen. Dabei ist jeder Fußgänger individuell, hat sein individuelles Ziel, eine eigene Geschwindigkeit und eine eigene Route. Unsere Betrachtung schließt individuelle Geschwindigkeiten aus, wir kommen später darauf zurück an welcher Stelle man hier Veränderungen treffen müsste.

Die Route orientiert sich normalerweise an der kürzesten Strecke zum Ziel und wird von Hindernissen wie Wänden und anderen Personen beeinflusst. Dabei kommt es zu verschiedenen Effekten:

1. Es bilden sich Spuren aus für Personen die sich in unterschiedliche Richtungen bewegen, ähnlich wie auf der Straße. Dies reduziert die Anzahl an Kollisionen und ermöglicht einen höheren Fluss.
2. Es bilden sich große Menschentrauben vor Ausgängen. Ein Ausgang ist i.d.R. eine starke Verengung, durch die viele Leute wollen. Dies verringert den Fluss.
3. An Durchgängen, durch die Personen in unterschiedliche Richtungen wollen, kommt es zu einem oszillierendem Durchgangsverhalten.

An diesem Wissen wollen wir uns in unserem Programm orientieren.

2. Funktionsweise der Simulation

Es soll hier Schritt für Schritt die Funktionsweise der Simulation erklärt werden. Unser Ansatz basiert auf dem Konzept eines zellulären Automaten. Hierzu wird der Raum in Zellen unterteilt, die jeweils einen Menschen „beinhalten“ können. Hieraus ergeben sich verschiedene Zustände für jede Zelle:

- Zelle ist leer
- Zelle ist besetzt
- Zelle ist ein Hindernis

Dies ergibt dann auch die Regeln für eine Bewegung - ein Fußgänger kann sich nur in eine Zelle bewegen, wenn diese leer ist.¹ Unser Programm ist jedoch kein richtiger zellulärer Automat, da die Zellen nicht über das ganze Programm bestehen bleiben, sondern in einzelnen Arrays gespeichert sind.

Wir lesen den Raum und die Fußgänger grafisch ein, in unserem Fall entspricht unsere Zelle also einem Pixel. Damit ist jede Zelle quadratisch. Für eine Bewegung kommen nur die anliegenden Zellen infrage, das bedeutet jeder Fußgänger hat theoretisch 4 Möglichkeiten zu gehen. Außerdem ist es für jede Person möglich in der ursprünglichen Zelle stehen zu bleiben. Die Wahrscheinlichkeit, in eine bestimmte Richtung zu gehen, hat grundlegend erst einmal mit der Nähe zum Ziel zu tun. Dies wird durch das Statische Feld S wiedergegeben.

2.1. Erstellen des statischen Feldes S

Das S-Feld wird benutzt, um ein Potenzialfeld zu konstruieren. Jeder Zelle wird ein bestimmter Wert zugeordnet. Dieser Wert gibt wieder, wieviele Schritte gegangen werden müssen um zum Ziel zu gelangen und kann als Abstand zum Ziel d_{ij} aufgefasst werden. Dafür wird beim Ziel begonnen, welches mit dem Wert 0 belegt wird. Dann werden die direkten Nachbarn (also die 4 möglichen Zielzellen), mit dem Wert 1 belegt, insofern sich eine Person darauf befinden darf. Von dort braucht man genau einen Schritt zum Ziel. Dann werden wiederum die Nachbarn der Nachbarn belegt. Hier muss nun berücksichtigt werden, dass einer der Nachbarn (das Ziel) schon belegt wurde! Es soll immer nur der kleinste Wert (die kürzeste Strecke) berücksichtigt werden, also wird der Wert für bereits belegte Zellen nicht überschrieben.

Die Prüfung erfolgt, bis alle Zellen mit einem Wert belegt wurden. Hindernisse werden dabei nicht als Nachbarn gewertet, der Zähler läuft also „um ein Hindernis herum“.² Um unsere Werte nun in ein Potenzial zu überführen, berücksichtigen wir die maximale Distanz d_{max} . Das Potential, und damit unser S Feld einer Zelle ij, wird damit bestimmt durch

¹Personen dürfen auch nicht die festgelegte Feldgrenze verlassen

²Wie das im Programm detailliert abläuft haben wir dargestellt in einem Programmablaufplan A.1 bzw. unter A.2 im Anhang.

$$S_{ij} = d_{max} - d_{ij} \quad (1)$$

Das größte Potential liegt dann beim Ziel mit

$$S_Z = d_{max} - d_Z = d_{max} - 0 = d_{max} \quad (2)$$

2.1.1. Mehrere Ziele

Gibt es mehrere Ziele, so muss für jedes Ziel ein eigenes S-Feld erstellt werden. Es wird dann also für jedes Ziel jeweils für jede Zelle ein Wert S_{kij} berechnet. Diese Werte gehen dann gewichtet in den Wert S_{ij} für jede Zelle ein. Es gilt dabei:

$$S_{ij} = \sum_k w_k \cdot S_{kij}$$

Die Gewichtung w_k ³ soll wiedergeben, dass die Personen unterschiedlich viel von den einzelnen Ausgängen wissen. Es ist bekannt, dass Personen (insbesondere in Paniksituationen) den Ausgang bevorzugen, durch den sie das Gelände betreten haben. Trotzdem wissen die Personen in der Regel von den anderen Ausgängen, beispielsweise weil sie an einem vorbeigelaufen sind oder weil sie eine Beschilderung gesehen haben. Um den Effekt wiederzugeben, dass sie am liebsten den gleichen Ausgang benutzen, wird für jeden Fußgänger individuell ein Ausgang bestimmt, der mit dem Faktor 2 in die Gewichtung eingeht. Alle anderen werden mit Werten zwischen 0 und 1 belegt. Somit ergibt sich immer eine klare Präferenz.

2.1.2. Spezialfall Korridor

Eine Situation wie in 1 skizziert ist in unserer Simulation als „Korridor“ gekennzeichnet. - ein langer, schmaler Gang mit zwei Ausgängen am linken und rechten Ende. Hierfür ergibt sich eine leicht andere Berechnung des S Feldes. Es wird lediglich die x Komponente als Abstand berücksichtigt. Würde das Feld wie oben beschrieben berechnet, würden sich starke Verdichtungen in der Mitte des Ganges ergeben. Ein Korridor ist vereinfacht ein ausgedehntes eindimensionales Problem - deshalb erfolgt die Berechnung auch eindimensional. Der Abstand zum Ziel ergibt sich dann aus der x-Koordinate der Zelle, jedoch unterschiedlich für die beiden unterschiedlichen Ziele:

Damit ergibt sich das Potential als Differenz aus maximalem x-Abstand und dem x Wert:

$$S_{ij} = x_{max} - x_{ij} \quad (3)$$

$$S_{ij} = x_{ij} \quad (4)$$

Wobei Formel 3 für den linken Rand und Formel 4 für den rechten Rand gilt.

³Im Programmcode wird w mit w_S bezeichnet. Da diese Bezeichnung hier in der mathematischen Umgebung von Latex für Verwirrungen geführt hätte, benutzen wir stattdessen w

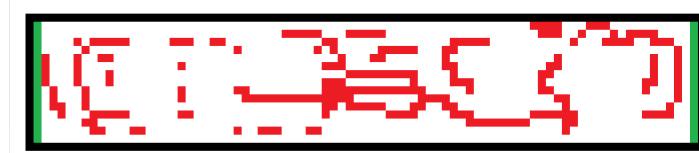


Abbildung 1: Korridor mit Personen in rot

2.2. Erstellen des Dynamischen Feldes D

Das S-Feld orientiert sich an reinem Wissen über die Ziele, kann also quasi als rein rationaler Einfluss beschrieben werden. In Umgebungen mit vielen Menschen gewinnen die Interaktionen jedoch mehr und mehr an Gewicht. Dies stellen wir durch ein dynamisches Feld dar. Fußgänger orientieren sich an anderen Fußgänger und laufen denjenigen mit einer ähnlichen Richtung bzw. mit demselben Ziel hinterher. Sie stellen also selbst so etwas wie ein Potenzial dar, welches die soziale Kraft beschreibt.

Um dieses Prinzip auf unseren zellulären Automat zu überführen, speichern wir die Aktivität aller Fußgänger ab. Wir führen nun zusätzlich zum S Feld ein zweites Feld ein. Das D-Feld wird für jeden Fußgänger individuell erstellt. Hier taucht auf, wo und in welche Richtung viele Menschen langgegangen sind.

Die genaue Erzeugung des D-Felds, für eine Person a, erfolgt so: es wird überprüft, ob eine andere Personen b die letzten zwei Schritte in exakt dieselbe Richtung wie Person a gegangen sind. Ist dies der Fall, wird der Wert des D-Feldes um eins erhöht in der Zelle, die Person b als vorletztes besetzt hatte.⁴

2.2.1. Modifizierungen

Wir nehmen noch zwei Modifizierungen am D-Feld vor. In der Realität ist eine zeitliche Abhängigkeit dieser sozialen Kraft gegeben - wenn eine Person vor langer Zeit mal diesen Weg gegangen ist, ist der Effekt verschwunden. Das bedeutet, der Effekt ist stärker wenn unmittelbar vor einem jemand läuft und schwächer, wenn das schon länger her ist.

Dieser zeitliche Abbau wird bei uns durch den Parameter *Decay* wiedergeben. Der Parameter nimmt Werte an zwischen 0 und 1 und gibt damit eine Wahrscheinlichkeit an, dass das D-Feld einer Zelle ij um eins verringert wird. Dieser Prozess wird genau so oft wiederholt, wie der Betrag des D-Felds ist. Das bedeutet für

$$D_{ij} = 4$$

wird der Prozess des Zerfalls 4 mal wiederholt. mit einer Wahrscheinlichkeit von *Decay*⁴ besitzt die Zelle danach den Wert null, mit einer Wahrscheinlichkeit von *Decay*⁴⁻¹ den Wert 1 und so weiter.⁵

⁴Auch hier gibt es einen konkreten Ablaufplan unter A.3.

⁵ siehe Programmablaufplan A.4

Ganz analog führen wir einen Parameter *Diffusion* ein. Damit wollen wir darstellen, dass Spuren „verwischen“. Eine Bewegung von Menschen wird oft nicht ganz klar örtlich abgegrenzt, sondern es ist hauptsächlich der grobe Ort und die Richtung von Bedeutung. Dieses Verwischen wird durch Übertragen von D-Feld Werten einer Zelle auf angrenzende Zellen dargestellt. Dabei ist der Wert *Diffusion* wieder eine Wahrscheinlichkeit zwischen 0 und 1 und der Vorgang wird genau $|D_{ij}|$ - mal wiederholt. Dabei wird Runde für Runde zufällig ausgewählt auf welche der angrenzenden Zellen ein Zählerwert übertragen wird.⁶ In der Simulation wollen wir den Einfluss dieser Parameter untersuchen.

2.3. Bewegungsvorgang

Nachdem wir unsere wesentlichen Bestandteile erklärt haben, wollen wir jetzt darauf eingehen wie genau die Bewegung der Personen erfolgt. Hierzu muss erklärt werden, wie das Bewegungsupdate der Personen erfolgt und wie die Bewegungsziele für jede einzelne Person ausgewählt werden.

2.3.1. Transition Matrix

Um auszuwählen, welchen Schritt die Person machen möchte, bilden wir eine 3x3 Transition Matrix. Sie bildet alle Nachbarn der Zelle ab. Die Einträge spiegeln Wahrscheinlichkeiten wieder, dass die Person dorthin geht bzw. gehen möchte (wir werden später sehen, dass es zu Konflikten kommen wird, wenn mehrere Personen in dieselbe Zelle wollen). Dabei muss beachtet werden, dass die Wahrscheinlichkeit normiert werden muss. Liegt nun also eine Transition Matrix mit Wahrscheinlichkeiten vor, muss die summierte Wahrscheinlichkeit bei genau 1 liegen - eine Bewegung (inkl. stehen bleiben) wird auf jeden Fall ausgeführt. Damit liegen auch die einzelnen Einträge zwischen 0 und 1.

Diagonale Bewegungen sind verboten, weshalb die Wahrscheinlichkeiten einer Bewegung dorthin sich zu 0 ergeben. Ebenso ausgeschlossen werden Bewegung auf Zellen, die zum jeweiligen Zeitpunkt bereits besetzt sind und Bewegung auf ein Hindernis. Eine Transition Matrix kann dann wie folgt aussehen, wenn beispielsweise das untere Feld besetzt ist:

0	0,4	0
0,2	0,25	0,15
0	0	0

Die Einträge in allen weiteren Zellen $\neq 0$ unserer Transitionmatrix berechnen sich wie folgt:

$$T_{ij} = N \cdot e^{S_{ij} + D_{ij}} \quad (5)$$

⁶ siehe Programmablaufplan A.5

- S := Wert des statischen Feldes
 D := Wert des dynamischen Feldes
 V := Normierungsfaktor

N ergibt sich also aus

$$N = 1 / \sum T_{i,j}$$

wobei die Summe über alle vier Nachbarfelder läuft. Wir benutzen eine e-Funktion, damit die Werte mehr variieren und sich damit klarere Bewegungspräferenzen ergeben.⁷

Um nun eine Zelle auszuwählen, wird eine Zufallszahl erzeugt, die ebenfalls zwischen 0 und 1 liegt. Damit wird entschieden, welche Bewegung ausgeführt wird. Näher beschrieben ist dies in Abbildung 2.

⁷siehe Programmablaufplan A.6

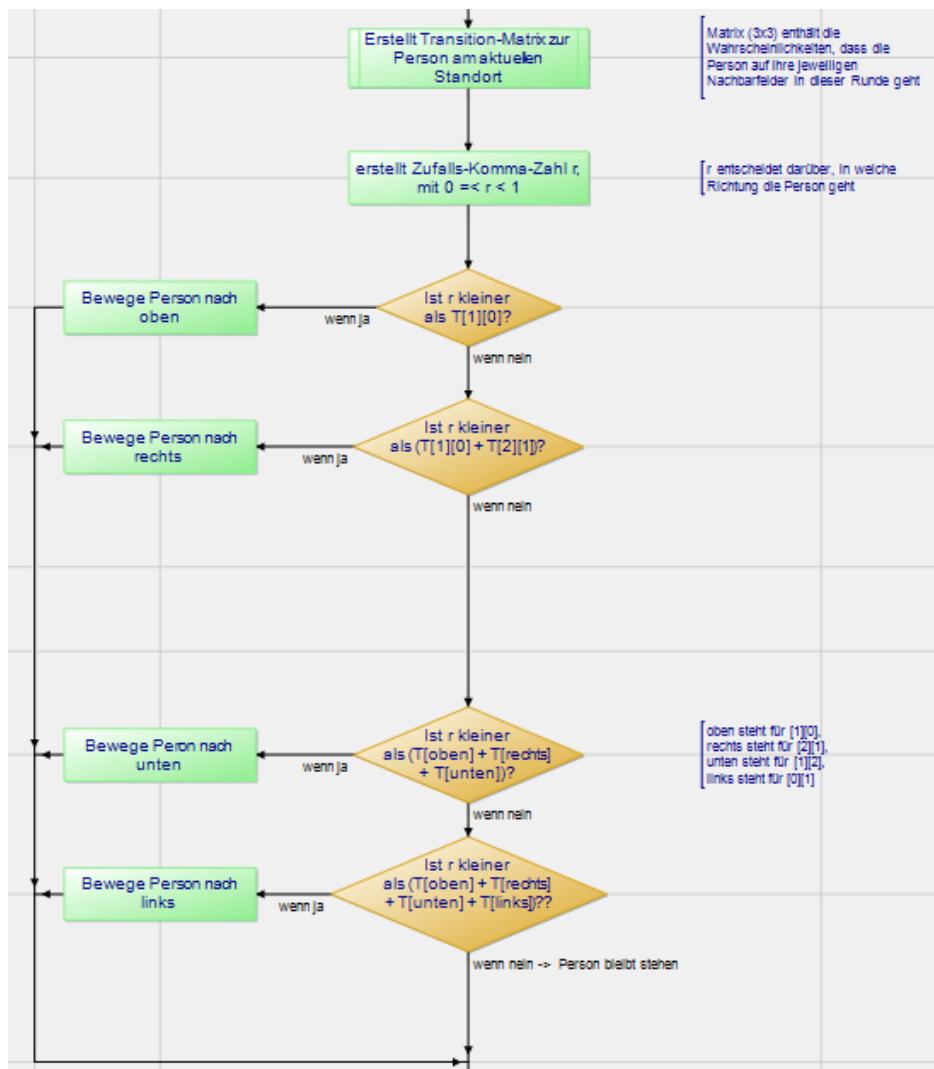


Abbildung 2: Bewegungsentscheidung einer Person

2.3.2. Gewichtung der Felder

In realen Situationen kann das Verhalten sehr stark von rationalem Verhalten abweichen. Nichtsdestotrotz haben wir keine völlig irrationale, randomisierte Bewegung - es findet also scheinbar je nach Situation eine Gewichtung zwischen rational und instinktiv statt. Um dies wiederzugeben, führen wir durch Gewichtungskoeffizienten für das D und S Feld ein. Hiermit können wir unterschiedliche Gewichtungen ausprobieren für die Simulation, um zu einem möglichst realistischem Ergebnis zu kommen und um bestimmte Situationen von extremer Panik darstellen zu können. Somit verändert sich die Berechnung der Einträge in der Transitionmatrix:

$$T_{ij} = N * e^{k_S * S_{ij} + k_D * D_{ij}} \quad (6)$$

Auch hier werden wir später in unseren Simulationen unterschiedliche Konfigurationen betrachten.

2.3.3. Updateregeln

Es ist zu unterscheiden zwischen zwei verschiedenen Arten von Updateregeln: sequenzielle und parallele. Bei sequenziellem Update wird eine Person nach der anderen betrachtet, also alle nacheinander. Paralleles Update bedeutet, dass alle Personen gleichzeitig in ihrer Position aktualisiert werden. Hier gibt es also einen Zeitschritt $t \rightarrow t + 1$, wobei alle Personen einen Zug machen (nämlich entweder stehenbleiben oder gehen). Beide Varianten haben wir eingebaut, näher erläutert im Anhang⁸. Das parallele Update kommt dem menschlichen Verhalten jedoch deutlich näher. Es ist jedoch trotzdem nicht perfekt, da mit unserem zellulärem Automaten lediglich Bewegungen von 1 Schritt pro Zeitschritt erlaubt sind. Hier lassen sich also keine unterschiedliche Bewegungsgeschwindigkeiten darstellen. Es gibt lediglich eine relative Geschwindigkeit zum Ziel, die gegeben ist insbesondere durch die Gewichtung des statischen Felds. Dies lässt sich gut in der Anwendung der Simulation zeigen.

2.3.4. Konflikte

Wie schon erwähnt, kann es im Zuge der Auswahl von Bewegungszielen zu Konflikten zwischen unterschiedlichen Personen kommen, insofern sie zur selben Zelle wollen. Ist dies der Fall, wird analog zur Transition Matrix eine Konfliktmatrix gebildet. Im Mittelpunkt steht die Zelle, um die der Konflikt ausgetragen wird. Die jeweiligen Personen werden dann an der Position eingetragen, an der sie relativ zu der Zelle stehen. Die Einträge in die Zelle kommen aus den Einträgen in der Transition Matrix der jeweiligen Personen. Auch hier wird wieder eine Normierung vorgenommen. Eine Konfliktmatrix kann dann genauso aussehen, wie unsere Transitionmatrix in [??](#). Der Konflikt wird dann analog zur Transitionmatrix über eine Zufallszahl zwischen 0 und 1 entschieden. Der Gewinner des Konflikts bewegt sich in die Zelle, die Verlierer müssen stehenbleiben.

Weiterhin gibt es einen Parameter, mit dem für alle Personen die Anzahl der Konflikte gezählt wird. Für alle beteiligten Personen wird dieser Zählindex dann nach dem Konflikt um eins erhöht. Dies wird später noch eine Rolle spielen. Sollte eine Person komplett von anderen Personen und/oder Hindernissen umgeben sein - das heißt sie ist gezwungen, stehen zu bleiben - wird dies ebenfalls als Konflikt gezählt.

⁸Programmablaufplan A.8

Reibung

Zusätzlich zu den sozialen Kräften, die wir im D-Feld berücksichtigt haben, gibt es eine weitere soziale Interaktion, die wir einbauen wollen. Kommt es zu einem Konflikt zwischen zwei Personen, kann es aufgrund von Höflichkeit, Verwirrung oder auch Überforderung dazu kommen, dass beide stehen bleiben. Dies kann man als Reibung zwischen den Personen (beziehungsweise Teilchen) auffassen.

Wir führen dafür also einen Parameter *Friction* ein. Dieser gibt die Wahrscheinlichkeit wieder, dass der Gewinner eines Konflikts stehenbleibt - also keiner der Personen geht. Der Konflikt wird logischerweise für beide beteiligten Personen trotzdem als Konflikt gezählt.

2.3.5. Panik

Eine besondere Funktion solcher Simulationen ist, die Auswirkungen von Panik auf Fußgängerodynamik zu untersuchen. Panik lässt sich als besonders irrationaler Zustand beschreiben, in dem das rationale Wissen in den Hintergrund gerät. Hier spielen also tatsächlich mehr die Interaktionen mit anderen Personen eine Rolle. Das bedeutet, in einem Zustand der Panik sollte sich in unserer Simulation die Gewichtung des D-Felds vergrößern im Vergleich zum S-Feld. Mit jedem Konflikt steigt die Panik einer Person an. Dieser Effekt kann ebenso abschwellen, sollte die Person wieder mehr Freiraum besitzen. Bei einem Konflikt wird der Wert k_D , also die Gewichtung des D-Felds, um eins erhöht. Hierbei werden allerdings nur direkt hintereinander folgende Konflikte berücksichtigt. Der Wert sinkt wieder um eins, wenn die Person keinen Konflikt hatte.

Es lässt sich zudem ein Parameter belegen, der die Schwelle an Konflikten bestimmt bis zum Panikausbruch der Person. Ab hier werden auch Personen in der Umgebung von der Panik angesteckt. Ist eine Person in Panik, wird nach jedem Zug der Zählindex der Konflikte aller Nachbarn um eins erhöht.⁹.

Nach ersten Simulationen haben wir festgestellt, dass Menschen in Panik wegen des zu hohen k_D selbst wenn sie sich in unmittelbarer Nähe zum Ziel befinden, trotzdem nicht dorhtin gehen. Dieses Verhalten ist jedoch so unrealistisch, dass wir eine Modifizierung treffen mussten. Wenn sich eine Person in der Nähe des Ziels befindet, wird die Panik automatisch auf 0 gesetzt. Der Einzugsradius beträgt dabei 5 Zellen. Damit wird verhindert, dass Menschen vor dem Ausgang alle anderen Personen blockieren.

⁹genauer im Anhang A.9

2.4. Zusammenfassung der Funktionsweise

Bevor wir uns der Anwendung und den einzelnen Tests widmen, wollen wir noch einmal kurz die grobe Funktionsweise zusammenfassen:

Durch einen grafischen Input, der in Objekte (Personen, Hindernisse, Ziele) umgewandelt wird, kreieren wir eine Ausgangssituation.¹⁰ Mit jedem Zeitschritt verändert sich die Position der Personen, was wir grafisch aktualisieren (s. Abb. 3). Die Bewegungen der Personen sind von verschiedenen Parametern abhängig: ihre Entfernung zum Ziel ist entscheidend (S-Feld), ebenso wie die Bewegung der sie umgebenden Menschen (D-Feld). Wird die Person oft daran gehindert, einen Zug zu machen (viele Konflikte), kann sie in Panik fallen. Dabei wird das rationale Wissen (S-Feld) vernachlässigt. Panik kann sich unter den Personen ausbreiten. Haben alle Personen ihr Ziel erreicht, wird die grafische Aktualisierung abgebrochen und die gesammelten Daten werden in einer Textdatei gespeichert. Somit ist es uns möglich, Evakuierungszeiten beziehungsweise Iterationen in Abhängigkeit von den unterschiedlichen Parametern zu analysieren.

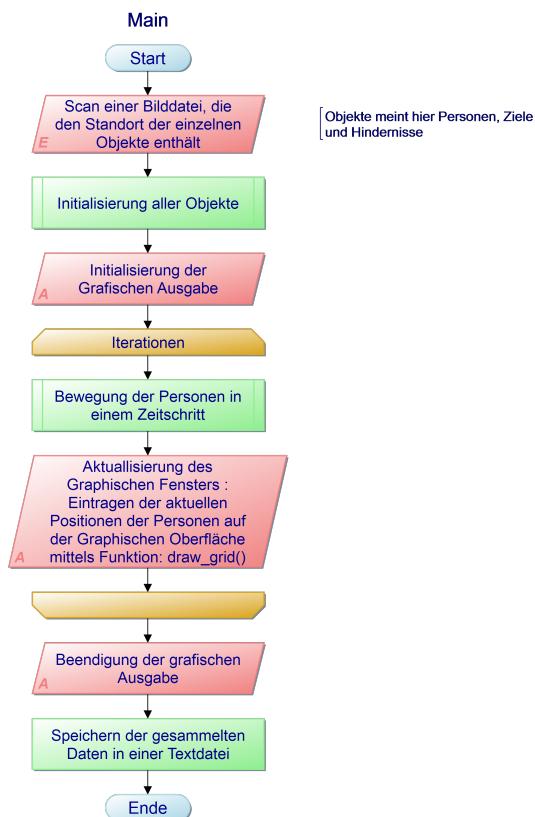


Abbildung 3: Grober Ablauf des Programms

¹⁰Im Anhang A.10 ist die Initialisierung aller Objekte dargestellt.

3. Eigenschaften des Systems

Zuerst wollen wir den Einfluss der unter Punkt 2 eingeführten Parameter auf das Verlassen eines einfachen, quadratischen Raumes untersuchen. Hierfür haben wir folgende Ausgangssituation kreiert, mit situationsabhängig unterschiedlichen Personendichten:

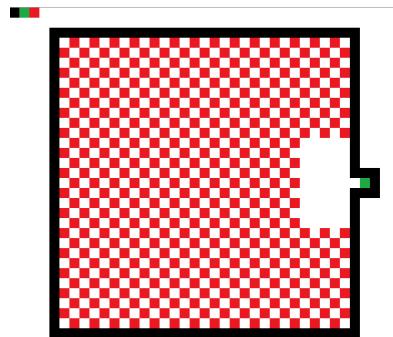


Abbildung 4: Einfacher Raum in schwarz, Personen in rot

3.1. Statisches Feld und Reibung

Zuerst untersuchen wir den Einfluss des statischen Feldes, indem wir dieselbe Simulation für verschiedene Werte von k_S durchführen. Dies haben wir für verschiedene Werte des Parameters *Friction* gemacht, um auch hier den Einfluss analysieren zu können.

Wir erwarten, dass mit steigendem Wert k_S auch die Evakuierungszeit sinken sollte. Denn für einen höheren Wert sollten die Menschen zielgerichteter laufen, also mit höherer Wahrscheinlichkeit den für sie kürzesten Weg einschlagen.

Für den *friction* Parameter erwarten wir gegenteiliges: gibt es mehr Reibung, also mehr Leute die stehen bleiben in unserer Simulation, so sollte es länger dauern bis alle Leute den Raum verlassen haben.

In dieser Situation wird der Einfluss des D-Feldes erst einmal vernachlässigt, weshalb k_D auf 0 gesetzt wurde. Hier sind die Daten, die wir aus unserem Programm erhalten haben:

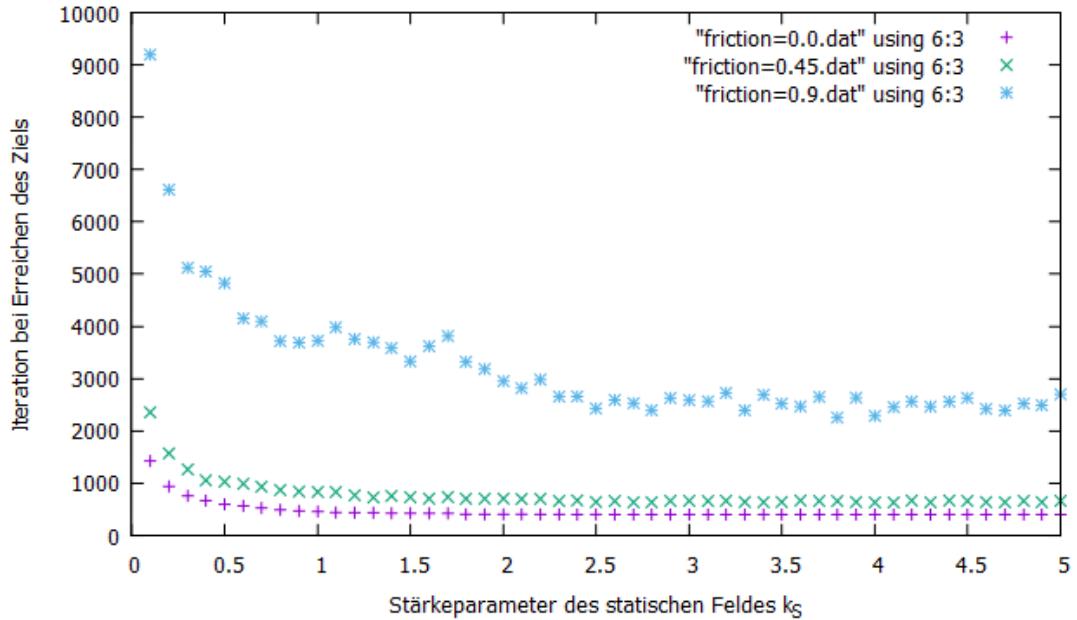


Abbildung 5: Evakuierung in Abhängigkeit von Friction

Es ist in der Grafik gut zu erkennen, dass die Anzahl der Iterationen mit steigendem Wert k_S sinkt. Dies sehen wir für alle drei Kurven, wobei wir bei der Kurve für *Friction* = 0.9 nur von einer Tendenz sprechen können. Hier haben wir immer wieder kleine Ausreißer, da bei höheren Werten Kettenreaktionen (einmal stehenbleiben führt zu vielen weiteren Stillständen) deutlich öfter auftreten können. Insgesamt können wir aber abbilden, was wir vermutet hatten. Der Einfluss scheint jedoch minimal zu sein ab Werten $k_S > 1,5$. Es ist also anzunehmen, dass in dieser Simulation alle Personen schon ab $k_S = 1,5$ den kürzesten Weg nehmen und nur äußerst selten davon abweichen.

Auch die Kurven für die unterschiedlichen Reibungswerte stimmen mit unserer Erwartung überein. Es ist gut zu sehen, dass die Evakuierungszeit für jeden einzelnen angenommen Wert k_S höher ist.

3.2. Dynamisches Feld und Diffusion

Wir wollen nun den Einfluss des D-Felds untersuchen. Für dieselbe Ausgangssituation wie oben, also gleicher Aufbau und gleiche Personenanzahl, schalten wir jetzt das D-Feld hinzu und halten den Wert für k_S fest. Nun untersuchen wir verschiedene Werte für k_D , also verschiedene Gewichtungen der beiden Felder und überprüfen damit verschieden starke Einflüsse der rationalen und der emotionalen Komponente. Dabei zeichnen wir dies für verschiedene Werte der *Diffusion* auf, um analog zur *Friction* im vorherigen Teil auch diesen Einfluss parallel untersuchen zu können.

Was erwarten wir? Mit steigender Gewichtung vom D-Feld sollte unsere Evakuierungszeit

steigen, da wir davon ausgehen dass die Personen von ihrem optimalen Weg abweichen. Der Einfluss der anderen Personen lenkt sie ab und zwingt sie zu Umwegen.

Denselben Effekt erwarten wir, wenn wir den *Diffusion*-Parameter erhöhen. Dies geht einher mit einer höheren Wahrscheinlichkeit einer Ausbreitung des D-Feldes und damit lokalen Verstärkungen des D-Feldes insgesamt. Aus diesem Grund würden wir auch hier davon ausgehen, dass die Evakuierungszeit mit steigendem Wert für die *Diffusion* steigt.

Für unsere Simulation haben wir die sonstigen Parameter wie folgt gewählt:

$$k_S = 2, Decay = 30, Friction = 0$$

Wir können unsere Ergebnisse also mit der unteren, lila Kurve im Punkt $k_S = 2$ vergleichen und diesen Punkt als Referenz benutzen. Auch hier sind die Iterationen wieder in Abhängigkeit von dem Parameter angegeben:

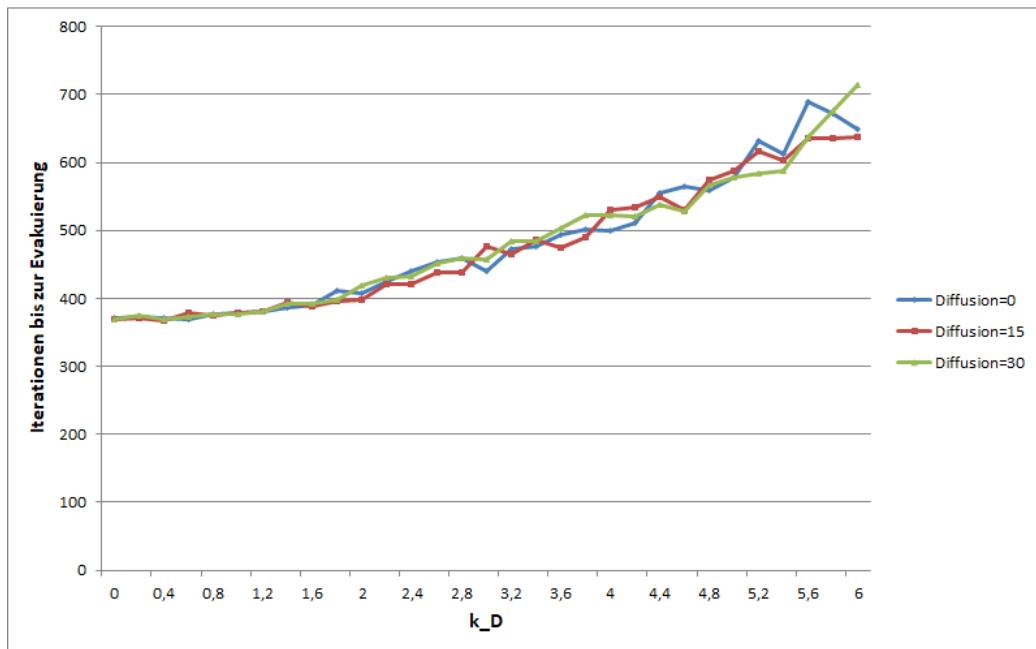


Abbildung 6: Evakuierung in Abhängigkeit vom D-Feld

Aus unseren Ergebnissen lassen sich mehrere Schlussfolgerungen treffen. Verglichen mit dem Wert in unserer ersten Grafik für die entsprechenden Werte, wo zwischen 300 und 400 Iterationen liegen, sind Werte in unserer Grafik 6 kongruent für kleine $k_D < 1,0$. Ab hier sehen wir allerdings einen erkennbaren Anstieg, der kontinuierlich auch für hohe Werte positiv ist. Wir schließen daraus, dass der Einfluss des D-Felds klein ist, solange k_D deutlich kleiner ist als k_S . In diesem Fall scheint die Schwelle ungefähr bei der Hälfte des Wertes von k_S zu liegen. Ab dort wird allerdings der erwartete Einfluss sichtbar und die Iterationen nehmen deutlich zu. Wir erkennen einen nicht-linearen Zusammenhang, weil der Anstieg nicht konstant bleibt sondern die Kurve sich parabelförmig nach oben formt. Bei $k_D = 2 \cdot k_S$ liegen wir schon bei 500 Iterationen und damit fast 150 % des Ausgangswertes. Wir sehen also deutlich, dass das D-Feld zu einer erheblichen Steigerung der Evakuierungszeit führt.

Für den Einfluss der *Diffusion* ergibt sich jedoch keine Tendenz. Es lassen sich nur lokal Aussagen treffen, wie beispielsweise zwischen $k_D = 4,4$ und $5,6$, wo sich unsere Kurve genau entgegengesetzt zu dem verhält, wie wir das erwartet hätten. Nämlich, dass die grüne Kurve (für $Diff = 30$) unter den beiden anderen liegt und somit in diesem Bereich einer schnelleren Evakuierung entspricht. In anderen Bereichen lässt sich diese Aussage jedoch genauso umdrehen, insgesamt ergibt sich hier kein klares Bild. Mit höheren Werten könnte man eventuell ein klareres Bild bekommen, jedoch hat uns bei Werten ab $Diff = 40$ die Rechenleistung gefehlt, um Simulationsdaten zu erhalten. Abschließend kann der Effekt der *Diffusion* also nicht analysiert werden.

3.3. Panikparameter

Als letztes wollen wir uns noch dem Einfluss unserer Panikschwelle auf die Evakuierung widmen. Hierzu bedienen wir und wieder der einfachen Ausgangssituation und tragen die Evakuierungszeit in Abhängigkeit von unserer Panikschwelle auf, welches wir für unterschiedliche Gewichtungen k_D vornehmen. Hierbei bleibt

$$k_S = 20$$

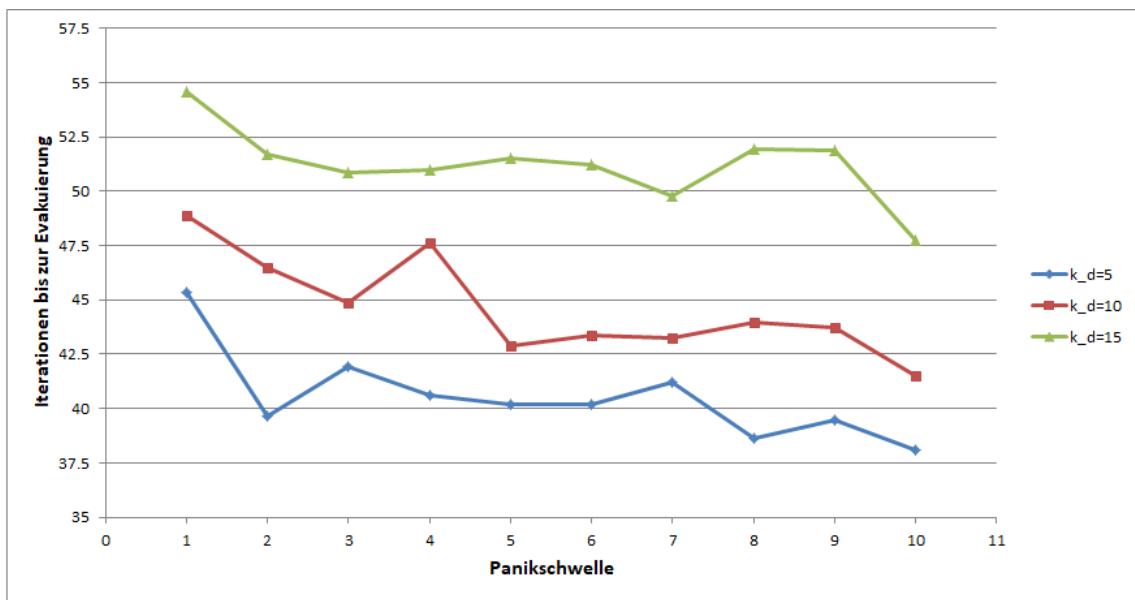


Abbildung 7: Evakuierung in Abhangigkeit der Panikschwelle

Fur alle drei Kurven ist eine klare Tendenz erkennbar. Die Kurve sinkt zwar nicht monoton, jedoch scheint eine hohere Panikschwelle eindeutig mit einer niedrigeren Evakuierungszeit einherzugehen, unabhangig von k_D . Dies entspricht einem realistischen Verhalten: fallen die Personen schnell in Panik (niedrige Panikschwelle), so werden sie schnell irrational und verlassen den optimalen Weg zum Ziel. Ist diese Schwelle jedoch hoher, so weichen sie weniger von diesem Weg ab und die Evakuierung geht deutlich schneller.

4. Anwendungen der Simulation

4.1. Einfluss von Hindernissen

Nachdem wir nun die grundlegenden Parameter mitsamt der Mechanismen und ihren Auswirkungen auf die Evakuierungszeit betrachtet haben, wollen wir zu den anfangs beschriebenen Situationen zurückkehren. Zuerst wollen wir auf den in der Einführung (1) unter 2.) aufgeführten Punkt eingehen, indem wir dieselbe Ausgangssituation wie gerade nutzen und verschiedene Hindernisse vor den Ausgang setzen. Es ist davon auszugehen, dass sich dies ebenso (wie die Parameter in den vorherigen Untersuchungen) auf die Evakuierungszeit auswirken wird.

Wir werden 4 verschiedene Hindernisse mit der Ausgangssituation ohne Hindernis vergleichen, wobei wir hier die Personenanzahl und die Größe des Raums verändert haben um Platz für unser Hindernis zu schaffen. In der Abbildung 8 ist dies zu erkennen. Unsere 4 Hindernisse sind: ein Quadrat direkt vor dem Ausgang, ein Quadrat leicht versetzt, eine Säule (bzw. unsere Approximation durch Quadrate) sowie eine Pfeilspitze.

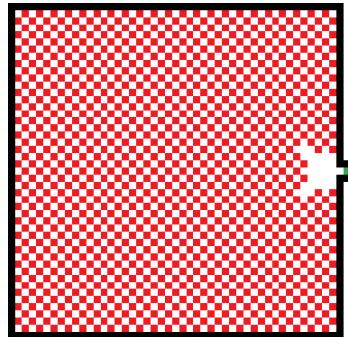


Abbildung 8: Ausgangssituation ohne Hinderniss

Intuitiv müsste die Evakuierungszeit größer sein mit einem Hindernisse, weil die Personen ja nun einen längeren Weg haben und an den Hindernissen vorbei müssen. Wir haben die gleichen Simulationen für alle 5 Situationen durchgeführt. Zuerst betrachten wir die Evakuierungszeit in Abhängigkeit von der Reibung, ohne D-Feld, mit folgenden Werten:

$$k_S = 5, k_D = 0$$

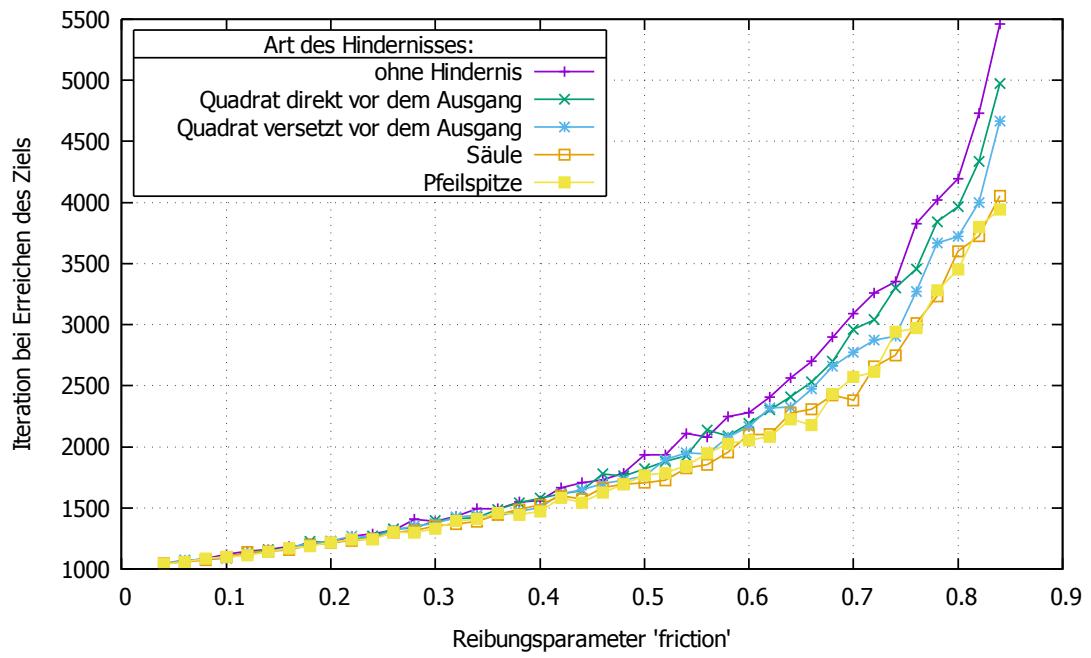
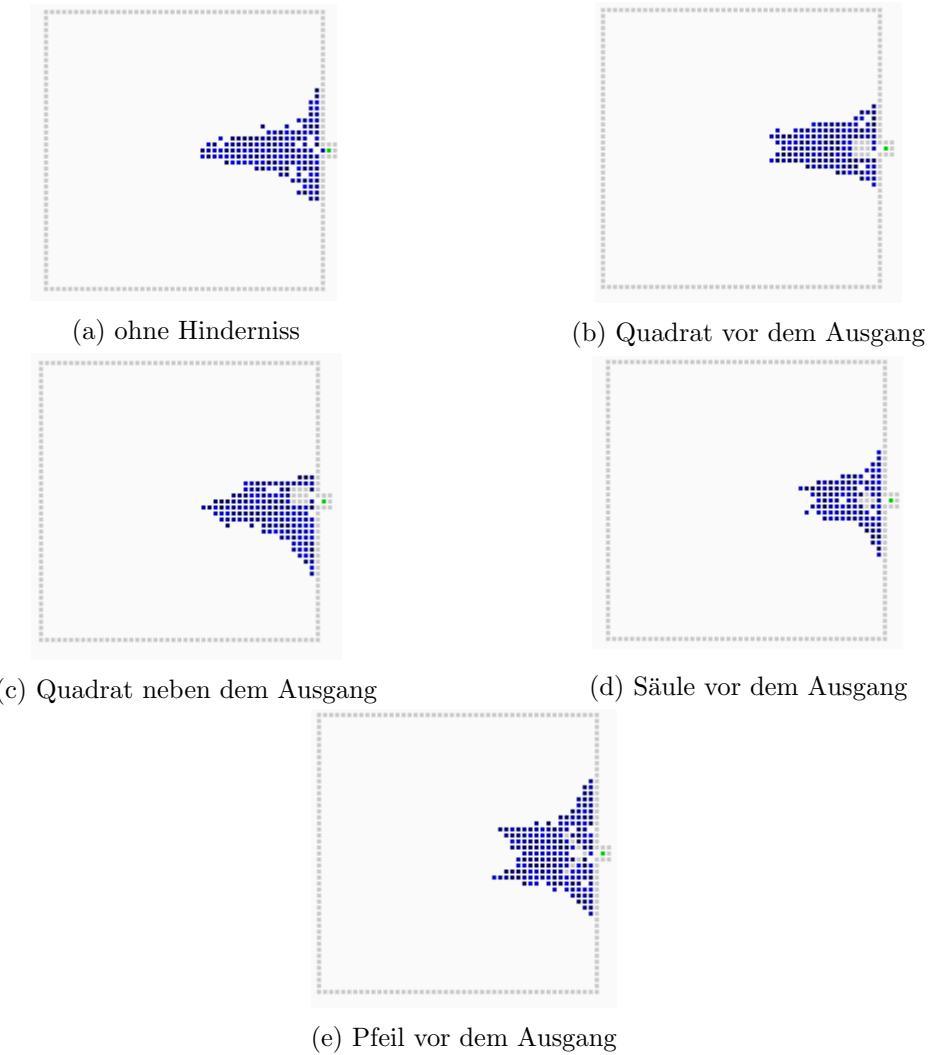


Abbildung 9: Evakuierung in Abhängigkeit von *friction*

Für den Bereich ab $friction = 0,4$ kriegen wir ein deutliches Bild: durch ein Hindernis vor dem Ausgang wird die Evakuierungszeit verringert, unabhängig von der Form des Hindernis. Für alle vier Hindernisse liegen die Werte kontinuierlich unter denen der Simulation ohne Hindernis, bis auf zwei Ausreißer des Quadrats vor dem Ausgang. Zur Erklärung schauen wir auf Standbilder aus der Simulation:



Auffällig ist, dass der Rückstau enorm ist und nahezu keine Verteilung in der Nähe vom Ausgang stattfindet, wenn kein Hindernis vor dem Eingang steht. Alle wollen auf direktem, geraden Wege zum Ziel und wählen deshalb einen sehr zentralen Weg. Es findet fast keine Aufgabelung vor dem Ziel statt.

Das statische Feld berücksichtigt Hindernisse, sodass der bevorzugte Weg automatisch um das Hindernis herum gewählt wird. Dadurch teilt sich die Masse besser auf, wenn ein Hindernis vor dem Ausgang steht und es entstehen weniger Konflikte. Nur bei Konflikten spielt eine höhere Reibung eine Rolle. Man erkennt diesen Effekt gut im Diagramm - erst bei hohen Reibungswerten wird der Unterschied klar erkennbar. Gibt es besonders viele Konflikte an der Spitze, so ist die Wahrscheinlichkeit groß dass vorne viele Personen stehen bleiben. Damit wird das gesamte Nachrücken verzögert. Bei den Hindernissen gibt es nur zwei Richtungen, von denen aus man auf das letzte Feld vor dem Ziel treten kann. Also gibt es auch hier weniger Konflikte, da sich ohne Hindernis häufig 3 Personen

um diesen Platz streiten.

Insgesamt bewirkt ein Hindernis durch die Aufteilung der Personenmasse, dass die durchschnittliche Distanz zum Ziel so gering wie möglich gehalten wird. Verteilen sich die Menschen auch in der Nähe vom Ziel, statt sich nur in einer Reihe zentral vor dem Ausgang aufzuhalten, so ist ein schnelleres Aufrücken möglich.

Zu Erkennen sind auch Unterschiede bei den einzelnen Hindernissen. Die Säule und die Pfeilspitze schneiden am Besten ab, wohingegen das Quadrat zentral vor dem Ausgang nur knapp besser abschneidet als die Simulation ohne Hindernis. Hier scheint die geometrische Form also schon eine Rolle zu spielen. Wenn man wieder auf die Teilchenebene zurückgeht, könnte man die Personen als Teilchenstrom interpretieren. Die Säule und die Pfeilspitze garantieren hier den besten Strom, das kantige Quadrat hingegen ist ein großer Widerstand und hindert die Teilchen somit mehr am „Vorbeifließen“.

Ebenso fällt auf, dass die asymmetrische Anordnung des Quadrats zu einer Verbesserung beträgt im Vergleich zum zentralen Quadrat. Dadurch gibt es zwar auf der einen Seite nur sehr wenige Leute, dafür auf der anderen jedoch eine deutliche größere Verteilung an der Wand. Dabei entstehen auf der oberen Seite relativ wenig Konflikte. Interessant für tiefere Untersuchungen wäre beispielsweise, wie sich symmetrische und asymmetrische Anordnung von einem oder mehreren Hindernissen auf die Evakuierungszeit auswirken. Nachdem wir im ersten Teil das D-Feld außer Betracht gelassen haben, wollen wir nun die Auswirkungen der Hindernisse mit veränderlichem D-Feld betrachten. Hierzu machen wir die gleichen Untersuchungen und setzen den Wert für $k_S = 0,7$ fest:

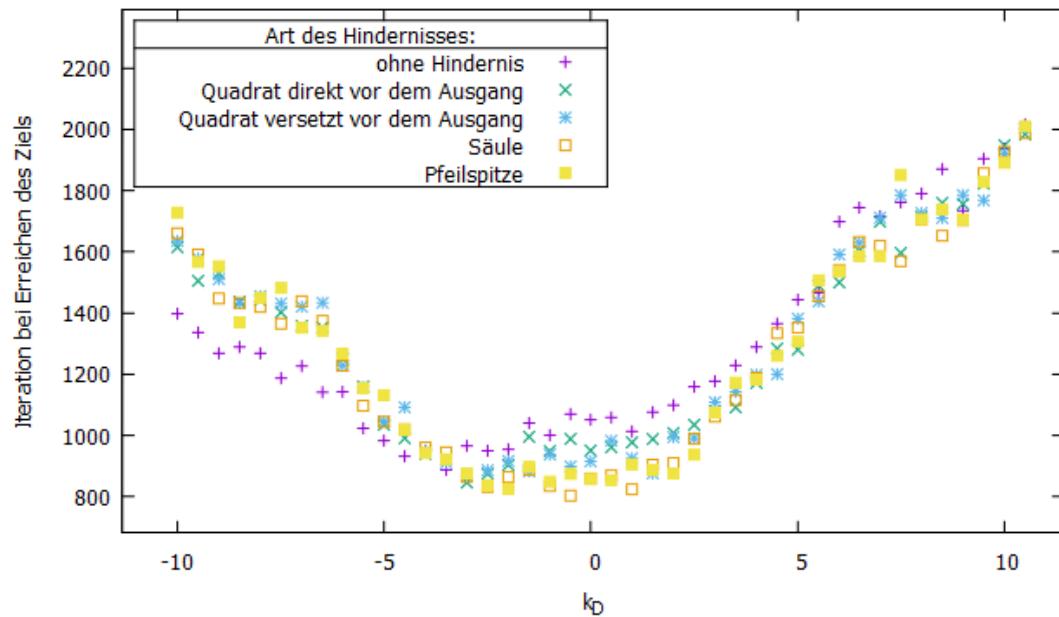


Abbildung 11: Evakuierung mit Hindernis in Abhängigkeit vom D-Feld

Der Effekt der Regulierung durch die Hindernisse ist auch hier zu erkennen. Ab $k_D = 8$ lassen sich keine konsistenten Aussagen mehr treffen, aber unterhalb dieser Schwelle ist der positive Effekt der Hindernisse auf die Evakuierungszeit deutlich. Im negativen Bereich für k_D gibt es jedoch einen umgekehrten Effekt: hier liegt die Kurve ohne Hindernis deutlich unter allen anderen. Negative Werte für k_D bedeutet, dass die Personen Wege vermeidet, die von Personen benutzt werden die in die gleiche Richtung laufen. Dies geht also mit einer repulsiven Wechselwirkung dieser Personen einher. Das Ergebnis im negativen Bereich stimmt nicht mit den Ergebnissen überein, die wir in der Bachelorarbeit¹¹ im Kapitel 5.1 Abb. 5.2 sehen können. Im positiven Teil sind sie jedoch vollständig übereinstimmend.

Die Ergebnisse lassen darauf schließen, dass es hier keine wesentlichen Unterschiede zwischen den einzelnen Hindernissen gibt. Mit zunehmendem Wert k_D wird der Unterschied zwischen ihnen immer kleiner, bis wir ungefähr ab k_D auch hier keine konsistenten Werte mehr bekommen. Um dies zu erklären, gehen wir wieder in unsere Teilchenanalogie über und betrachten den Effekt von k_D auf diesen Strom: durch einen höheren Wert k_D ist die Haftung zwischen Personen größer, da sie stärker von ihnen angezogen werden und daher Grüppchen gebildet werden. In der Hydrodynamik würde man von einer gewissen Viskosität sprechen. Dadurch kann der Strom nicht mehr mit derselben Geschwindigkeit an den Hindernissen vorbei, wobei es auch nahezu keine Rolle mehr spielt wie dieses Hindernis geometrisch beschaffen ist. Nichtsdestotrotz erkennen wir aber insgesamt auch hier in einer Situation mit Unordnung, dass die Evakuierung durch Hindernisse vor dem Ausgang unterstützt wird.

4.2. Flaschenhals-Effekt

Als nächstes wollen wir uns einer weiteren Situation widmen. Gibt es einen Durchgang, durch den Menschen in beide Richtungen wollen, ist ein oszillierendes Durchgangsverhalten zu beobachten. Dies kommt zustande durch den steigenden sozialen Druck auf die vorderste Person, der durch zu langes Warten entsteht. Abwechselnd baut dieser sich auf der jeweils wartenden Seite auf, bis er zu groß wird und die eigene Bewegung forciert wird. Derselbe Prozess beginnt dann auf der anderen Seite.

Um diese Situation darzustellen und mit unserem Programm nachzustellen, haben wir einen einfachen Raum mit zwei Hälften und einem kleinen Durchgang kreiert (s. Abb. 12). Die Zielauswahl wurde so manipuliert, dass die Personen nur das Ziel in der andere Hälfte des Raumes auswählen und somit zwangsweise durch den Durchgang müssen.

¹¹Christian Nitzsche; „Cellular automata modeling for pedestrian dynamics“

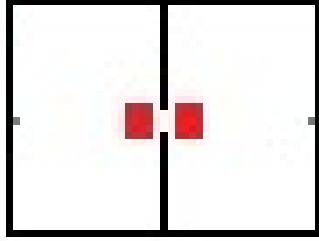


Abbildung 12: Ausgangssituation mit schmalem Durchgang

Wir haben diese Situation wieder unter zwei verschiedenen Bedingungen simuliert: einmal mit und einmal ohne D-Feld. Wir betrachten nun die Evakuierungszeit in Abhängigkeit von der Personenanzahl und erhalten mit den im Diagramm angegebenen Parametern die folgenden Ergebnisse:

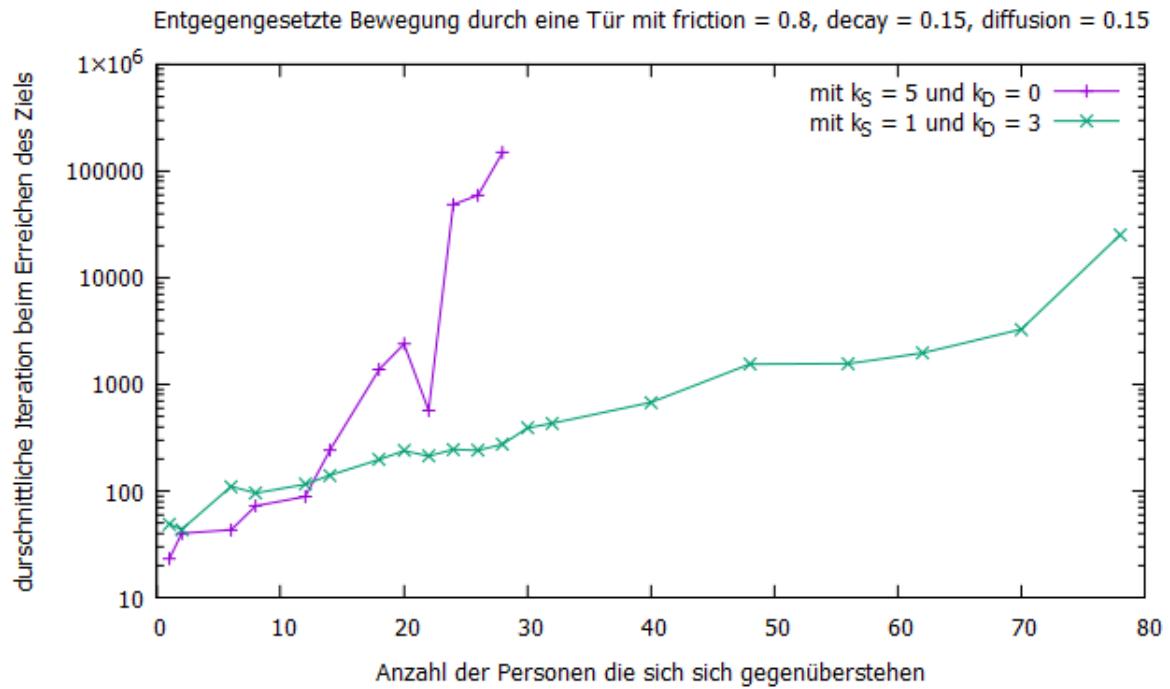


Abbildung 13: Durchgangszeit in Abhängigkeit von der Personenzahl

Es ist auffällig, dass sich die beiden Kurven genau einmal schneiden und dabei der Effekt genau umgedreht wird. Hier lässt sich die Grafik in zwei Teile trennen, nämlich vor und nach $n = 12$. Bei einer kleinen Personenanzahl wird das Durchgehen durch das D-Feld

beeinträchtigt und verlangsamt. Bei einer großen Anzahl von Personen sieht das wiederum anders aus: Hier scheint sich das D-Feld positiv auf das Durchgehen auszuwirken. Wir schauen uns wieder ein Standbild der Simulation an:



Abbildung 14: Standbild (ohne Einfluss vom D-Feld)

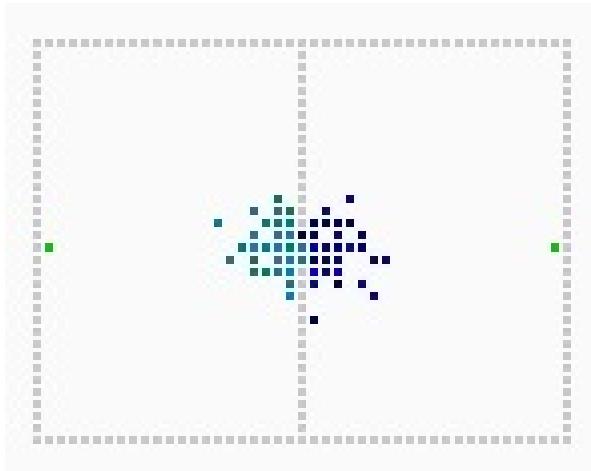


Abbildung 15: Standbild (mit Einfluss vom D-Feld)

Die Bilder entsprechen unserer Erwartung: durch das D-Feld entsteht Unordnung, die sich in einigen Lücken und abseits des Durchgangs stehenden Personen in Abb. 15 zeigt. In Abb. 14 zeigt sich jedoch, dass kaum Lücken vorhanden sind. Deshalb kann der Durchgang nicht passiert werden, da er vollständig blockiert wird und die Wahrscheinlichkeit, dass alle Personen zur Seite gehen, extrem klein ist.

Durch das D-Feld werden die Personen jedoch zum Teil von ihrem Weg abgebracht und somit entstehen Lücken für andere Personen. Weiterhin breitet sich das Feld nach und nach aus und kann größer werden, im Gegensatz zum statischen Feld.

Hierbei ist jedoch zu beachten, dass dieser Einfluss bei einer kleinen Personenanzahl lediglich verwirrend ist für die Personen, da sie ja eigentlich freie Bahn hätten. Durch den temporären Einfluss anderer Personen kann es jedoch sein, dass sie den kürzesten Weg verlassen. Deshalb ist der positive Einfluss des D-Felds erst ab einer größeren Personenanzahl vorhanden.

Leider konnten wir für größere Personenzahlen keine weiteren Daten sammeln, da die Simulationen sehr viele Iterationen erforderten. Außerdem war die Varianz so groß, dass wir uns auf diesen aussagekräftigen Bereich beschränken mussten.

4.3. Korridor

Als nächstes wollen wir den schon in mehreren Abschnitten erwähnten Korridor betrachten. In Abschnitt 2.1.2 hatten wir ein eigenes Potenzial eingeführt für den Korridor. Wir wollen untersuchen, ob wir den Effekt der Spurbildung nachweisen können. Dazu kommt es, wenn 2 verschiedene Personengruppen genau entgegengesetzt zueinander laufen. Dies minimiert Kollisionen und garantiert somit einen bestmöglichen Fluss.

Um den Effekt über einen längeren Zeitraum beobachten zu können, kreieren wir einen unendlich langen Korridor indem wir Personen, die bei ihrem Ziel angekommen sind wieder auf die andere Seite „beamten“. Dort beginnt für sie erneut der Weg zum Ziel auf der anderen Seite des Korridors. Unser Korridor sieht aus wie in 2.1.2 beschrieben.

Da wir Probleme bei der Darstellung der Spurbildung hatten, haben wir folgendes Verhalten programmiert: tritt ein Konflikt auf, so ist es wahrscheinlicher dass die Person danach in Bewegungsrichtung nach rechts ausweicht. Dies entspricht dem natürlichen Verhaltensmuster von Menschen, zumindest dort wo Rechtsverkehr herrscht. Dies haben wir implementiert, indem das D-Feld der Zelle rechts von der Bewegungsrichtung um 1 erhöht wird. Mit diesem kleinen Zusatz konnten wir die Spurbildung gut simulieren:

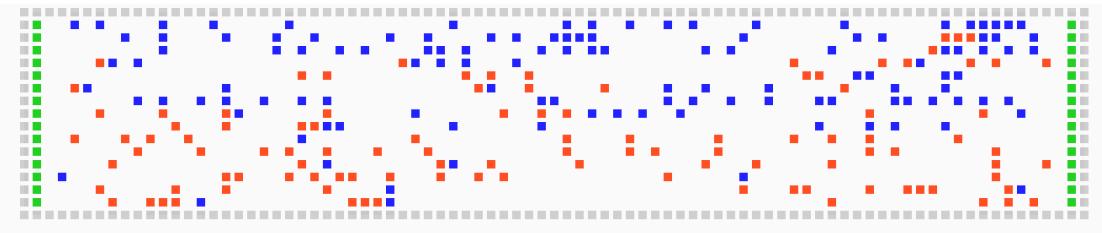


Abbildung 16: Standbild der Simulation

Die Personen die mit blau markiert sind gehen nach links, während die Personen in rot nach rechts gehen. Man kann optisch erkennen, dass hier eine räumliche Trennung stattgefunden hat. Auch quantitativ lässt sich dies erkennen: 97 blaue Personen befinden sich in der oberen Hälfte des Korridors, während sich dort lediglich 24 rote befinden. In der unteren Hälfte wiederum befinden sich bloß 17 blaue Personen und 76 rote. Die 16 Personen, die sich genau in der Mitte befinden, lassen wir außer Acht.

Wir wollen nun noch untersuchen, ob wir diesen Effekt noch verstärken können durch kleine Hindernisse in der Mitte des Ganges. Deshalb haben wir die Ausgangssituationen abgeändert und die gleiche Simulation noch einmal durchgeführt:

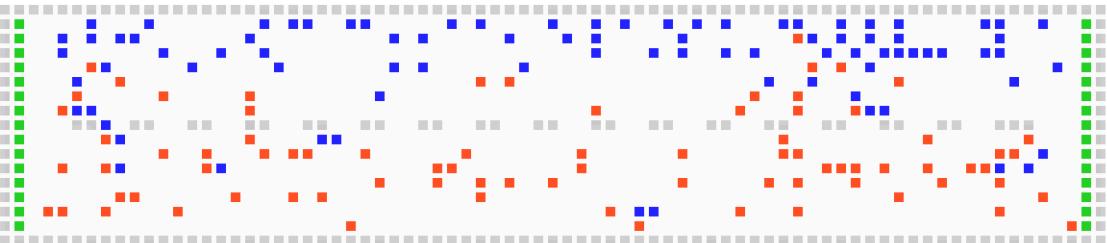


Abbildung 17: Standbild mit Hindernissen

In der oberen Hälfte sind nun 75 blaue Personen und 19 rote Personen zu finden. In der unteren hingegen befinden sich 61 rote und nur 10 blaue. Zur Überprüfung lässt sich der Anteil von Personen in der „falschen Spur“ berechnen. Insgesamt haben wir für die beiden Situationen einen Anteil von

$$19/155 = 0,176$$

bzw.

$$41/224 = 0,192$$

Wir konnten den Anteil der Personen in der „falschen Spur“ also insgesamt leicht senken durch die Hindernisse in der Mitte. Es ist anzunehmen, dass dieser Effekt in der Realität jedoch deutlich größer sein dürfte. Denn wie wir in Abschnitt 4.1 gesehen haben, ist die Separation durch Hindernisse durchaus bedeutend. Grundlegend konnten wir den Effekt allerdings sehr gut darstellen.

4.4. Weitere Situationen

Mit dem Programm ist es möglich nahezu beliebig schwierige Grundrisse für die Simulation auszuwählen. Die Grenzen des Programmes sind hier vor allem, dass es nur zweidimensionale Grundrisse aufnehmen kann, also theoretisch nur eine Etage möglich ist und dass bei zu großen Grundrissen die Simulationsgeschwindigkeit abnimmt. Indem man versucht ein mehrstöckiges Gebäude auf 2 Dimensionen zu projizieren kann man jedoch trotzdem mehrstöckige Gebäude simulieren. Wie hier beschrieben, haben wir dies für die ersten beiden Etagen des Physik-Instituts Potsdam ausprobiert. Der grobe Simulationsverlauf ist in den folgenden Abbildungen zu erkennen. Zur Simulation

wurden folgende Einstellungen verwendet:

$$\begin{aligned}k_S &= 2 \\k_D &= 2 \\ \text{decay parameter} &= 20 \\ \text{diffusion parameter} &= 15\end{aligned}$$

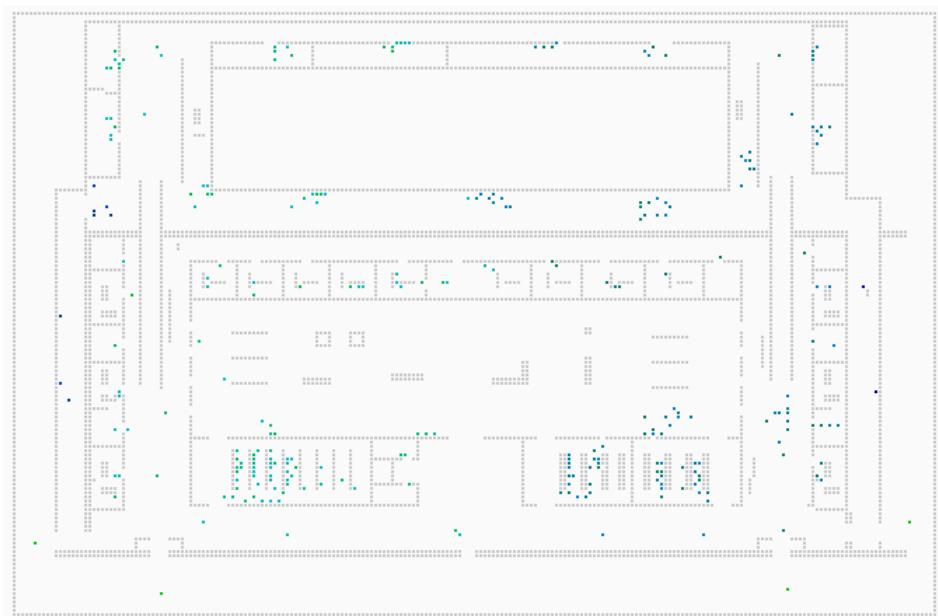


Abbildung 18: Ausgangssituation Golden Cage

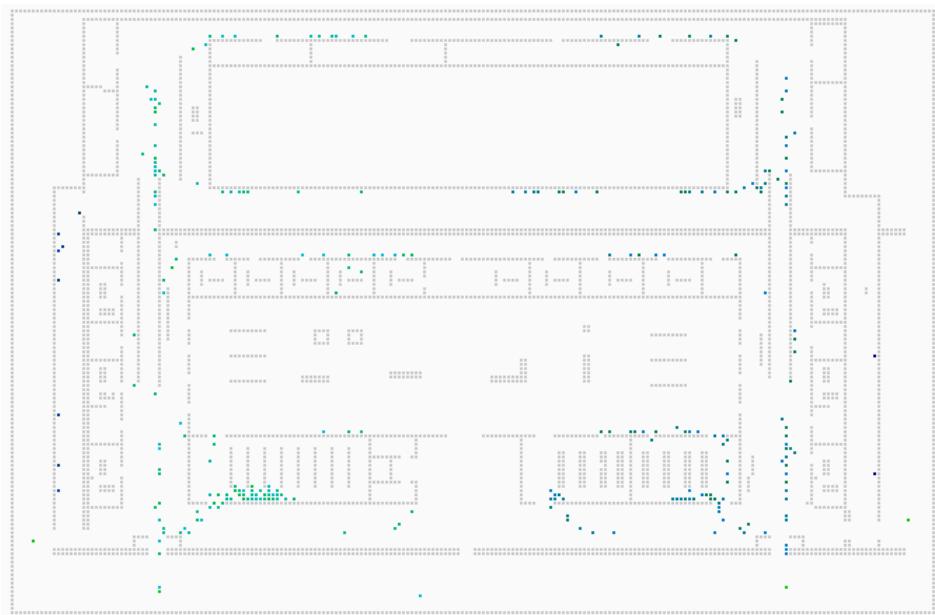


Abbildung 19: Simulationsverlauf Golden Cage

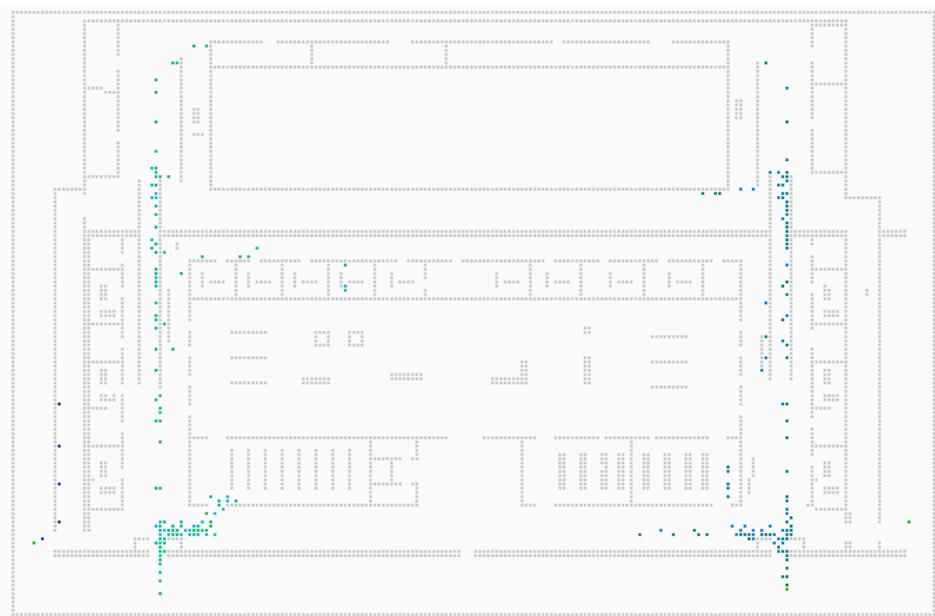


Abbildung 20: Simulationsverlauf Golden Cage

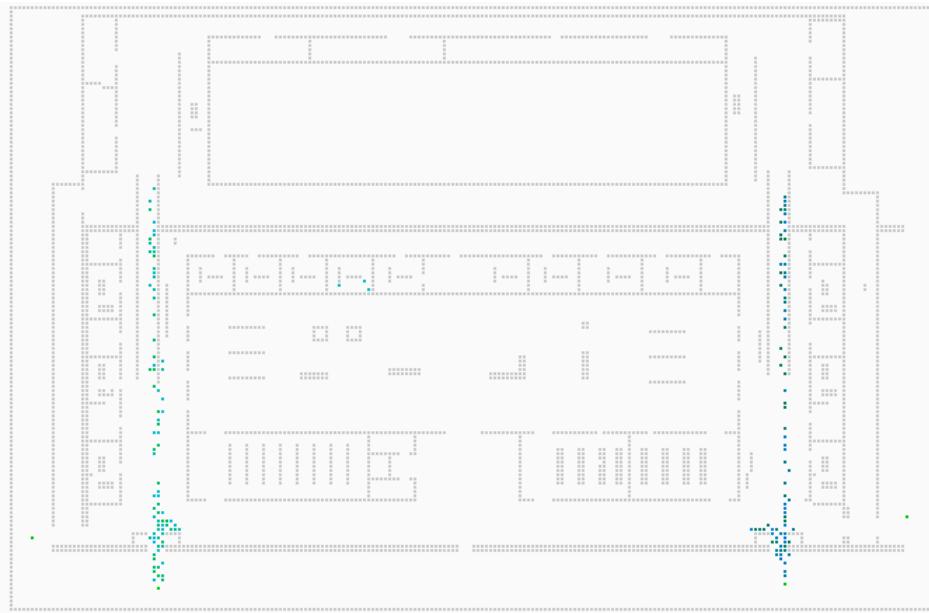


Abbildung 21: Simulationsverlauf Golden Cage

Hier kann man gut Hindernisse erkennen, die zu einem Stau der sich im Gebäude befindlichen Personen führen kann. Dies ist zum Beispiel in der Abbildung 19 beim Ausgang des großen Hörsaals, sowie bei der Abbildung 20 an den beiden Hauptausgängen. Bei einer genaueren Analyse bzw. mit einem genaueren Grundriss, könnte man mithilfe der Simulation solche Hindernisse erkennen und durch bauliche Maßnahmen die Evakuierungsgeschwindigkeit verbessern.

Andererseits erkennt man bei der Benutzung eines solch großen Grundrisses die Schwächen der Simulation. Lässt man zum Beispiel die Personen während der Evakuierung zu dem am weitesten entfernten Ausgang laufen und generiert dadurch Konflikte von Personen, die in einem Gang ein genau entgegengesetztes Ziel haben, so kann man hier unrealistisches Verhalten der Personen erkennen. Die folgenden drei Abbildungen zeigen den Verlauf einer Simulation, die genau den oben genannten Einstellungen folgt.

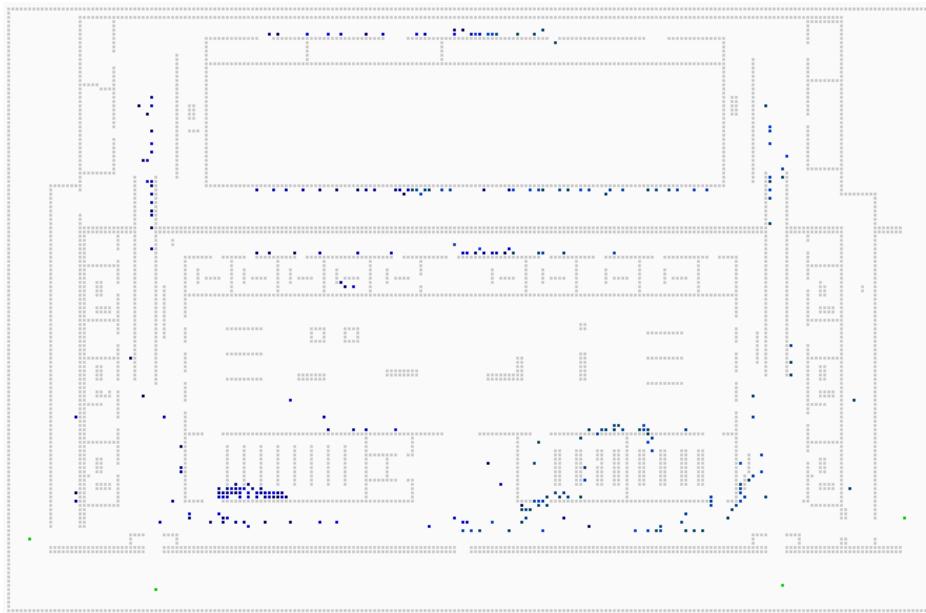


Abbildung 22: Personen laufen zum entferntesten Ausgang

Wie man in Abbildung 23 im Bereich des obersten und zweiten waagerechten Korridor erkennen kann, bewegen sich die Personen dicht an der Wand entlang. Da dies beide Personengruppen machen, die genau in die entgegengesetzte Richtung wollen, kommt es hier zu einem Konflikt. Dieser Konflikt wird aber im realen Leben dadurch verhindert, dass die gesamte Breite des Ganges ausgenutzt wird.

Grund für das unrealistische Verhalten ist hierbei die Methode wie das S-Feld erzeugt wird. Hierbei gibt es einen Gradienten zur oberen Grenze des Ganges. Die Lösung wäre, dass solche Gänge ähnlich wie in der Simulation der Korridore behandelt werden. Hierbei gibt es keinerlei Stärkegradienten und das S-Feld hängt nur von der x-Koordinate des Feldes ab (s. 2.1.2). Dies ist aber relativ schwierig zu implementieren, denn es müsste automatisch vom Programm erkannt werden, dass es sich um einen Korridor handelt und für diesen Bereich einen anderen Erzeugungsalgorithmus für das S-Feld benutzen.

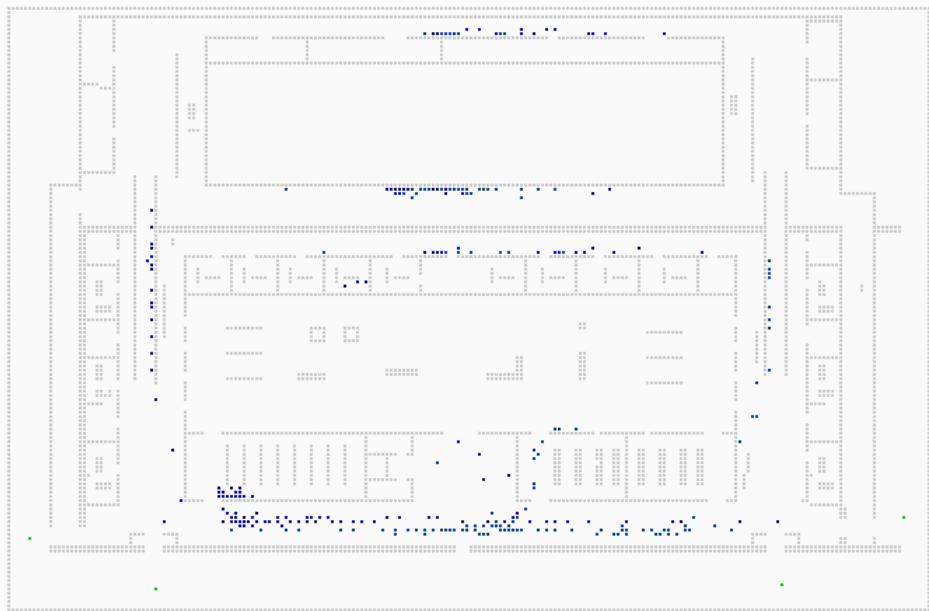


Abbildung 23: Problemzone Korridor (oben)

In der Abbildung 24 kann man erkennen, dass sich die Konfliktparteien trotzdem irgendwann trennen. Da es sich hierbei aber auch um einen Flaschenhals-Effekt handelt, ist diese Art der Konfliktlösung nur auf eine geringe Anzahl der teilnehmenenden Konfliktpartner beschränkt. Bei einer großen Anzahl von Personen, würde sich die Situation nicht mehr auflösen. Die Problematik ist im Abschnitt 4.2 näher beschrieben. Auch hierbei hilft wieder eine Verstärkung des D-Feldes, da dadurch mehr Lücken gelassen werden.

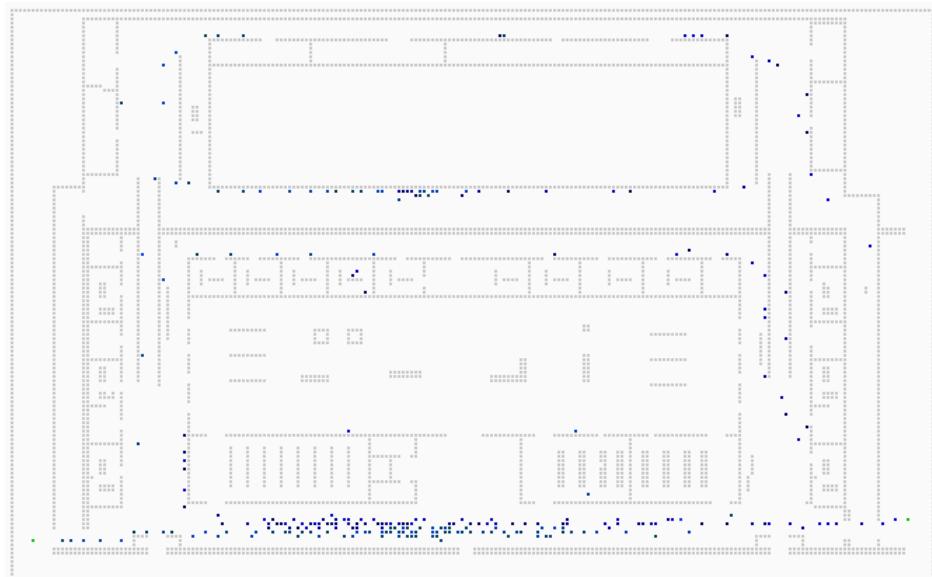


Abbildung 24: Auflösung der Konflikte

5. Probleme

Mit unserem Programm konnten wir insgesamt viele Effekte von Fußgängern reproduzieren. Nichtsdestotrotz haben sich Lücken in unserem Programm offenbart.

Wir können kein oszillierendes Durchgangsverhalten reproduzieren. Dafür fehlt uns eine von der Dauer des Stehenbleibens abhängige Kraft, die größer wird in Richtung des Ziels wenn die Wartezeit größer wird. Oder alternativ eine längere Reichweite der Interaktion mit anderen Personen. In unserer Simulation interagieren die Personen nur mit ihren unmittelbaren Nachbar über das D-Feld, beziehungsweise zum Teil auch darüber hinaus durch die *Diffusion*. Wir konnten das Problem nur über einen größeren Anteil an randomisierter Bewegung lösen.

Weiterhin haben wir uns dazu enstschlossen zur Reduzierung der Komplexität am Modell des zellulären Automaten weitestgehend festzuhalten und auf individuelle Unterschiede zwischen den Personen zu verzichten, obwohl es mit unserem Programm theoretisch einstellbar ist. Es ist anzunehmen, dass Bewegungsgeschwindigkeiten, Reibungswerte und auch die Gewichtung zwischen rationalem und emotionalem Bewusstsein unterschiedlich verteilt sind. Dies könnte man beispielsweise durch eine Gaußverteilung mathematisch darstellen und implementieren. Uns ist es nur möglich, relative Zielgeschwindigkeiten anzugeben, die dem Wert k_S entsprechen.

Um individuelle Geschwindigkeiten, also unterschiedlich große Schritte pro Zeitschritt, darstellen zu können, müsste man jedoch von unserem Ansatz abweichen und zu einem Agent-Based-Model übergehen. Dies würde jede Person einzeln modellieren und würde eine individuelle und komplexere Betrachtung zulassen. In der Bachelorarbeit¹² wird auf Ansätze hingewiesen, zelluläre Automaten auf Multispeed Varianten zu überführen.

Weiterhin ist es nicht möglich, Personenzüge zu simulieren. Damit sind Personen gemeint, die direkt hintereinander stehen und sich in dieselbe Richtung bewegen. Ist ein Nachbarfeld zum Zeitpunkt t besetzt, so wird dieses Feld als Bewegungsziel ausgeschlossen. Wird das Nachbarfeld nun durch einen Schritt der betreffenden Person leer, wird es zum Zeitpunkt $t+1$ also trotzdem nicht besetzt. Hierfür müssten entweder eine Priorisierung von Personen erfolgen oder man müsste zum sequentiellen Update übergehen, welches alle Personen nacheinander betrachtet.

¹²Christian Nitzsche; „Cellular automata modeling for pedestrian dynamics“

6. Fazit und Ausblick

Unser Ziel war es, Fußgänger dynamiken möglichst realistisch abzubilden. Um real anwendbare Daten für Räume oder Veranstaltungsorte zu erhalten, ist eine exakte Kalibrierung der Parameter anhand einem reellen Verhalten in einem ähnlichen Aufbau vonnöten.

Wir können festhalten, dass wir es geschafft haben eine größtenteils realistische Abbildung zu erstellen. Viele der zu untersuchenden Effekte konnten wir mit unserem Programm nachstellen.

Die emotionale Komponente bei menschlichen Bewegungen haben wir mit dem dynamischen Feld dargestellt, welches die Evakuierungszeit erhöht. Eine extreme Ausprägung hiervon ist die Panik, bei der dieses Verhalten verstärkt wird. Weitere soziologische Phänomene (Höflichkeit, Überforderung usw.) haben wir über den ebenfalls verlangsamenden Faktor *Friction* dargestellt.

Den positiven Einfluss von Hindernissen konnten wir ebenso darstellen wie die Abhängigkeit von der geometrischen Beschaffenheit. Wir erhalten dabei ziemlich kongruente Ergebnisse zu denen in Christian Nitzsche: „Cellular automata modeling for pedestrian dynamics“.

Weiterhin konnten wir zeigen, dass sich in einem horizontalen Korridor parallele Spuren bilden. Eine Spurstabilisierung durch in der Mitte platzierte Hindernisse konnten wir nur teilweise nachweisen.

Ebenfalls Schwierigkeiten hatten wir mit dem oszillierenden Durchgangsverhalten. Unsere Lösung beruhte ausschließlich auf einer zufälligeren Bewegung durch Verstärkung des D-Felds.

Um diese und andere Phänomene ausführlicher zu analysieren, ist die individualisierte Betrachtung der Personen besonders wichtig. Hier könnte man bspw. individuelle Geschwindigkeiten darstellen, aber ebenso Gruppenzugehörigkeiten darstellen durch eine Viskosität von einzelnen Personen zueinander. Weiterhin lassen sich natürlich unterschiedliche Personendichten in den unterschiedlichsten Aufbauten mit verschiedenen Hindernissen betrachten. In dem uns gesteckten Rahmen hat unsere Umsetzung jedoch gut funktioniert und die meisten vorgegebenen Situationen gut abgebildet.

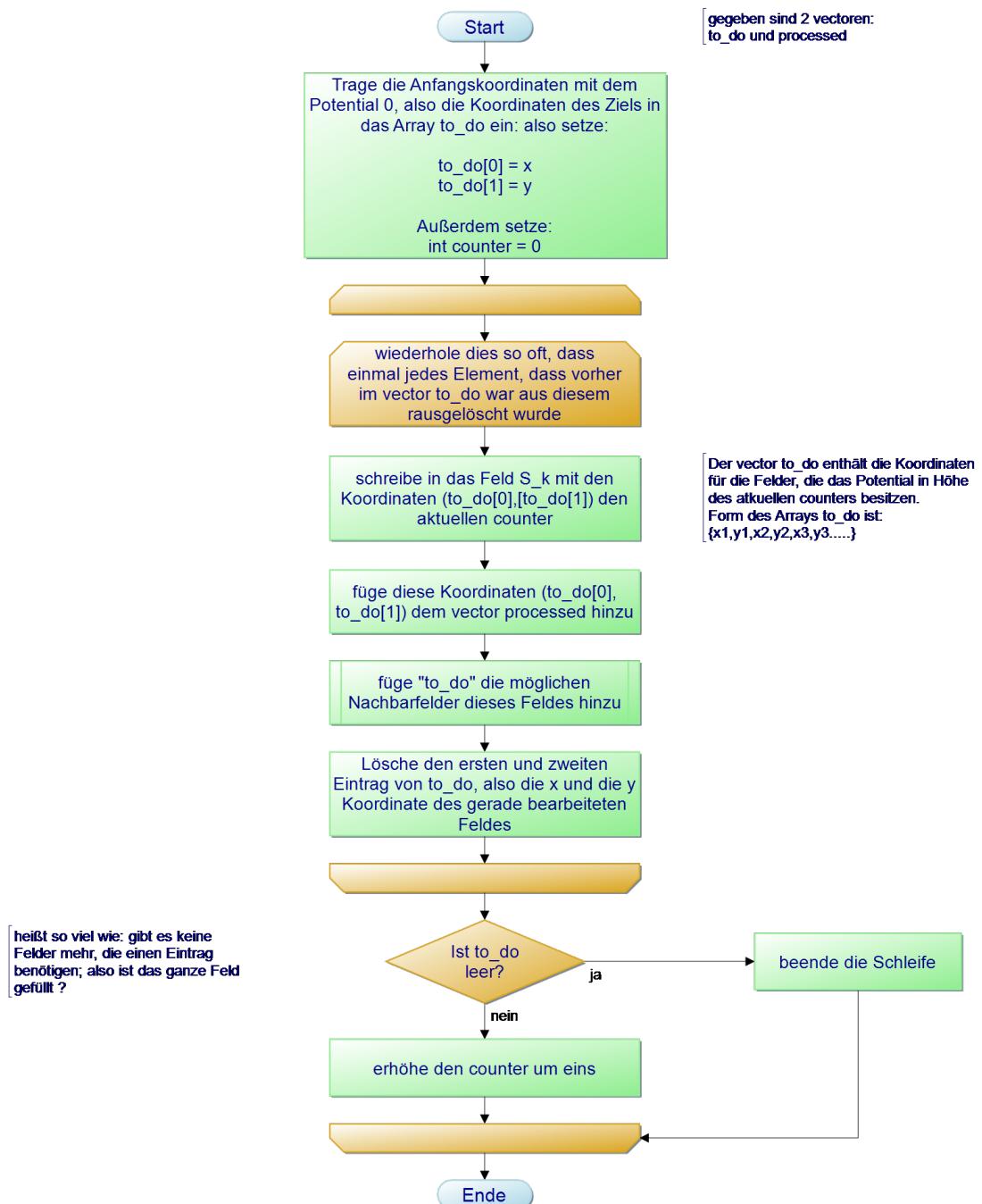
7. Quellen

- DIRK HELBING, ANDERS JOHANSSON: Pedestrian, Crowd and Evacuation Dynamics
- C.BURSTEDDE, K.KLAUCK, A.SCHADSCHNEIDER, J.ZITTARTZ: Simulation of pedestrian dynamics using a two-dimensional cellular automaton
- CHRISTIAN NITZSCHE: Cellular automata modeling for pedestrian dynamics

A. Programablaufpläne

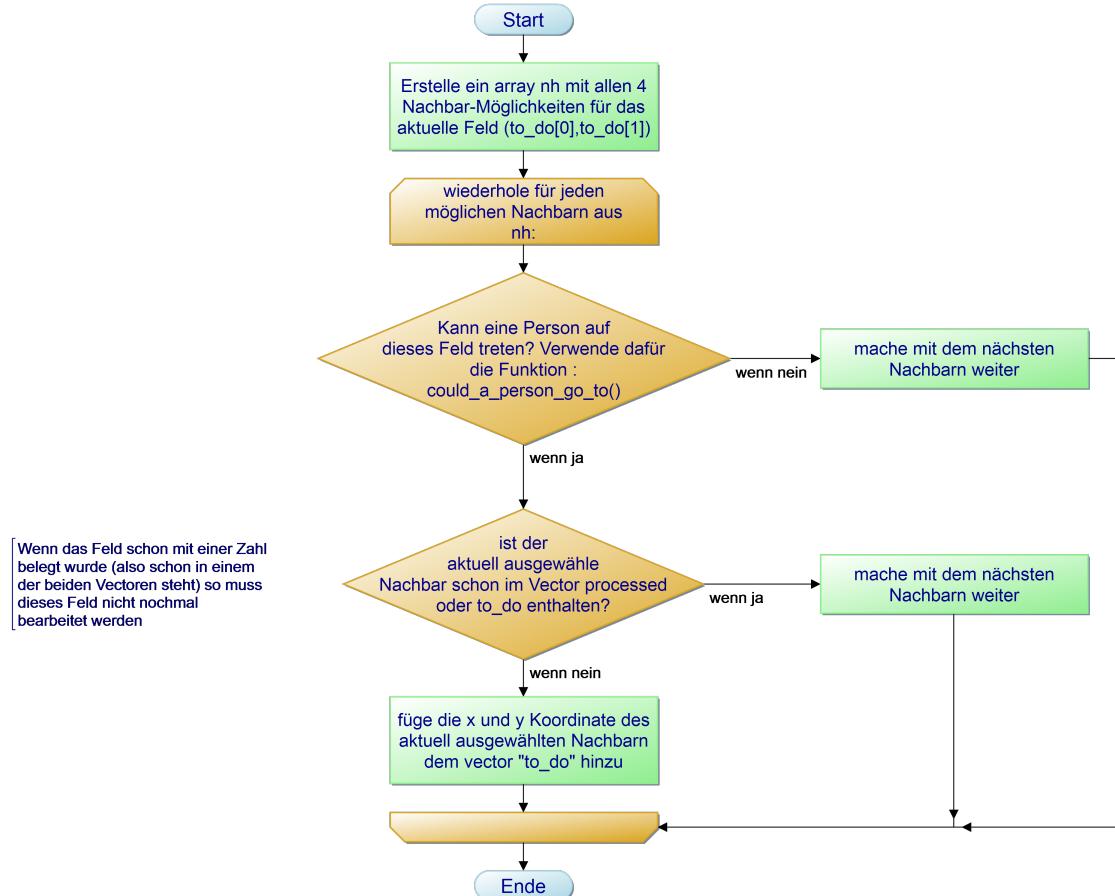
A.1. Erzeugen des statischen Feldes

Erzeugt das statische Feld S_k des Ziels mit der Nummer k

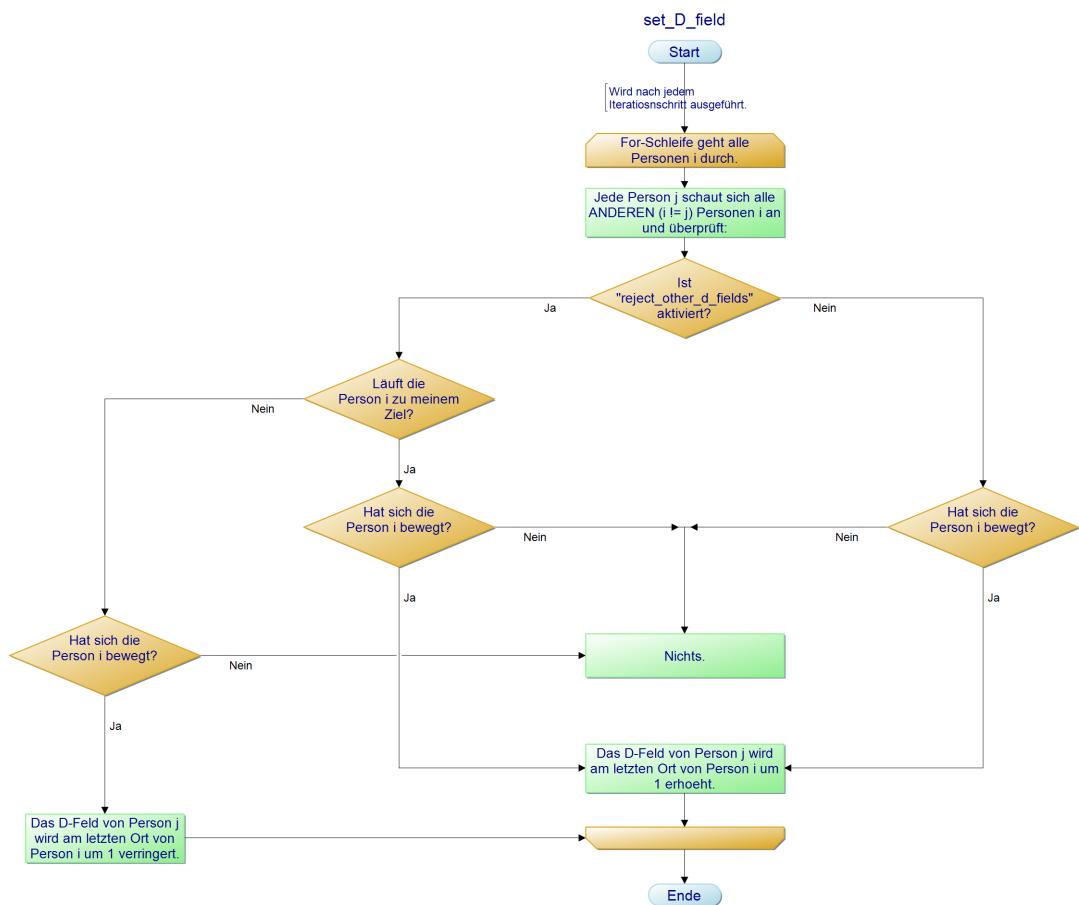


A.2. Ablaufen aller Nachbarfelder

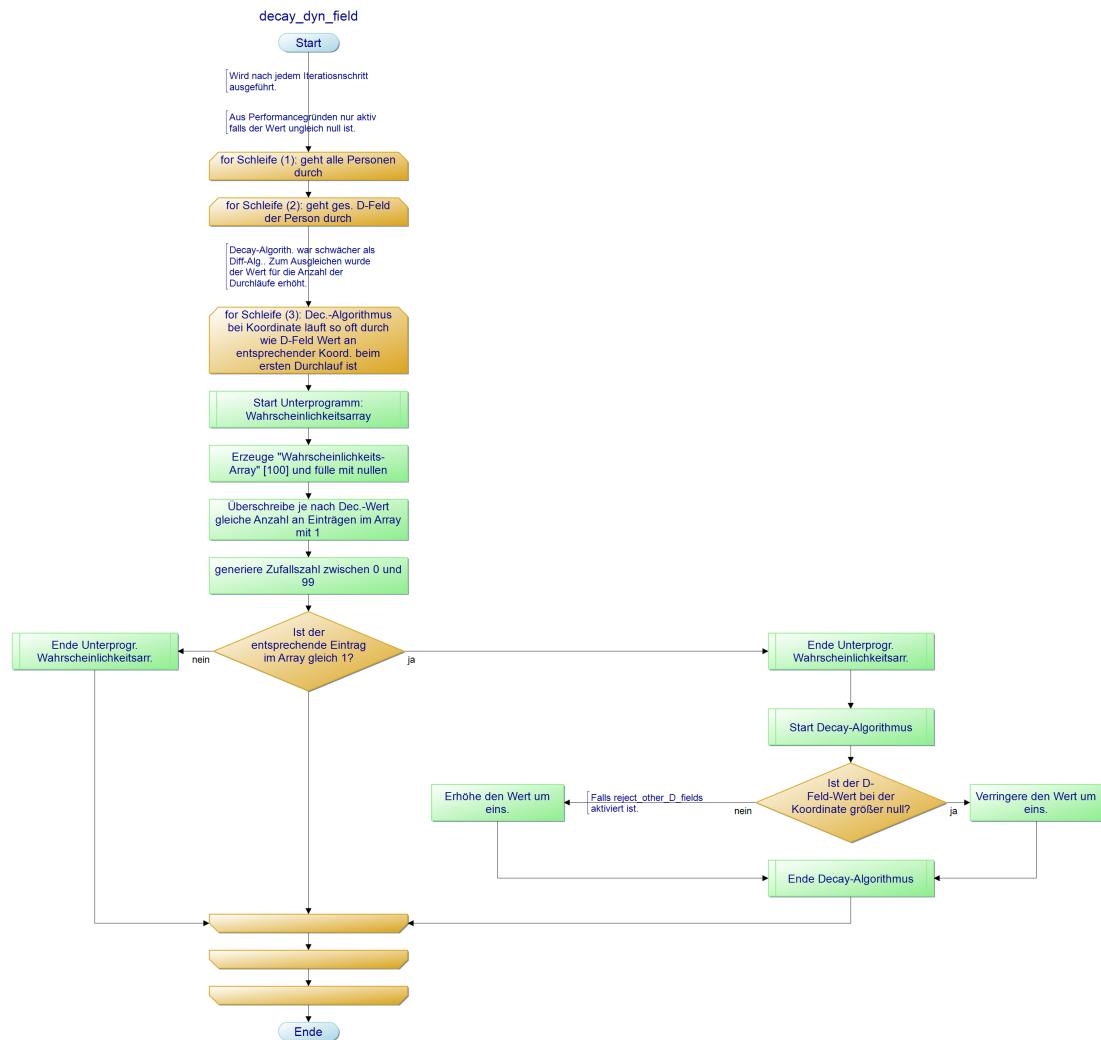
füge "to_do" die möglichen Nachbarfelder dieses Feldes hinzu



A.3. Erzeugen des dynamischen Feldes

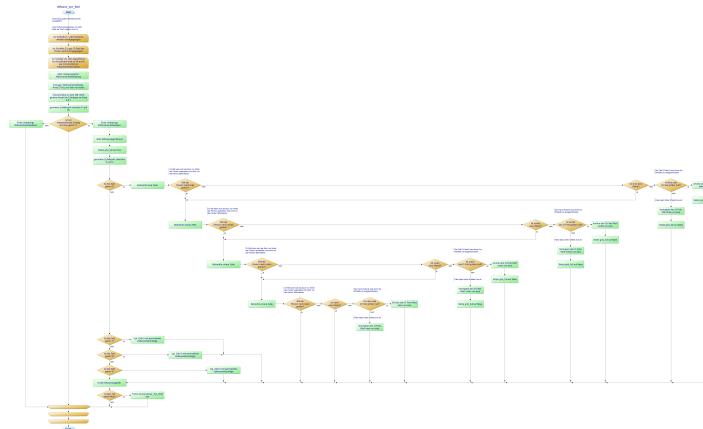


A.4. Decay des dynamischen Feldes

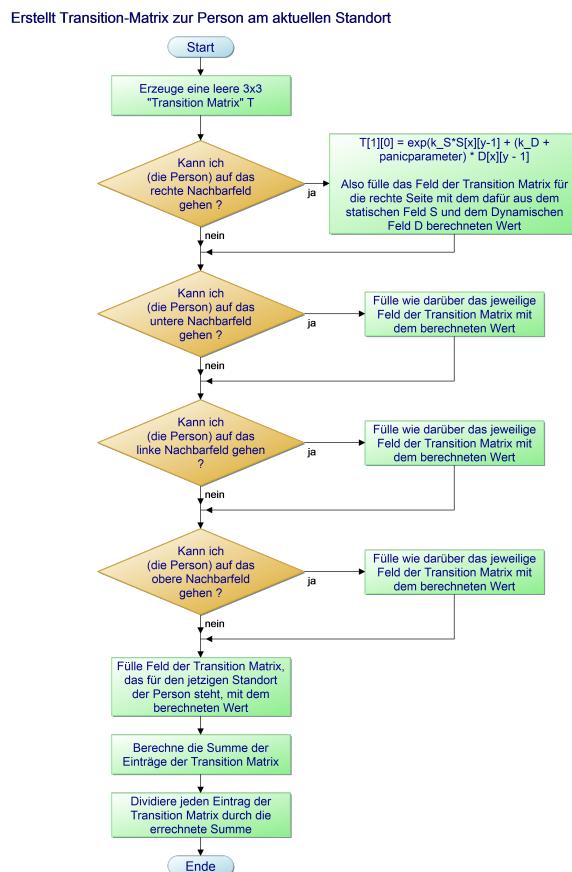


A.5. Diffusion des dynamischen Feldes

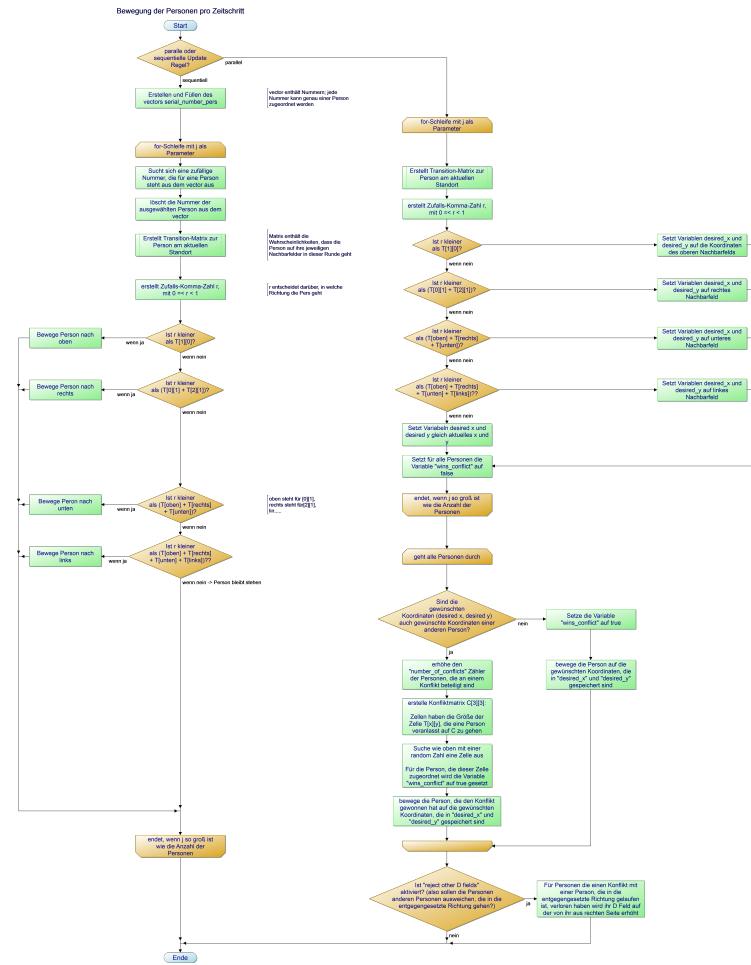
Durch Vergrößern kann diese etwas zu groß geratene Abbildung lesbar gemacht werden.



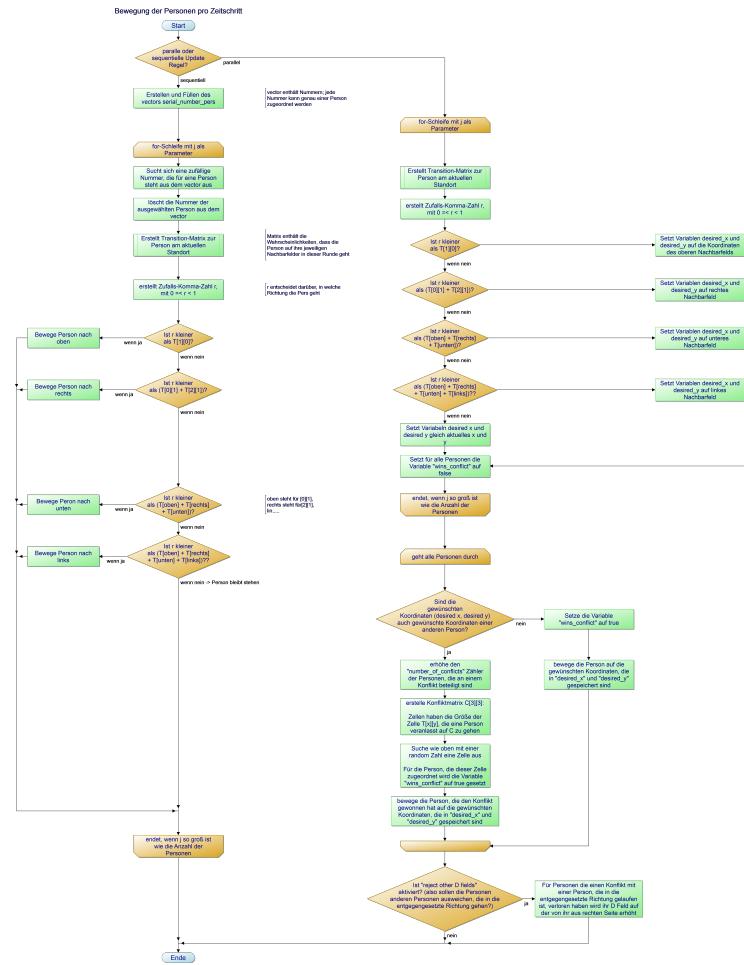
A.6. Erstellen der Transition Matrix



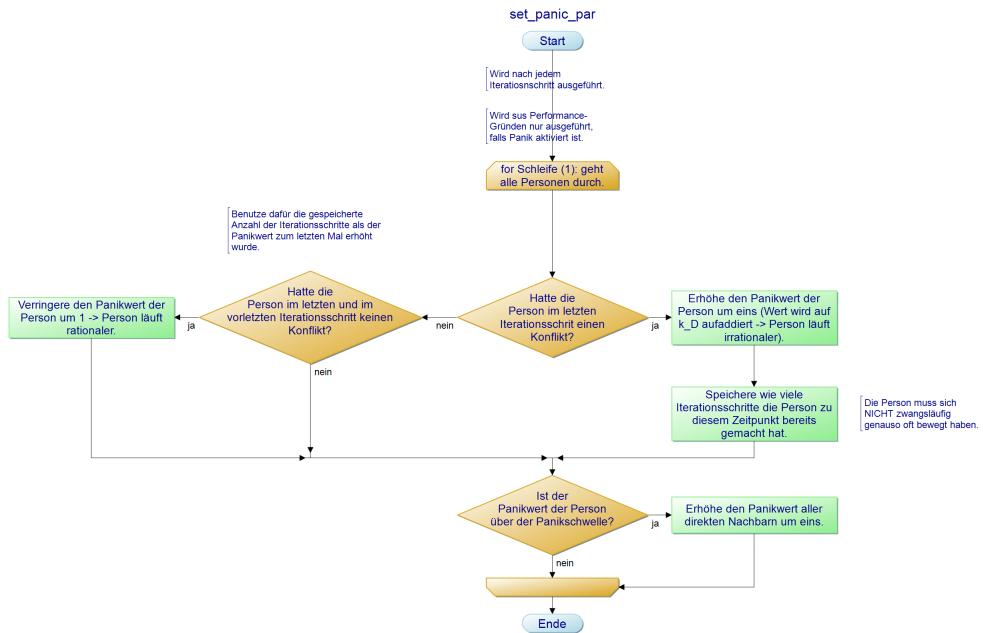
A.7. Bewegungsentscheidung einer Person



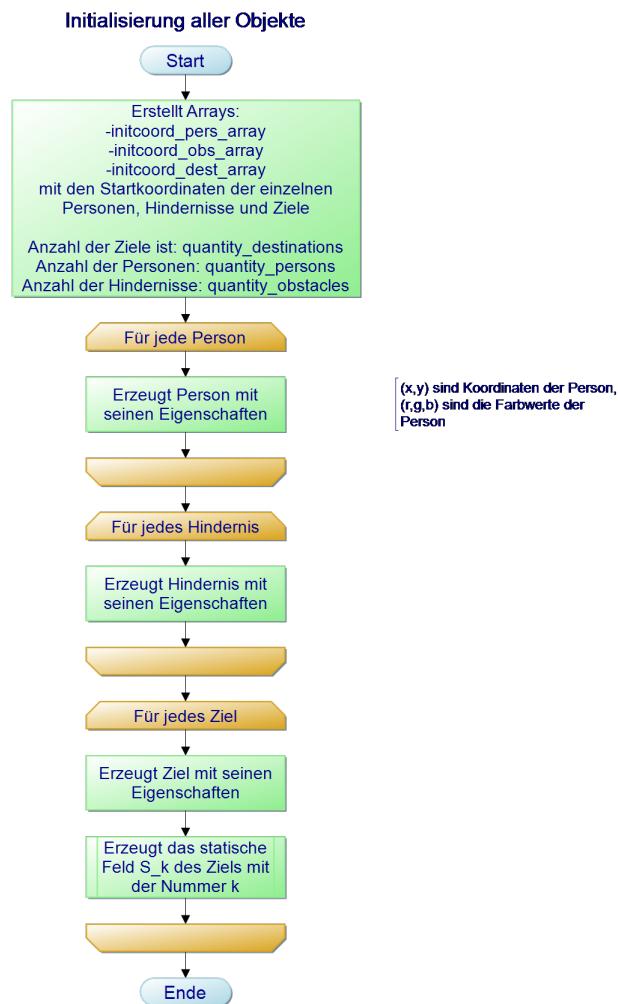
A.8. Bewegung der Personen



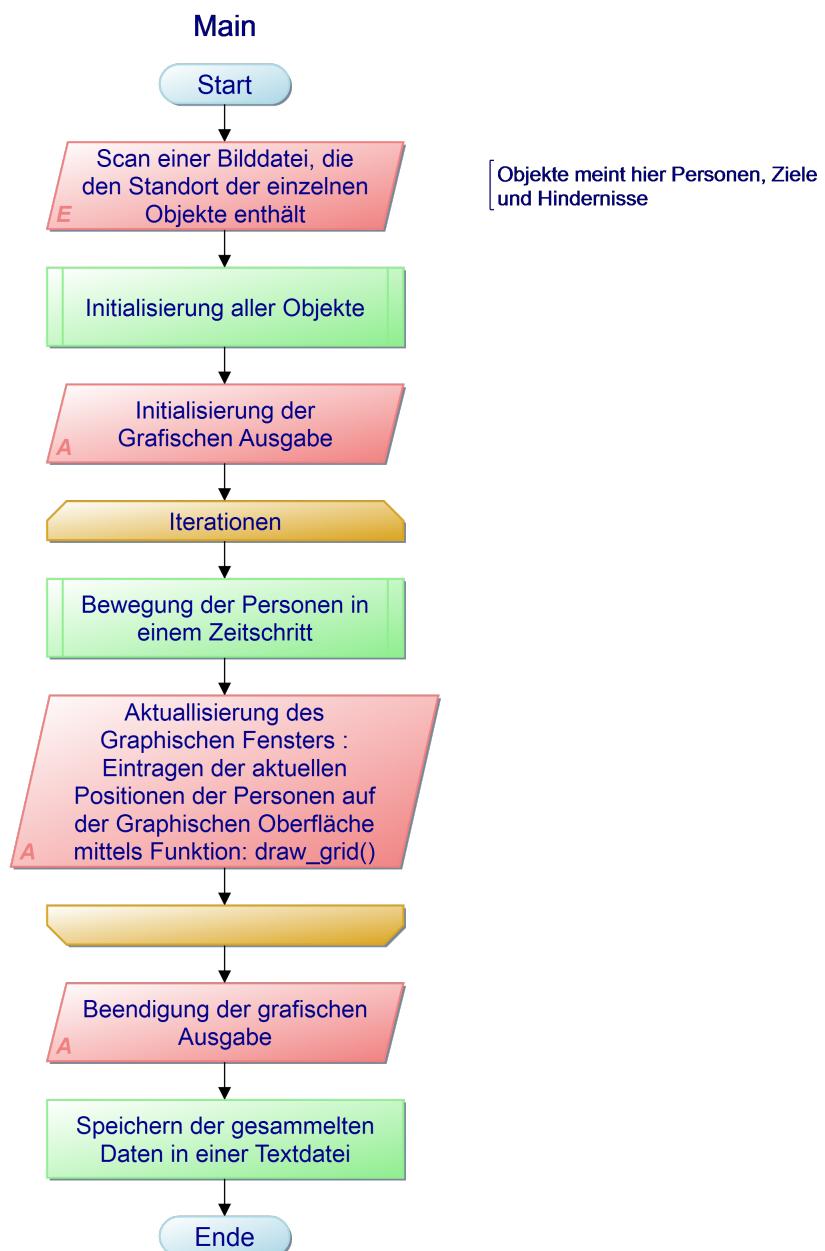
A.9. Panikschwelle



A.10. Initialisierung aller Objekte



A.11. Funktion der Hauptklasse



B. Quellcode

B.1. initial situation.cpp

```
1 // FEHLER BEHEBEN : T BESITT KEINEN WERT FÜRS STEHEN BLEIBEN; DA PERS
2 #ifndef initial_situation
3 #define initial_situation
4 #include <SDL.h>
5 #include <SDL_image.h>
6 #include <vector>
7 #include <string>
8
9 using namespace std;
10
11 /*
12 Erklärung der Parameter:
13 grid_height: Gibt die Höhe des Feldes wieder, in dem die Personen simuliert werden
14 grid_width: Gibt die Breite des Feldes wieder, in dem die Personen simuliert werden
15 max_number_of_iterations: Gibt die maximale Anzahl der Simulationsschritte an, die durchlaufen werden
16 iteration_break_condition: Gibt an ob die Simulation auch schon vorher gestoppt werden soll, wenn alle
Personen ihre Ziele erreicht haben
17 plant_layout: Gibt den Namen des Bildes an, welches den Anfangssituation der Hindernisse, Personen und
Ziele darstellt. Dieses Bild wird im Laufe der Simulation geladen
18 movement_update: Gibt an ob sich die Personen parallel oder hintereinander Bewegen sollen. Verschiedene
Effekte sind nur mit der parallelen Methode möglich (z.B. das Ausbrechen von Panik)
19 grafic_delay: Gibt an wie schnell die Iterationsschritte abgearbeitet werden [in ms]
20 decay_param: legt fest, wie stark das D Feld nach jeder Iteration verkleinert wird
21 diffusion_param: legt fest, wie schnell sich das D Feld nach jeder Iteration ausbreitet
22 panik_aktiviert: legt fest, ob bei der Simulation die Panik der Personen einbezogen wird
23 panik_schwelle: legt die Schwelle an Konflikten fest, ab der die Personen in Panik verfallen
24 corridor_conditions: muss aktiviert sein, damit das statische Feld im Falle, dass ein Korridor sammuliert
wird richtig gebildet wird
25 reject_other_D_fields: wenn dies aktiviert ist stoßen sich Personen, die in unterschiedliche Richtungen
laufen voneinander ab
26 unite_destinations_if_possible: legt fest ob Ziele die beieinander liegen die selben Wissenswerte w_S
erhalten
27 take_closest_exit: ist dies aktiviert, so wird w_S in jeder Iteration so verändert, dass das nächste Ziel
angesteuert wird
28
29 Wie setze ich die Variablen richtig?
30 Schritt:
31 1 - plant_layout wählen; Ist die Datei nicht vorhanden, so wird eine Fehlermeldung beim Ausführen
angezeigt.
32 2 - grid_height und grid_width an die Situation anpassen; Damit die Simulation richtig funktioniert muss
der gesamme Grundriss im von der Höhe und der Breite gesetzten Rechteck liegen
33
34 Die restlichen Optionen können nach Belieben eingestellt werden und werden zu keinem Fehler in der
Simulation führen.
35 */
36
37
38 const static int grid_height = 150;
39 const static int grid_width = 270;
40
41
42 static int max_number_of_iterations = 100000;
43 static bool iteration_break_condition = true; //kann das Program auch vorher schon abbrechen(wenn alle
Personen im Ziel sind)?
44
45 static const char plant_layout[] = "golden_cage_neu.bmp"; //Name des Gebäudeplans
46 static const char movement_update = 'p'; //'s' - sequential, 'p' - parallel
47 //BEIM PARALLELEN NOCHMAL NACHSCHAUEN: C[][] WIRD WIRKLICH RICHTIG GEWÄHLT ?? was hat es mit den eisen in
der Matrix zu tun?
48
49 static int grafic_delay = 0; // Je höher, desto langsamer aktualisiert sich die grafische Anzeige
50
51 static int decay_param = 20; //Zerfallsparameter fürs dynamische Feld [0,100]
52 static int diffusion_param = 15; //Verteilungsparameter fürs dynamische Feld [0,100] ERZEUGT FEHLER BEIM
AUSFÜHREN!
53
```

```

54 static int panik_aktiviert = false;
55 static int panik_schwelle = 10000; //ab welcher Anzahl von konflikten geraet jmd in panik
56
57 /*
58 Veränderungen am Ablauf des Programms, wenn "reject_other_D_fields" aktiviert ist:
59 - die Personen erhalten nur für exakt einen Ausgang den Wissensstand 1 (Ueberschreibung der Wissensstände
in Funktion set_model_parameter())
60      -> jede Person kennt also nur ein Ziel. Die Nummer dieses Ziels wird in der Variable
"numb_selected_dest" in der Personenklasse gespeichert
61 - Das D Feld von einer Person wird von anderen Personen nur erhöht, wenn sich diese in die Richtung
bewegen, in die das statische Feld der Person zeigt
62 */
63 static bool corridor_conditions = false; //Korridor muss waagerecht liegen; aktiviert automatisch
unite_destinations_if_possible
64 static bool reject_other_D_fields = true; //(noch nicht eingebaut) Ist für die Simulation für den Korridor
nötig, bei dem die Menschen mit unterschiedlichen Zielen das D Feld der Menschen mit einem anderen Ziel
abstoßend finden
65 static bool unite_destinations_if_possible = false; //(nur möglich wenn reject_other_D_fields aktiv ist)
Vereinigt Ziele die genau nebeneinanderliegen zu einem Ziel (w_S wird kopiert)
66 static string take_which_exit = "far"; // wenn "far" oder "near": w_S wird so verändert, dass jede Person
den Ausgang nimmt, der für sie am weitesten/nahesten ist. Bei "default" wird nichts ueberschrieben
67 /*
68 Erklärung zur Benutzung des Analysedurchlaufs:
69 Wenn die Daten des Simulationslaufs gespeichert werden sollen, so muss "execute" aktiviert sein.
70
71 Wenn die Parameter der Simulation so verändert werden sollen, dass die unten aufgeführten Werte von jedem
Objekt angenommen werden,
72 so muss "execute" aktiviert sein und der jeweilige Parameter muss positiv gewählt sein. Alle so gewählten
Parameter werden immer in allen Objekten verändert.
73 Zuweisungen für einzelne Objekte sind also nicht möglich.
74
75 Wenn das Programm über die Shell aufgerufen werden soll, bzw. der Aufruf durch die vorher programmierte
batch Datei(auf einem Windows Rechner)
76 stammt, so muss der Parameter foreign_call aktiviert sein. Die angegebenen Parameter werden dann ignoriert
und die Eingabe der Parameter erfolgt über
77 die Batch Datei, bzw. den Aufruf.
78 Die Parameter bei einem solchen Aufruf sind der Reihe nach: k_S, k_D, w_S, friction.
79
80 Ist fälschlicherweise der Parameter "foreign_call" aktiviert wird das Programm abstürzen.
81
82
83
84 Erklärung der Effekte der einzelnen Parameter auf die Bewegung einer Person:
85 k_S: Einfluss des statischen Feldes auf die Bewegungen der Personenu
86 k_D: Einfluss des dynamischen Feldes auf die Bewegungen der Personen
87 w_S: Wissensstand der Personen über die einzelnen Ausgänge (wenn der Einfluss vom statischen Feld nicht
beeinflusst werden soll, sollte w_S = 1 gewählt sein !!!
88 friction: Gibt die Wahrscheinlichkeit an, dass sich eine Person in einem Iterationsschritt nicht bewegt,
obwohl sie es dürfte
89 alpha:
90 delta:
91 */
92
93
94 struct analysis_run{
95     bool execute = true; //wenn true: Werte in dieser Strukturen werden dann an die Objekte übergeben und
die Abfrage an den Benutzer entfallen
96     bool foreign_call = false; //experimentell; Werte werden mit der Konsole hinzugefügt, dies kann für die
Analyse benutzt werden
97     // wird hier ein negativer eintrag gewählt, so wird dieser Parameter nicht gesetzt
98
99     double k_S = 2; //Einfluss von s auf die Bewegung der Personen
100    double k_D = 2; //Einfluss von D auf die Bewegung der Personen
101    double w_S = -1; //Wissen der Personen über die Ausgänge (zufällig im default)
102    double friction = 0.2; //zufällig im default
103

```

```
104  };
```

```
105
```

```
106 #endif
```

```
107
```

```
108
```

```
109
```

```
110
```

B.2. main s.cpp

```
1 #include <SDL.h>
2 #include <SDL_image.h>
3 #include <stdio.h>
4 #include <iostream>
5 #include <vector>
6 #include <string>
7 #include "klassen.cpp"
8 #include <fstream>
9 #include <time.h>
10
11
12 using namespace std;
13
14
15 //##### initial situation
16 #include "initial_situation.cpp"
17 //##### initial situation
18
19
20
21
22
23 ##### Grundriss einlesen
24 Uint32 getpixel_function(SDL_Surface *surface, int x, int y){ //Quelle:
http://sdl.beuc.net/sdl.wiki/Pixel\_Access; unter getpixel in SDL Paket enthalten, Liest Farbe eines Pixels aus
25
26
27     int bpp = surface->format->BytesPerPixel;
28     /* Here p is the address to the pixel we want to retrieve */
29     Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp;
30
31     switch(bpp) {
32     case 1:
33         return *p;
34         break;
35
36     case 2:
37         return *(Uint16 *)p;
38         break;
39
40     case 3:
41         if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
42             return p[0] << 16 | p[1] << 8 | p[2];
43         else
44             return p[0] | p[1] << 8 | p[2] << 16;
45         break;
46
47     case 4:
48         return *(Uint32 *)p;
49         break;
50
51     default:
52         return 0;      /* shouldn't happen, but avoids warnings */
53     }
54 }
55 void set_init_vectors(SDL_Surface * surface,vector<vector<int>> &initcoord_pers_vec,vector<vector<int>>
&initcoord_dest_vec,vector<vector<int>> &initcoord_obst_vec, int p_x, int p_y, int d_x, int d_y, int o_x, int
o_y){//Liest den Grundriss ein
56
57     vector<int> ith_coord;
58     for (int y=0; y< grid_height; y++)
59     {
60         for (int x=0; x< grid_width; x++)
61         {
62             if (getpixel_function(surface, x, y)== getpixel_function(surface, p_x, p_y))
63             {
```

```

64         ith_coord.clear();
65         ith_coord.push_back(x);
66         ith_coord.push_back(y);
67
68         initcoord_pers_vec.push_back(ith_coord);
69     }
70     else if (getpixel_function(surface, x, y) == getpixel_function(surface, o_x, o_y))
71     {
72         ith_coord.clear();
73         ith_coord.push_back(x);
74         ith_coord.push_back(y);
75
76         initcoord_obst_vec.push_back(ith_coord);
77     }
78     else if (getpixel_function(surface, x, y) == getpixel_function(surface, d_x, d_y))
79     {
80         ith_coord.clear();
81         ith_coord.push_back(x);
82         ith_coord.push_back(y);
83
84         initcoord_dest_vec.push_back(ith_coord);
85     }
86 }
87
88 }
89
90 void print_init_vector(vector<vector<int>> &initcoord_vec){
91     cout << "Init Koordinaten sind:" << endl;
92     for(int i = 0; i < initcoord_vec.size(); i++){
93         cout << "(" << initcoord_vec[i][0] << ";" << initcoord_vec[i][1] << ")" << " ";
94     }
95     cout << endl;
96 }
97 //#### Grundriss einlesen
98
99
100
101 //#### Grafikausgabe
102 void clear_drawing(SDL_Renderer *renderer){// clears the whole screen/pigment the whole screen white
103     //NOCHMAL DURCH RICHTIGEN BEFEHL ERSETZEN
104     SDL_SetRenderDrawColor(renderer,250,250,250,0);
105     const SDL_Rect scrrect = {0,0,grid_width*10,grid_height*10}; //declare rectangle, which contains the
whole screen -> NOCHMAL NACHBESSERN : NUR BENÄ?TIGTER PLATZ
106     SDL_RenderFillRect(renderer, &scrrect);
107 }
108 void draw_grid(vector<person> &pa, vector<destination> &da, vector<obstacle> &oa, SDL_Renderer *renderer
){ //draw all objects on screen
109     clear_drawing(renderer); // pigment the screen white
110     for(int p = 0; p < pa.size(); p++){
111         SDL_SetRenderDrawColor(renderer, pa[p].r, pa[p].g, pa[p].b, 0);
112         SDL_RenderDrawPoint(renderer, pa[p].x, pa[p].y);
113     }
114     for(int d = 0; d < da.size(); d++){
115         SDL_SetRenderDrawColor(renderer,da[d].r, da[d].g, da[d].b, 0);
116         SDL_RenderDrawPoint(renderer, da[d].x, da[d].y);
117     }
118     for(int o = 0; o < oa.size(); o++){
119         SDL_SetRenderDrawColor(renderer,oa[o].r, oa[o].g, oa[o].b, 0);
120         SDL_RenderDrawPoint(renderer, oa[o].x, oa[o].y) ;
121     }
122     SDL_RenderPresent(renderer);
123 }
124 void draw_grid(vector<person> &pa, vector<destination> &da, vector<obstacle> &oa, SDL_Renderer *renderer,
int magnification_factor){//draw all objects on screen with a magnification ; mÄ?glicherweise eine grÄ?ere
VergrÄ?Ä?erung ermÄ?glichen ?

```

```

125
126     clear_drawing(renderer); // pigment the screen white
127     if (magnification_factor == 1){
128         draw_grid(pa,da,oa,renderer);
129     }
130     if (magnification_factor == 2){
131         int shift_factor = magnification_factor; //shift of all points because of the magnification
132         int quantity_drawing_points = 4; //quadrieren vom magnification factor
133
134         for(int p = 0; p < pa.size(); p++){
135             SDL_SetRenderDrawColor(renderer, pa[p].r, pa[p].g, pa[p].b, 0);
136             SDL_Point drawing_points[quantity_drawing_points] = {
137                 {pa[p].x + pa[p].x * shift_factor , pa[p].y + pa[p].y * shift_factor},
138                 {pa[p].x + pa[p].x * shift_factor + 1 , pa[p].y + pa[p].y * shift_factor},
139                 {pa[p].x + pa[p].x * shift_factor , pa[p].y + pa[p].y * shift_factor + 1},
140                 {pa[p].x + pa[p].x * shift_factor + 1 , pa[p].y + pa[p].y * shift_factor + 1},
141             };
142             SDL_RenderDrawPoints(renderer,drawing_points,quantity_drawing_points);
143         }
144
145         for(int d = 0; d < da.size(); d++){
146             SDL_SetRenderDrawColor(renderer, da[d].r, da[d].g, da[d].b, 0);
147             SDL_Point drawing_points[quantity_drawing_points] = {
148                 {da[d].x + da[d].x * shift_factor , da[d].y + da[d].y * shift_factor},
149                 {da[d].x + da[d].x * shift_factor + 1 , da[d].y + da[d].y * shift_factor},
150                 {da[d].x + da[d].x * shift_factor , da[d].y + da[d].y * shift_factor + 1},
151                 {da[d].x + da[d].x * shift_factor + 1 , da[d].y + da[d].y * shift_factor + 1}
152             };
153             SDL_RenderDrawPoints(renderer,drawing_points,quantity_drawing_points);
154         }
155
156         for(int o = 0; o < oa.size(); o++){
157             SDL_SetRenderDrawColor(renderer,oa[o].r, oa[o].g, oa[o].b, 0);
158             SDL_Point drawing_points[quantity_drawing_points] = {
159                 {oa[o].x + oa[o].x * shift_factor , oa[o].y + oa[o].y * shift_factor},
160                 {oa[o].x + oa[o].x * shift_factor + 1 , oa[o].y + oa[o].y * shift_factor},
161                 {oa[o].x + oa[o].x * shift_factor , oa[o].y + oa[o].y * shift_factor + 1},
162                 {oa[o].x + oa[o].x * shift_factor + 1 , oa[o].y + oa[o].y * shift_factor + 1}
163             };
164             SDL_RenderDrawPoints(renderer,drawing_points,quantity_drawing_points);
165         }
166
167     }
168     SDL_RenderPresent(renderer);
169 }
170 ##### Grafikausgabe
171
172
173 ##### Vorgehen waehrend Iteration
174 void move_people_sequential(vector<person> &persvec, vector<obstacle> &obstvec, vector<destination> &destvec, vector <int > &propability_arr_diff, vector<int> &propability_arr_dec){
175     //cout << "SIND GERADE HIER AM ARBEITEN" << endl;
176     //Vector wird mit allen Nummern gefÃ¤llt; jede Nummer kann genau einer Person zugeordnet werden kann
177     vector<int> serial_number_pers;
178     for(int i = 0; i < persvec.size(); i++){
179         serial_number_pers.push_back(i);
180         //cout << i;
181     }
182     //cout << endl;
183     //Bewegung jeder Person:
184     for(int j = 0; j < persvec.size(); j++){
185         //Aussuchen: wer ist dran?
186         int l = rand() % (persvec.size() - j);
187         //cout << "l: " << l << endl;
188         int whose_turn = serial_number_pers[l];
189         //cout << "whose turn: " << whose_turn << endl;

```

```

190     serial_number_pers.erase(serial_number_pers.begin() + 1);
191
192 //Abrufen: Erstellen der transition Matrix:
193 persvec[j].set_T(obstvec,persvec);
194
195 //Aussuchen: Bewegungsrichtung:
196 double r = ((rand() % 100) / 100.);
197 //cout << "Zufallszahl r: " << r << endl;
198
199 if(r < persvec[j].get_T(1,0)){// Bewegung nach oben?
200     persvec[j].moveto(persvec[j].x, persvec[j].y - 1);
201 }
202 else if(r < (persvec[j].get_T(1,0) + persvec[j].get_T(2,1))){//nach rechts?
203     persvec[j].moveto(persvec[j].x + 1, persvec[j].y);
204 }
205 else if(r < (persvec[j].get_T(1,0) + persvec[j].get_T(2,1) + persvec[j].get_T(1,2))){//nach unten?
206     persvec[j].moveto(persvec[j].x, persvec[j].y + 1);
207 }
208 else if(r < (persvec[j].get_T(1,0) + persvec[j].get_T(2,1) + persvec[j].get_T(1,2) + persvec[j].get_T(0
,1))){//nach links?
209     persvec[j].moveto(persvec[j].x - 1, persvec[j].y);
210 }
211 else{//stehen bleiben
212 }
213
214 }
215
216 }
217 void move_people_parallel(vector<person> &persvec, vector<obstacle> &obstvec, vector<destination> &destvec,
vector <int > &propability_arr_diff, vector<int > &propability_arr_dec){
218 //Jede Person entscheidet nun auf welche Koordinaten (desired_x,desired_y) sie gehen möchte
219 for(int i = 0; i < persvec.size(); i++){
220     //Voreinstellung der variable "wins_conflict", die für die nächste Schleife benötigt wird:
221     persvec[i].wins_conflict = false;
222     persvec[i].had_a_conflict = false;
223
224     //jede Person setzt ihre Transition Matrix:
225     persvec[i].set_T(obstvec,persvec);
226
227     //für jede Person wird eine random Zahl zwischen 0 und 1 erstellt:
228     double r = (rand() % 100) / 100. ;
229
230     //Ermittelt nun stochastisch in welche Richtung sich die Person bewegen möchte:
231     if(r < persvec[i].get_T(1,0)){ //nach oben?
232         persvec[i].desired_x = persvec[i].x;
233         persvec[i].desired_y = persvec[i].y - 1;
234         persvec[i].desired_direction = 'o';
235     }
236     else if(r < (persvec[i].get_T(1,0) + persvec[i].get_T(2,1))){//nach rechts?
237         persvec[i].desired_x = persvec[i].x + 1;
238         persvec[i].desired_y = persvec[i].y;
239         persvec[i].desired_direction = 'r';
240     }
241     else if(r < (persvec[i].get_T(1,0) + persvec[i].get_T(2,1) + persvec[i].get_T(1,2))){//nach unten?
242         persvec[i].desired_x = persvec[i].x;
243         persvec[i].desired_y = persvec[i].y + 1;
244         persvec[i].desired_direction = 'u';
245     }
246     else if(r < (persvec[i].get_T(1,0) + persvec[i].get_T(2,1) + persvec[i].get_T(1,2) + persvec[i].
get_T(0,1))){//nach links?
247         persvec[i].desired_x = persvec[i].x - 1;
248         persvec[i].desired_y = persvec[i].y;
249         persvec[i].desired_direction = 'l';
250     }
251     else{// stehen bleiben?
252         persvec[i].desired_x = persvec[i].x;

```

```

253         persvec[i].desired_y = persvec[i].y;
254         persvec[i].wins_conflict = true; //Möchte die Person stehen bleiben, so gewinnt diese Person
255         immer den Konflikt
256     }
257 }
258
259 ///Entscheidung welche Person sich bewegen darf und welche beispielsweise bei einem Konflikt stehen bleiben
muss:
260 for(int i = 0; i < persvec.size(); i++){
261     vector<int> conflict_partner;//Enthält Nummer der Konfliktpartner
262
263     //Fügt sich selbst zu den Konfliktpartnern hinzu:
264     conflict_partner.push_back(i);
265
266     //Fügt die anderen Konfliktpartner hinzu:
267     for(int j = 0; j < persvec.size(); j++){
268         //die Personen, die das selbe Ziel haben, werden zum Vektor conflict_partner hinzugefügt:
269         if(persvec[j].desired_x == persvec[i].desired_x && persvec[j].desired_y == persvec[i].desired_y
270             && i != j){
271             conflict_partner.push_back(j);
272         }
273     }
274     //Überprüft ob der Konflikt ausgetragen werden muss:
275     bool conflict_done = false;
276     //Überprüft ob nur eine Person im vector ist
277     if(conflict_partner.size() == 1){
278         persvec[conflict_partner[0]].wins_conflict = true;
279         conflict_done = true;
280     }
281     else{//Überprüft, ob dieser Konflikt schon ausgetragen wurde, also ob eine Person, des conflict
partner vectors, schon unter wins_conflict ein "true" zu stehen hat:
282         for(int j = 0; j < conflict_partner.size(); j++){
283             if(persvec[conflict_partner[j]].wins_conflict == true){
284                 conflict_done = true;
285             }
286             persvec[conflict_partner[j]].had_a_conflict=true;
287         }
288     }
289     //Wenn der Konflikt noch nicht ausgetragen wurde, also conflict_done == false ist, so wird dieser
ausgeführt:
290     if(conflict_done == false){
291         conflict con = conflict(persvec[conflict_partner[0]].desired_x,persvec[conflict_partner[0]].desired_y,
292         conflict_partner, persvec);
293         persvec[con.number_of_winner].wins_conflict = true;
294     }
295     //Reset des Vektors:
296     conflict_partner.clear();
297 }
298
299 //Bewegt die Personen, die die Konflikte gewonnen haben
300 for(int i = 0; i < persvec.size(); i++){
301     if(persvec[i].wins_conflict == true){
302         persvec[i].moveto(persvec[i].desired_x, persvec[i].desired_y, persvec[i].had_a_conflict);
303         //Nach konflikt muss hier noch true übergeben werden
304     }
305     //Die Personen die den Konflikt verloren haben erhalten ein D Feld steigerung auf der von ihnen aus
rechts gesehenem Feld -> Dies soll zu einer besseren Konfliktbewältigung führen:
306     else if(corridor_conditions == true){
307         int radius = 5; //Radius um die gefragt Person, die einen Konflikt hat, welcher angibt von
welchen Nachbarpersonen das D Feld verändert wird
308         if(persvec[i].desired_direction == 'o'){
309             //Veränderung des D Feldes der Nachbarpersonen
310             for(int j = 0; j < persvec.size(); j++){

```

```

311             if(persvec[j].desired_direction == 'o' && i!=j && persvec[j].x < persvec[i].x + radius
312             && persvec[j].x > persvec[i].x - radius && persvec[j].y < persvec[i].y + radius && persvec[j].y > persvec[i].y -
313             radius){
314                 if(persvec[j].could_I_go_to(persvec[j].x + 1,persvec[j].y,obstvec,persvec)){
315                     persvec[j].D[persvec[j].x + 1][persvec[j].y]++;
316                 }
317             }
318             //Veränderung des eigenen D Feldes
319             if(persvec[i].could_I_go_to(persvec[i].x + 1,persvec[i].y,obstvec,persvec)){
320                 persvec[i].D[persvec[i].x + 1][persvec[i].y]++;
321                 persvec[i].D[persvec[i].x + 1][persvec[i].y]++;
322             }
323         }
324         else if(persvec[i].desired_direction == 'r'){
325             for(int j = 0; j < persvec.size(); j++){
326                 if(persvec[j].desired_direction == 'r' && i!=j && persvec[j].x < persvec[i].x + radius
327                 && persvec[j].x > persvec[i].x - radius && persvec[j].y < persvec[i].y + radius && persvec[j].y > persvec[i].y -
328                 radius){
329                     if(persvec[j].could_I_go_to(persvec[j].x,persvec[j].y + 1,obstvec,persvec)){
330                         persvec[j].D[persvec[j].x][persvec[j].y + 1]++;
331                     }
332                 }
333                 if(persvec[i].could_I_go_to(persvec[i].x,persvec[i].y + 1,obstvec,persvec)){
334                     persvec[i].D[persvec[i].x][persvec[i].y + 1]++;
335                     persvec[i].D[persvec[i].x][persvec[i].y + 1]++;
336                 }
337             }
338             else if(persvec[i].desired_direction == 'u'){
339                 for(int j = 0; j < persvec.size(); j++){
340                     if(persvec[j].desired_direction == 'u' && i!=j && persvec[j].x < persvec[i].x + radius
341                     && persvec[j].x > persvec[i].x - radius && persvec[j].y < persvec[i].y + radius && persvec[j].y > persvec[i].y -
342                     radius){
343                         if(persvec[j].could_I_go_to(persvec[j].x - 1,persvec[j].y,obstvec,persvec)){
344                             persvec[j].D[persvec[j].x - 1][persvec[j].y]++;
345                         }
346                         if(persvec[i].could_I_go_to(persvec[i].x - 1,persvec[i].y,obstvec,persvec)){
347                             persvec[i].D[persvec[i].x - 1][persvec[i].y]++;
348                             persvec[i].D[persvec[i].x - 1][persvec[i].y]++;
349                         }
350                     }
351                     else if(persvec[i].desired_direction == 'l'){
352                         for(int j = 0; j < persvec.size(); j++){
353                             if(persvec[j].desired_direction == 'l' && i!=j && persvec[j].x < persvec[i].x + radius
354                             && persvec[j].x > persvec[i].x - radius && persvec[j].y < persvec[i].y + radius && persvec[j].y > persvec[i].y -
355                             radius){
356                                 if(persvec[j].could_I_go_to(persvec[j].x,persvec[j].y - 1,obstvec,persvec)){
357                                     persvec[j].D[persvec[j].x][persvec[j].y - 1]++;
358                                 }
359                                 if(persvec[i].could_I_go_to(persvec[i].x,persvec[i].y - 1,obstvec,persvec)){
360                                     persvec[i].D[persvec[i].x][persvec[i].y - 1]++;
361                                     persvec[i].D[persvec[i].x][persvec[i].y - 1]++;
362                                 }
363                             }
364                         }
365                     }
366                 }
367             }
368     bool has_pers_reached_destination(vector<destination> &destvec, vector<person> &persvec){//Ueberprueft ob

```

```

die Person das Ziel erreicht hat
369     bool return_value = false;
370
371     for(int i = 0; i < persvec.size(); i++){
372         for(int j = 0; j < destvec.size(); j++){
373             int nh[10] = { //neighbours of the selected destination
374                 destvec[j].x, destvec[j].y + 1,
375                 destvec[j].x + 1, destvec[j].y,
376                 destvec[j].x, destvec[j].y - 1,
377                 destvec[j].x - 1, destvec[j].y,
378                 destvec[j].x, destvec[j].y
379             };
380             for(int k = 0; k < 5; k++){
381                 if(persvec[i].x == nh[2*k] && persvec[i].y == nh[2*k + 1] && persvec[i].evacuated ==
382                     false){//Ist die Person ein Nachbar des Ziels?
383                     if (corridor_conditions==true)
384                     {
385                         int max_x=0;
386                         int min_x=grid_width;
387                         for (int i=0; i<destvec.size(); i++)
388                         {
389                             if (destvec[i].x > max_x)
390                             {
391                                 max_x=destvec[i].x;
392                             }
393                             if (destvec[i].x < min_x)
394                             {
395                                 min_x=destvec[i].x;
396                             }
397                             if (destvec[j].x==max_x)
398                             {
399                                 persvec[i].moveto(min_x+2, destvec[j].y);
400                             }
401                             if (destvec[j].x==min_x)
402                             {
403                                 /*cout << "max_x= " << max_x << endl;
404                                 cout << "persvec[i].x= " << persvec[i].x << endl;*/
405                                 persvec[i].x=max_x-2;
406                                 //cout << "persvec[i].x= " << persvec[i].x << endl;
407                             }
408                             persvec[i].evacuated = false;
409                         }
410                     }
411                     else
412                     {
413                         persvec[i].moveto(destvec[j].x,destvec[j].y);
414                         persvec[i].evacuated = true; //damit sich die Person nicht mehr aus dem Ziel
hinausbewegt
415                         persvec[i].iteration_when_evacuated = persvec[i].iteration; // Stoppt
"Iterationsmessung"
416                         persvec[i].end_time_measurement();// Stoppt Zeitmessung
417                         return_value = true;
418                     }
419                 }
420             }
421             if(persvec[i].x == nh[2*4] && persvec[i].y == nh[2*4 + 1] && persvec[i].evacuated == true
422                 && return_value == false && corridor_conditions==false){
423                 persvec[i].aax = persvec[i].ax;
424                 persvec[i].aay = persvec[i].ay;
425                 persvec[i].ax = persvec[i].x;
426                 persvec[i].ay = persvec[i].y;
427             }
428         }
429     }
429     return return_value;

```

```

430  }
431  void update_object_parameters(int iteration, vector<person> &persvec, vector<destination> &destvec, vector<
432 //Erneuert Parameter, wird nach jedem Iterationsschritt aufgerufen
433  for(int j = 0; j < persvec.size(); j++)
434  {
435      //cout << "persvec[j].x= " << persvec[j].x << endl;
436      persvec[j].iteration = iteration;
437      if(corridor_conditions == false){ persvec[j].renew_w_S_and_S(destvec,foreign_call); }
438      persvec[j].last_movement_direction = persvec[j].set_last_movement_direction(persvec[j].ax, persvec[
439      j].ay, persvec[j].x, persvec[j].y);
440      persvec[j].a_last_movement_direction = persvec[j].set_last_movement_direction(persvec[j].aax,
441      persvec[j].aay, persvec[j].ax, persvec[j].ay);
442      persvec[j].set_D_3(persvec, j);
443      persvec[j].set_panic_par(persvec, j, iteration);
444      persvec[j].diffusion_dyn_f(propability_arr_diff, persvec, persvec[j].x, persvec[j].y, j, obstvec,
445      propability_arr_dec);
446      persvec[j].decay_dyn_f(propability_arr_dec, persvec, j);
447  }
448 //Wenn Ziele nebeneinanderliegen, kann folgende Funktion ausgeführt werden, damit die Personen beide Ziele
449 //als ein Ziel" sehen (w_S wird bei beiden gleich gesetzt)
450 void unite_destinations(vector <person> &persvec, vector <destination> &destvec){
451     for(int l = 0; l < destvec.size(); l++){//Damit auch Nachbarn von Nachbarn angepasst werden
452         (destvec.size() ist die maximal nötige Anzahl an Iterationen)
453         for(int i = 0; i < destvec.size(); i++){
454             for(int j = 0; j < destvec.size(); j++){
455                 if((destvec[i].x == destvec[j].x + 1 && destvec[i].y == destvec[j].y) || (destvec[i].x ==
456                 destvec[j].x - 1 && destvec[i].y == destvec[j].y) || (destvec[i].x == destvec[j].x && destvec[i].y == destvec[j].
457                 y + 1) || (destvec[i].x == destvec[j].x && destvec[i].y == destvec[j].y - 1)){//wenn ja dann sind beides
458                 Nachbarn
459                 if(l == destvec.size()){
460                     destvec[i].dest_neighbours.push_back(j);
461                 }
462                 for(int k = 0; k < persvec.size(); k++){
463                     if(persvec[k].w_S[i] > persvec[k].w_S[j]){
464                         persvec[k].w_S[j] = persvec[k].w_S[i];
465                     }
466                     else{
467                         persvec[k].w_S[i] = persvec[k].w_S[j];
468                     }
469                 }
470             }
471         }
472 //#####
473 //#####
474 void adapt_w_S_has_only_one_destination(vector<person> &persvec, vector<destination> &destvec){/// Sorgt
475 //dafür, dass alle Personen nur ein Ziel kennen, dies wird für die Simulation des Korridors benötigt
476     //cout << "HIER !" << endl;
477     for(int i = 0; i < persvec.size(); i++){
478         //Setzt alle w_S Parameter einer Person auf 0 außer die von einem einzigen zufällig ausgewähltem Ziel:
479         bool w_S_modified = false; //zur Überprüfung der korrekten Ausführung der Änderung von w_S
480         //Überschreibung des vectors w_S, damit nur noch ein Eintrag von w_S die Zahl 1 enthält (alle
481         anderen werden mit 0 gefüllt)
482         persvec[i].set_w_S(0.0);
483         persvec[i].set_w_S(1,false);
484         for(int j = 0; j < destvec.size(); j++){
485

```

```

484         if(persvec[i].w_S[j] == 1){
485             persvec[i].numb_selected_dest = j;
486             //Änderung der Farbe der Personen
487             persvec[i].g = (int)(j * 250 / destvec.size());
488
489             w_S_modified = true;
490         }
491     }
492     //Fehlerueberpruefung:
493     if(w_S_modified == false){
494         cout << "Fehler: w_S wurde nicht dem Betriebsmodus angepasst" << endl;
495     }
496 }
497 }
498 }
499 void set_analyse_parameters(analysis_run &ana_run, char *k_S, char *k_D, char *w_S, char *friction, char *
w_decay, char *w_diffusion){//setzt Parameter aus einem Ausruf aus der Shell
500     ana_run.k_S = atoi(k_S) / 1000.;
501     ana_run.k_D = atoi(k_D) / 1000.;
502     ana_run.w_S = atoi(w_S) / 1000.;
503     ana_run.friction = atoi(friction) / 1000.;

504     decay_param = atoi(w_decay);
505     diffusion_param = atoi(w_diffusion);

506     cout << ana_run.k_S << ";" << ana_run.k_D << ";" << ana_run.w_S << ";" << ana_run.friction << ";" <<
decay_param << ";" << diffusion_param << endl;
509 }
510 void set_model_parameters(analysis_run ana_run, vector<person> &persvec, vector<destination> &destvec,
vector<obstacle> &obstvec){//setzt Parameter aller Personen; dies ist fÃ¼r die Analyse der Evakuierungszeit
unabdingbar
511     if(ana_run.execute == true){
512         for(int i = 0; i < persvec.size(); i++){
513             if(ana_run.k_S >= 0){
514                 persvec[i].k_S = ana_run.k_S;
515             }
516             if(ana_run.k_D >= -20){
517                 persvec[i].k_D = ana_run.k_D;
518             }
519             if(ana_run.w_S >= 0){
520                 persvec[i].set_w_S(ana_run.w_S); //Setzt w_S aller Ziela auf 1
521             }
522             if(ana_run.friction >= 0){
523                 if(ana_run.friction <= 1){
524                     persvec[i].friction = ana_run.friction;
525                 }
526                 else{
527                     cout << "Fehler - der Friction Parameter kann nicht groesser als 1 sein!" << endl;
528                     break;
529                 }
530             }
531         }
532     }
533     if(corridor_conditions == true){//Handelt es sich bei der geforderten Situation um einen Korridor, so
ist reject_other_D_fields automatisch aktiviert
534         adapt_w_S_has_only_one_destination(persvec,destvec);
535         unite_destinations_if_possible = true;
536         for(int i = 0; i < persvec.size(); i++){
537             persvec[i].set_S_corridor(persvec,destvec,obstvec);
538         }
539     }
540     if(reject_other_D_fields == true && corridor_conditions == false){
541         adapt_w_S_has_only_one_destination(persvec,destvec);
542     }
543     if(unite_destinations_if_possible == true){
544         unite_destinations(persvec,destvec);

```

```

545     }
546     if(take_which_exit == "near"){ //Wenn es einen näheren Ausang gibt wird w_S so verändert, dass dieser
547     angesteuert wird
548     for(int j = 0; j < persvec.size(); j++){
549         // Suchen des Ziels, zu welchem die Person am wenigsten Schritte machen muss
550         int max_S_k = 0;
551         int numb_new_dest;
552         for (int i = 0; i < destvec.size(); i++){
553             if (destvec[i].S_k[persvec[j].x][persvec[j].y] > max_S_k){
554                 max_S_k = destvec[i].S_k[persvec[j].x][persvec[j].y];
555                 numb_new_dest = i;
556                 //cout << min_S_k << endl;
557             }
558         }
559         //Veränderung von w_S, alle weiter entfernten Ziele erhalten ein w_S von 0
560         for (int i = 0; i < persvec[j].w_S.size(); i++){
561             if(i != numb_new_dest){
562                 persvec[j].w_S[i] = 0;
563             }
564             else{
565                 persvec[j].w_S[i] = 1;
566                 persvec[j].g = (int)(i * 250 / destvec.size()); //Ändert die Farbe der Personen, damit
567                 klar ersichtlich ist, welche PErsone welche Ziel ansteuert
568             }
569         }
570     }
571     else if(take_which_exit == "far"){
572     for(int j = 0; j < persvec.size(); j++){
573
574         // Suchen des Ziels, zu welchem die Person am meisten Schritte machen muss
575         int min_S_k = 100000;
576         int numb_new_dest;
577         for (int i = 0; i < destvec.size(); i++){
578             if (destvec[i].S_k[persvec[j].x][persvec[j].y] < min_S_k){
579                 min_S_k = destvec[i].S_k[persvec[j].x][persvec[j].y];
580                 numb_new_dest = i;
581                 //cout << min_S_k << endl;
582             }
583         }
584         //Veränderung von w_S, alle weiter entfernten Ziele erhalten ein w_S von 0
585         for (int i = 0; i < persvec[j].w_S.size(); i++){
586             if(i != numb_new_dest){
587                 persvec[j].w_S[i] = 0;
588             }
589             else{
590                 persvec[j].w_S[i] = 1;
591                 persvec[j].g = (int)(i * 250 / destvec.size()); //Ändert die Farbe der Personen, damit
592                 klar ersichtlich ist, welche PErsone welche Ziel ansteuert
593             }
594         }
595     }
596 }
597
598 void evacuation_analysis(vector<person> &persvec){ // Analysiert die Evakuierungszeit der Personen, sollte
599     nur ausgeführt werden, wenn vorher "set_model_parameters" angewendet wurde, also analysis_run.execute aktiviert
600     ist
601     //öffnet ein Dokument, in dem alle Daten gespeichert werden:
602     fstream f;
603     f.open("daten.dat", ios::app);
604
605     // Berechnet die durchschnittliche Evakuierungszeit der Personen, die ihre Ziele erreichen
606     double average_evac_time = 0;
607     int number_evac_pers = 0;

```

```

606     for(int i = 0; i < persvec.size(); i++){
607         if(persvec[i].evacuated == true){
608             //cout << i << " endtime: " << persvec[i].time_end << endl;
609             average_evac_time = average_evac_time + persvec[i].evacuation_time;
610             number_evac_pers++;
611         }
612     }
613     average_evac_time = average_evac_time / number_evac_pers;
614     // Berechnet die durchschnittlich benötigte Iterationsanzahl, damit die Personen ans Ziel kommen:
615     double average_evac_iteration = 0;
616     for(int i = 0; i < persvec.size(); i++){
617         if(persvec[i].evacuated == true){
618             //cout << i << "enditeration: " << persvec[i].iteration_when_evacuated << endl;
619             average_evac_iteration = average_evac_iteration + persvec[i].iteration_when_evacuated;
620         }
621     }
622     average_evac_iteration = average_evac_iteration / number_evac_pers;
623
624     //Berechnet die durchschnittliche Anzahl an Kollisionen der einzelnen Personen:
625     double average_number_conflicts = 0;
626     for(int i = 0; i < persvec.size(); i++){
627         //cout << i << "Anzahl Konflikte: " << persvec[i].number_of_conflicts << endl;
628         average_number_conflicts = average_number_conflicts + persvec[i].number_of_conflicts;
629     }
630     average_number_conflicts = average_number_conflicts / number_evac_pers;
631
632     //Schreibt berechnete Daten in das geöffnete Dokument
633     //Reihenfolge der Daten ist: Name Grundris, Durchschnittliche Evakuierungszeit, Durchschnittliche
634     //Iteration bei Evakuierung, Anzahl der Personen, die das Ziel nicht erreichen, Anzahl der Personen, k_S, k_D,
635     //w_S, friction, decay, diffusion Update Regel, Grafik_Delay, Durchschnittliche Anzahl der Kollisionen
636     f << (string) plant_layout << " " << average_evac_time << " " << average_evac_iteration << " " <<
637     persvec.size() - number_evac_pers << " " << persvec.size() << " " << persvec[0].k_S << " " << persvec[0].k_D <<
638     " " << persvec[0].w_S[0] << " " << persvec[0].friction << " " << decay_param << " " << diffusion_param << " " <<
639     movement_update << " " << grafic_delay << " " << average_number_conflicts << endl;
640     f.close();
641     /*
642     Density
643     panic
644     k_D
645     k_S
646     w_S - Wissen über das Ziel
647     omega - steht für w_S
648     */
649     //#####
650     //##### Analyse
651     void lege_und_printe_grunriss_auf_dfeld (vector<person> &persvec, vector<obstacle> &obstvec, vector<
652     destination > &destvec, int ith_person, vector<vector<int>> &initcoord_pers_vec)
653     {
654         //erzeugt grundriss
655         int static Grundriss [grid_width][grid_height];
656         for (int x=0; x<grid_width; x++)
657         {
658             for (int y=0; y<grid_height; y++)
659             {
660                 Grundriss[x][y]=0;
661             }
662         }
663
664         for (int i=0; i< obstvec.size(); i++)
665         {

```

```

666     Grundriss [obstvec[i].x][obstvec[i].y]=-10;
667 }
668
669 for (int j=0; j< destvec.size(); j++)
670 {
671     Grundriss [destvec[j].x][destvec[j].y]=-20;
672 }
673
674 for (int k=0; k < persvec.size(); k++)
675 {
676     Grundriss [initcoord_pers_vec[k][0]][initcoord_pers_vec[k][1]]=-30;
677 }
678
679 // addiert grundriss mit d feld
680 int static Grundriss_D_Feld[grid_width][grid_height];
681 for (int y=0; y<grid_height; y++)
682 {
683     for (int x=0; x<grid_width; x++)
684     {
685         Grundriss_D_Feld[x][y]=persvec[ih_person].D[x][y]+Grundriss[x][y];
686     }
687 }
688 //print
689 for (int y=-1; y<grid_width; y++)
690 {
691     if (y== -1)
692     {
693         cout << " ";
694     }
695     if (0<=y && y <=9)
696     {
697         cout << " " << y ;
698     }
699     if (10<=y && y <=99)
700     {
701         cout << y ;
702     }
703     for (int x=0; x<grid_height; x++)
704     {
705
706         if (y== -1)
707         {
708             if (0<=x && x <=9)
709             {
710                 cout << " " << x << " ;";
711             }
712             if (10<=x && x <=99)
713             {
714                 cout << " " << x << " ;";
715             }
716         }
717         //cout << Grundriss_D_Feld[x][y] << " ;"
718         if (y>=0)
719         {
720             if (0<= Grundriss_D_Feld[x][y] && Grundriss_D_Feld[x][y]<=9)
721             {
722                 cout << " " << Grundriss_D_Feld[x][y] << " ;";
723             }
724             else if (10<=Grundriss_D_Feld[x][y] && Grundriss_D_Feld[x][y] <= 99)
725             {
726                 cout << " " << Grundriss_D_Feld[x][y] << " ;";
727             }
728             else
729             {
730                 cout << Grundriss_D_Feld[x][y] << " ;";
731             }

```

```

732         }
733     }
734     cout << endl;
735   }
736 }
737
738 //test
739
740
741
742 int main(int argc, char* args[]){
743   //Ueberpruefung der angegebenen Parameter:
744   ifstream file_test(plant_layout);
745   if(!file_test){
746     std::cout << "Der Name des Gebaeudeplans wurde falsch eingegeben oder diese Datei existiert nicht.
Korrigieren Sie die Eingabe der Variable plant_layout! " << endl;
747     return EXIT_SUCCESS;
748   }
749
750   cout << "Programm wurde gestartet;" << endl;
751
752   srand (time(NULL));
753
754
755   analysis_run ana_run;
756   //Fuer den Aufruf ueber die Shell, bzw fuer den Aufruf ueber die Batch Datei:
757   if(ana_run.foreign_call == true){
758     set_analyse_parameters(ana_run, args[1], args[2], args[3], args[4], args[5], args[6]);
759   }
760
761   cout << "Grundriss wird geladen..." << endl;
762
763   //Initialisation SDL um den gespeicherten Grundriss zu laden
764   SDL_Event Event;
765   SDL_Window* Window;
766   SDL_Surface* screen_surface;
767   SDL_Surface* bmp_surf;
768   SDL_PixelFormat *fmt;
769   SDL_Init( SDL_INIT_VIDEO );
770   Window = SDL_CreateWindow( "Grundriss", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, grid_width,
grid_height, SDL_WINDOW_SHOWN );
771   screen_surface = SDL_GetWindowSurface( Window );
772   bmp_surf = SDL_LoadBMP(plant_layout);
773
774   cout << "Grundriss geladen;" << endl;
775
776   //Zeigt Grundriss, mit dem das Programm arbeiten wird
777   SDL_BlitSurface( bmp_surf, NULL, screen_surface, NULL );
778   SDL_UpdateWindowSurface( Window );
779
780   //get start conditions
781   vector <vector <int >> initcoord_dest_vec;
782   vector <vector <int >> initcoord_pers_vec;
783   vector <vector <int >> initcoord_obst_vec;
784   set_init_vectors(bmp_surf,initcoord_pers_vec,initcoord_dest_vec,initcoord_obst_vec,2,0,1,0,0,0);
//befÃ¶lilt die initcoord Vektoren
785   int quantity_persons = initcoord_pers_vec.size();
786   int quantity_destinations = initcoord_dest_vec.size();
787   int quantity_obstacles = initcoord_obst_vec.size();
788
789   //Ausgabe der Koordinaten der noch zu erstellenden Personen, Hindernissen, Zielen
790   cout << "-----" << endl;
791   cout << "dest " ;
792   print_init_vector(initcoord_dest_vec);
793   cout << "Anzahl Ziele: " << initcoord_dest_vec.size()<< endl;
794   cout << "-----" << endl;

```

```

795     cout << "obst " ;
796     print_init_vector(initcoord_obst_vec);
797     cout << "Anzahl Gegenstaende: " << initcoord_obst_vec.size()<< endl;
798     cout << "-----" << endl;
799
800
801     cout << "pers " ;
802     print_init_vector(initcoord_pers_vec);
803     cout << "Anzahl Personen: " << initcoord_pers_vec.size()<< endl;
804     cout << "-----" << endl;
805
806 //Schliessen des SDL_Fensters
807 while (ana_run.foreign_call == false) {if (SDL_PollEvent(&Event) && Event.type == SDL_QUIT){break;}}
//HÄsst Fenster so lange offen bis es per Hand geschlossen wird
808     SDL_FreeSurface( bmp_surf );
809     bmp_surf = NULL;
810     SDL_DestroyWindow( Window );
811     Window = NULL;
812     SDL_Quit();
813
814
815
816
817 //test
818
819 vector <int > propability_arr_diff(100);
820 vector <int> propability_arr_dec(100);
821
822 cout << "Wahrscheinlichkeitsarrays wurden erstellt;" << endl;
823
824 //test
825
826 ##### object declaration
827 //declaration of used objects:
828     vector<person> persvec;
829     persvec.resize(quantity_persons);
830
831     vector<destination> destvec;
832     destvec.resize(quantity_destinations);
833
834     vector<obstacle> obstvec;
835     obstvec.resize(quantity_obstacles);
836 //construction of used objects
837     for(int o = 0; o < quantity_obstacles; o++){
838         obstvec[o] = obstacle(initcoord_obst_vec[o][0],initcoord_obst_vec[o][1],quantity_obstacles,
839         quantity_destinations,quantity_persons);
840     }
841     for(int d = 0; d < quantity_destinations; d++){
842         destvec[d] = destination(initcoord_dest_vec[d][0],initcoord_dest_vec[d][1],obstvec,
843         quantity_obstacles,quantity_destinations,quantity_persons);
844     }
845     for(int p = 0; p < quantity_persons; p++){
846         persvec[p] = person(initcoord_pers_vec[p][0],initcoord_pers_vec[p][1],destvec,quantity_obstacles,
847         quantity_destinations,quantity_persons);
848     }
849
850     set_model_parameters(ana_run, persvec, destvec, obstvec);
851
852 ##### object declaration
853
854     cout << "Objekte wurden erstellt;" << endl;
855     cout << " " << endl;

```

```

856     cout << "k_D: " << persvec[0].k_D << endl;
857     cout << "k_S: " << persvec[0].k_S << endl;
858
859     cout << " " << endl;
860     cout << "Diff. Parameter: " << diffusion_param << endl;
861     cout << " " << endl;
862     cout << "Dec. Parameter: " << decay_param << endl;
863
864     cout << " " << endl;
865     cout << "corridorconditions: " << corridor_conditions << endl;
866
867     cout << " " << endl;
868     cout << "reject_other_d_fields: " << reject_other_D_fields << endl;
869
870     cout << " " << endl;
871     cout << "grafic delay: " << grafic_delay << endl;
872
873     cout << " " << endl;
874     cout << "Bewegungsupdate: " << movement_update << endl;
875
876
877
878
879 //##### visual output 1
880     SDL_Renderer *renderer = NULL;
881     SDL_Window *window = NULL;
882     SDL_Init(SDL_INIT_VIDEO);
883     SDL_CreateWindowAndRenderer(grid_width*3, grid_height*3, 0, &window, &renderer);
884     SDL_Event event;
885 //#####
886
887
888
889 //test
890
891 //persvec[0].print_S();
892 //persvec[1].print_S();
893 /*
894 for(int i = 0; i < 4;i++){
895     cout << "Hier w_S:" << endl;
896     for(int j = 0; j < persvec[i].w_S.size(); j++){
897         cout << persvec[i].w_S[j] << ";";
898     }
899     cout << endl << "ausgewähltes dest: " << persvec[i].numb_selected_dest << endl;
900 }
901 */
902 //test
903 //Zeichnet einmal das Feld
904 draw_grid(persvec,destvec,obstvec,renderer,2);
905
906 for(int i = 0; i < max_number_of_iterations; i++){
907 //#####
908     //Damit das Programm beim Bewegen des Fensters nicht abstürzt und das Fenster geschlossen werden kann:
909     while(SDL_PollEvent(&event)){
910         if (event.type == SDL_QUIT){
911             SDL_DestroyRenderer(renderer);
912             SDL_DestroyWindow(window);
913             SDL_Quit();
914             return EXIT_SUCCESS;
915         }
916     }
917
918     //Haben Personen das Ziel erreicht:
919     has_pers_reached_destination(destvec,persvec);
920
921     //Bewege Personen:

```

XXX

```

922     if(movement_update == 's'){
923         move_people_sequential(persvec,obstvec,destvec, propability_arr_diff, propability_arr_dec);
924     }
925     else if(movement_update == 'p'){
926         move_people_parallel(persvec,obstvec,destvec, propability_arr_diff, propability_arr_dec);
927     }
928     else {
929         cout << "Fehler in der Eingabe: movement_update kann nur 'p' oder 's' sein" << endl;
930     }
931
932 //Abbruchbedingung, wenn die max_number_of_iterations zu hoch gewÃ¤hlt wurde
933 bool b_c = true;
934 for (int j = 0; j < persvec.size(); j++){
935     if(persvec[j].evacuated == false){
936         b_c = false;
937     }
938 }
939 if (b_c == true){
940     break;
941 }
942
943 //Erneuere die Parameter der Objekte, die sich im Laufe der Iteration verÃ¤ndert haben:
944 update_object_parameters(i,persvec,destvec,propability_arr_diff,propability_arr_dec, obstvec,
ana_run.foreign_call);
945
946 ////test
947
948 /*if (i%25==0)
949 {
950     cout << "
951     cout << "===== " << endl;
952     cout << "Personen haben sich zum " << i << "ten mal bewegt!" << endl;
953     cout << "===== " << endl;
954
955     cout << " " << endl;
956     cout << "D feld von person: 1" << endl;
957     persvec[1].print_D();
958
959     cout << "
960     cout << "Grundriss + dfeld von Person:" << "1" << endl;
961     lege_und_printe_grunriss_auf_dfeld(persvec, obstvec, destvec, 1, initcoord_pers_vec);
962 }*/
```

```
986     SDL_DestroyWindow(window);
987     SDL_Quit();
988
989
990
991
992
993
994
995     return EXIT_SUCCESS;
996 }
```

B.3. klassen.cpp

```
1 #include "initial_situation.cpp"
2 #include <stddef.h>
3 #include <iostream>
4 #include <time.h>
5 #include <stdio.h>
6 #include <vector>
7 #include <math.h>
8 #include <time.h>
9 #include <stdlib.h>
10
11 using namespace std;
12
13
14
15 class obstacle
16 {
17 public:
18 // constructors
19 obstacle(){};
20 obstacle(int nx, int ny, int q_obst, int q_dest, int q_pers){
21     x = nx;
22     y = ny;
23
24     setrgb(200,200,200);
25
26     quantity_obstacles = q_obst;
27     quantity_destinations = q_dest;
28     quantity_persons = q_pers;
29 };
30 obstacle(int nx, int ny, int f1, int f2, int f3, int q_obst, int q_dest, int q_pers){
31     x = nx;
32     y = ny;
33
34     setrgb(f1,f2,f3);
35
36     quantity_obstacles = q_obst;
37     quantity_destinations = q_dest;
38     quantity_persons = q_pers;
39 };
40 // constructors
41
42 // methods
43 void setrgb(int f1, int f2, int f3){
44     r = f1;
45     g = f2;
46     b = f3;
47 };
48 void moveto(int xn, int yn){
49     x = xn;
50     y = yn;
51 }
52 bool is_it_here(int qx, int qy){// has the object the coordinates (qx,qy) ?
53     if (x == qx && y == qy){
54         return true;
55     }
56     else {
57         return false;
58     }
59 }
60 // methods
61
62 //quantities
63 int quantity_obstacles;
64 int quantity_destinations;
65 int quantity_persons;
66
```

```

67 //coordinates
68     int x;
69     int y;
70 //colour
71     int r;
72     int g;
73     int b;
74
75 private:
76
77 };
78
79
#####
80
#####
81
82 class destination
83 {
84 public:
85 // constructors
86     destination(){
87     destination(int a, int b,vector<obstacle> &obstvec, int q_obst, int q_dest, int q_pers){
88         x = a;
89         y = b;
90         setrgb(0,200,0);
91         quantity_obstacles = q_obst;
92         quantity_destinations = q_dest;
93         quantity_persons = q_pers;
94
95         if(corridor_conditions == false){
96             set_static_field_k(obstvec); //wird für den Fall des Korridors nicht benötigt
97         }
98     }
99     destination(int a, int b, int f1, int f2, int f3,vector<obstacle> &obstvec, int q_obst, int q_dest,
int q_pers){
100        x = a;
101        y = b;
102        setrgb(f1,f2,f3);
103        quantity_obstacles = q_obst;
104        quantity_destinations = q_dest;
105        quantity_persons = q_pers;
106
107        if(corridor_conditions == false){
108            set_static_field_k(obstvec); //wird für den Fall des Korridors nicht benötigt
109        }
110    }
111 // constructors
112
113 // methods
114     void setrgb(int f1, int f2, int f3){
115         r = f1;
116         g = f2;
117         b = f3;
118     };
119     void moveto(int xn, int yn){
120         x = xn;
121         y = yn;
122     }
123     bool is_it_here(int qx, int qy){// has the object the coordinates (qx,qy) ?
124         if (x == qx && y == qy){
125             return true;
126         }
127     else {

```

```

128         return false;
129     }
130 }
131 // methods
132
133 //quantities
134 int quantity_obstacles;
135 int quantity_destinations;
136 int quantity_persons;
137 //coordinates
138     int x;
139     int y;
140 //coordinates
141 //colour
142     int r;
143     int g;
144     int b;
145 //colour
146 //Static field S_k, which is created by this destination:
147     int S_k[grid_width][grid_height];
148
149
150     bool could_a_person_go_to (int qx, int qy, vector<obstacle> &obstvec){//delivers true, if the person
could stay at (x,y)/ could move to this cell
151     //###cells out of borders arent available for a person:
152     if ((qy >= grid_height) || (qx >= grid_width)){
153         return false;
154     }
155     if ((qy < 0) || (qx < 0)){
156         return false;
157     }
158     //###cells filled by an obstacle isn't available for a person:
159     bool return_value = true;
160     for(int i = 0; i < quantity_obstacles; i++){
161         //cout << "x: " << obstvec[i].x << ", y: " << obstvec[i].y << endl;
162         if((obstvec[i].x == qx) && (obstvec[i].y == qy)){
163             return_value = false;
164         }
165     }
166     return return_value;
167 }
168 vector<int> to_do;
169 vector<int> processed;
170 void set_static_field_k(vector<obstacle> &obstvec){
171     //Setzt alle Einträge von S_k auf 0
172     for(int g = 0; g < grid_width; g++){
173         for(int h = 0; h < grid_height; h++){
174             S_k[g][h] = 0;
175         }
176     }
177     //Trägt Anfangskoordinaten in den vector ein
178     to_do.push_back(x);
179     to_do.push_back(y);
180
181     int counter = 0;
182
183     while(true){
184         //sichert die aktuelle Größe des vectors to_do für die nächste for-Schleife
185         int ntodo = to_do.size();
186         if (ntodo % 2 != 0){//Fehleranzeige, falls was schief läuft
187             cout << "Ein Fehler ist augetreten! die Anzahl ntodo ist nicht durch zwei teilbar." <<
endl;
188         }
189         for(int j = 0; j < (ntodo / 2); j++){
190
191             //Schreibe das zugehörige Potential in das statische Feld:

```

```

192             S_k[to_do[0]][to_do[1]] = counter;
193         //cout << S_k[8][8] << endl;
194         //if(counter == 3){print_S_k();}
195         //Diese Koordinaten sind nun abgearbeitet und werden dem vector processed übergeben
196         processed.push_back(to_do[0]);
197         processed.push_back(to_do[1]);
198         //Suche Nachbarn für das Feld S_k mit den Koordinaten (to_do[0],to_do[1]) und schreibe
diese in den vector to_do
199         int nh[8] = { //sind alle möglichen Nachbarn
200             to_do[0],to_do[1] + 1,
201             to_do[0] + 1,to_do[1],
202             to_do[0],to_do[1] - 1,
203             to_do[0] - 1,to_do[1]
204         };
205         for (int l = 0; l < 4; l++){//für jeden möglichen Nachbarn folgt:
206             if (could_a_person_go_to(nh[2*l],nh[2*l+1],obstvec) == false){// wenn eine Person hier
nicht drauf darf
207                 //cout << "Nicht erlaubte Felder could: " << nh[2*l]<< " ;" << nh[2*l+1] << endl;
208                 continue;
209             }
210             //Ist der aktuell ausgewählte Nachbar schon im Vector processed oder to_do enthalten:
211             int con = false;
212             for(int a = 0; a < processed.size()/2; a++){
213                 if (processed[2*a] == nh[2*l] && processed[2*a+1] == nh[2*l+1]){
214                     con = true;
215                 }
216             }
217             for(int b = 1; b < to_do.size()/2; b++){
218                 if(to_do[2*b] == nh[2*l] && to_do[2*b+1] == nh[2*l+1]){
219                     con = true;
220                 }
221             }
222             if(con == true){
223                 continue;
224             }
225             to_do.push_back(nh[2*l]);
226             to_do.push_back(nh[2*l+1]);
227         }
228         //Lösche nun die bearbeiteten Koordinaten (to_do[0],to_do[1])
229         to_do.erase(to_do.begin());
230         to_do.erase(to_do.begin());
231     }
232     if(to_do.size() == 0){
233         break;
234     }
235     counter++;
236 }
237 }
238 //Potentialfeld besitzt am Ausgang das größte Potential und nimmt von da an ab; also Werte müssen noch
"umgekehrt" werden:
239     //Finden des größten Eintrags in S_k:
240     int max_pot = 0;
241     for(int i = 0; i < grid_width; i++){
242         for(int j = 0; j < grid_height; j++){
243             if(S_k[i][j] > max_pot){
244                 max_pot = S_k[i][j];
245             }
246         }
247     }
248     //umkehren" der Einträge: also Potential verläuft vom Eingang aus gesehen von groß nach klein
249     for(int i = 0; i < grid_width; i++){
250         for(int j = 0; j < grid_height; j++){
251             if(could_a_person_go_to(i,j,obstvec)){
252                 S_k[i][j] = max_pot - S_k[i][j];
253             }
254         }

```

```

255         }
256     }
257 }
258 int get_S_k(int x, int y){
259     return S_k[x][y];
260 }
261 void print_S_k(){
262     cout << "-----" << endl;
263     for(int j = 0; j < grid_height; j++){
264         for(int i = 0; i < grid_width; i++){
265
266             if (S_k[i][j] > 9 && S_k[i][j] <= 99){cout << " " << S_k[i][j] << ";" ;}
267             else if (S_k[i][j] > 99){cout << S_k[i][j] << ";" ;}
268             else {cout << " " << S_k[i][j] << ";" ;}
269
270         }
271         cout << endl;
272     }
273 }
274
275 //fuer das vereinigen der Ziele:
276 vector<int> dest_neighbours;
277
278 private:
279
280 };
281
282
#####
283
#####
284
285 class person
286 {
287 public:
288 // constructors
289 person(){};
290 person(int nx, int ny,vector<destination> &destvec, int q_obst, int q_dest, int q_pers){
291     x = nx;
292     y = ny;
293     ax = x;
294     ay = y;
295     aax = ax;
296     aay = ay;
297     int colour_variation = (rand() % 150) - 75; //leichte Farbvariation, damit die einzelnen Personen
voneinander unterscheiden werden können
298     setrgb(0,0,150 + colour_variation);
299     quantity_obstacles = q_obst;
300     quantity_destinations = q_dest;
301     quantity_persons = q_pers;
302     //###Zu set_w_S
303     int p_d[1]; //bevorzugtes Ziel
304
305     p_d[0] = rand() % quantity_destinations; // bevorzugtes Ziel wird zufällig ausgewählt
306     set_w_S(true,1,p_d, rand() % (quantity_destinations) + 1); //die Person kennt also mindestens
eines der Ziele sehr gut .. der Rest wird zufällig entschieden
307
308
309     if(corridor_conditions == false){
310         set_S_normal(destvec);
311     }
312
313     set_D_on_zero();
314     //Zufälliges setzen des "Friction" Parameters:

```

```

315         friction = (rand() % 300) / 1000;
316         evacuated = false;
317         number_of_conflicts = 0;
318     };
319     person(int nx, int ny, int f1, int f2, int f3, vector<destination> &destvec, int q_obst, int q_dest,
320     int q_pers){
321         x = nx;
322         y = ny;
323         ax = x;
324         ay = y;
325         aax = ax;
326         aay = ay;
327
328         setrgb(f1,f2,f3);
329
330         quantity_obstacles = q_obst;
331         quantity_destinations = q_dest;
332         quantity_persons = q_pers;
333
334         //###Zu set_w_S
335         int p_d[1]; //bevorzugtes Ziel
336         p_d[0] = rand() % quantity_destinations; // bevorzugtes Ziel wird zufällig ausgewählt
337         set_w_S(true,1,p_d, rand() % (quantity_destinations) + 1); //die Person kennt also mindestens
eines der Ziele sehr gut .. der Rest wird zufällig entschieden
338
339         if(corridor_conditions == false){
340             set_S_normal(destvec);
341         }
342
343
344         set_D_on_zero();
345
346         //Zufälliges setzen des "Friction" Parameters:
347         friction = (rand() % 300) / 1000;
348
349         evacuated = false;
350         number_of_conflicts = 0;
351     };
352 // constructors
353
354 // methods
355     void setrgb(int f1, int f2, int f3){
356         r = f1;
357         g = f2;
358         b = f3;
359     };
360
361     double friction = 0.0; // Wahrscheinlichkeit, dass sich die Person nicht bewegt, obwohl sie sich
bewegen sollte
362
363     void moveto(int xn, int yn, bool after_conflict=false){
364         double r = (rand() % 1000) / 1000.0; // Zufallszahl
365         if((evacuated == false && r >= friction) || (evacuated == false && after_conflict == false)){
//Wenn die vor der Bewegung kein Konflikt stattgefunden hat wird die Bewegung auf jeden Fall ausgeführt; mit
Konflikt nur zu einer bestimmten Wahrscheinlichkeit, die von der "friction" abhängt
366
367             //set_D(persvec, xn, yn, propability_arr_diff, propability_arr_dec, obstvec); -> jetzt in
update_objekt_parameter
368             aax = ax;
369             aay = ay;
370             ax = xi;
371             ay = yi;
372             x = xn;
373             y = yn;
374         }

```

```

440     char set_last_movement_direction(int ax, int ay, int jx, int jy){ //ax steht für "altes x", jx fuer
"jetziges x", setzt die Variable last_movement_direction
441         if(jx > ax){
442             //Fehlerueberpruefung: ist die Bewegung genau 1 lang?:
443             if (jx != ax + 1 && evacuated == false && corridor_conditions ==false){
444                 cout << "Fehler: Die Person hat sich nicht exakt um 1 bewegt,1" << endl;
445             }
446             if(jy != jy && evacuated == false&& corridor_conditions ==false){
447                 cout << "Fehler: Bewegung der Person wurde falsch ausgeführt; x und y Koordinaten wurden
gleichzeitig verändert" << endl;
448             }
449
450             return 'r';
451         }
452         else if(jx < ax){
453             //Fehlerueberpruefung: ist die Bewegung genau 1 lang?:
454             if (jx != ax - 1 && evacuated == false&& corridor_conditions ==false){
455                 cout << "Fehler: Die Person hat sich nicht exakt um 1 bewegt,2" << endl;
456             }
457             if(jy != jy && evacuated == false&& corridor_conditions ==false){
458                 cout << "Fehler: Bewegung der Person wurde falsch ausgeführt; x und y Koordinaten wurden
gleichzeitig verändert" << endl;
459             }
460
461             return 'l';
462         }
463         else if(jy > ay){
464             //Fehlerueberpruefung: ist die Bewegung genau 1 lang?:
465             if (jy != ay + 1 && evacuated == false&& corridor_conditions ==false){
466                 cout << "Fehler: Die Person hat sich nicht exakt um 1 bewegt,3" << endl;
467             }
468             if(jx != jx && evacuated == false&& corridor_conditions ==false){
469                 cout << "Fehler: Bewegung der Person wurde falsch ausgeführt; x und y Koordinaten wurden
gleichzeitig verändert" << endl;
470             }
471
472             return 'u';
473         }
474         else if(jy < ay){
475             //Fehlerueberpruefung: ist die Bewegung genau 1 lang?:
476             if (jy != ay - 1 && evacuated == false&& corridor_conditions ==false){
477                 cout << "Fehler: Die Person hat sich nicht exakt um 1 bewegt,4" << endl;
478             }
479             if(jx != jx && evacuated == false&& corridor_conditions ==false){
480                 cout << "Fehler: Bewegung der Person wurde falsch ausgeführt; x und y Koordinaten wurden
gleichzeitig verändert" << endl;
481             }
482
483             return 'o';
484         }
485         else{//Person ist stehen geblieben
486             return 's';
487         }
488     }
489
490     //colour
491     int r;
492     int g;
493     int b;
494
495
496
497 // ##### Dynamic floor field DZur Uni Potsdam(Brandenburg):
498 double k_D = 1;
499 int panic_par;
500 int iterat_val;

```

```

501     int D[grid_width][grid_height];
502
503     void set_D_on_zero()
504     {
505         //setzt D feld auf null
506         for (int i = 0; i < grid_width; i++)
507         {
508             for(int j = 0; j < grid_height; j++)
509             {
510                 D[i][j] = 0;
511             }
512         }
513     }
514
515     void set_D_3 (vector <person> &persvec, int j)
516     {
517         if (k_D!=0)
518         {
519
520             for (int i=0; i< persvec.size(); i++) //gehe alle personen durch
521             {
522                 if (i!=j) //nicht von der eigenen spur beeinflussen
523                 {
524                     if((reject_other_D_fields == true && followed_the_pers_my_S(persvec[i],2) == true) ||
525 reject_other_D_fields == false) //bei mehreren zielen kann eine spur nur entstehen wenn die person in die
526                     gleich richtung laeuft
527                     {
528                         if ((persvec[i].ax!=persvec[i].x || persvec[i].ay!=persvec[i].y) && persvec[i].
529 evacuated == false) // überprüfe ob sich die person i bewegt hat
530                         {
531                             persvec[j].D[persvec[i].ax][persvec[i].ay]++;
532                         }
533                     if ((persvec[i].ax!=persvec[i].x || persvec[i].ay!=persvec[i].y) && persvec[i].
534 evacuated == false) // überprüfe ob sich die person i bewegt hat
535                         {
536                             persvec[j].D[persvec[i].ax][persvec[i].ay]--;
537                         }
538                     /*
539                     else
540                     {
541                         cout << "Fehler in set_D" << endl;
542                     }*/
543                 }
544             }
545         }
546     }
547
548     void decay_dyn_f(vector <int> &propability_arr_dec, vector <person> &persvec, int i)
549     {
550         if(decay_param == 0){return;}
551         bool grid_full=true;
552         //geht die gesamte fläche durch
553         for (int x=0; x< grid_width; x++)
554         {
555             for (int y=0; y<grid_height; y++)
556             {
557                 int d_abs_beginn = abs(persvec[i].D[x][y]);
558                 for (int j=0; j<d_abs_beginn; j++) //je staerker das d feld am punkt x y ist, desto
559                 {

```

```

501     int D[grid_width][grid_height];
502
503     void set_D_on_zero()
504     {
505         //setzt D feld auf null
506         for (int i = 0; i < grid_width; i++)
507         {
508             for(int j = 0; j < grid_height; j++)
509             {
510                 D[i][j] = 0;
511             }
512         }
513     }
514
515     void set_D_3 (vector <person> &persvec, int j)
516     {
517         if (k_D!=0)
518         {
519
520             for (int i=0; i< persvec.size(); i++) //gehe alle personen durch
521             {
522                 if (i!=j) //nicht von der eigenen spur beeinflussen
523                 {
524                     if((reject_other_D_fields == true && followed_the_pers_my_S(persvec[i],2) == true) ||
525 reject_other_D_fields == false) //bei mehreren zielen kann eine spur nur entstehen wenn die person in die
526                     gleich richtung laeuft
527                     {
528                         if ((persvec[i].ax!=persvec[i].x || persvec[i].ay!=persvec[i].y) && persvec[i].
529 evacuated == false) // überprüfe ob sich die person i bewegt hat
530                         {
531                             persvec[j].D[persvec[i].ax][persvec[i].ay]++;
532                         }
533                     if ((persvec[i].ax!=persvec[i].x || persvec[i].ay!=persvec[i].y) && persvec[i].
534 evacuated == false) // überprüfe ob sich die person i bewegt hat
535                         {
536                             persvec[j].D[persvec[i].ax][persvec[i].ay]--;
537                         }
538                     /*
539                     else
540                     {
541                         cout << "Fehler in set_D" << endl;
542                     }*/
543                 }
544             }
545         }
546     }
547
548     void decay_dyn_f(vector <int> &propability_arr_dec, vector <person> &persvec, int i)
549     {
550         if(decay_param == 0){return;}
551         bool grid_full=true;
552         //geht die gesamte fläche durch
553         for (int x=0; x< grid_width; x++)
554         {
555             for (int y=0; y<grid_height; y++)
556             {
557                 int d_abs_beginn = abs(persvec[i].D[x][y]);
558                 for (int j=0; j<d_abs_beginn; j++) //je staerker das d feld am punkt x y ist, desto
559                 {

```

```

560         //zerfall des d Feldes:
561         //-----
562         //schreibe wahrscheinlichkeitsarray
563         for (int k=0; k < propability_arr_dec.size(); k++)
564         {
565             propability_arr_dec[k]=0;
566         }
567         if (decay_param!=0)
568         {
569             for (int k=0; k<decay_param ; k++)
570             {
571                 propability_arr_dec[k]=1;
572             }
573         }
574         /*if (i==1)
575         {
576             cout << "Person " << i << " hat die Koordinaten: (x;y) = " << "(" << persvec[i].x
577             << ";" << persvec[i].y << ")" << endl;
578             cout << "D-Feld von Person " << i << " ungleich null am Punkt: (x;y) = "
579             << "(" << persvec[i].x << "," << persvec[i].y << ")" << endl;
580             cout << "#####" << endl;
581         }*/
582
583         //generiere zufällige zahl und überprüfe ob d feld zerfallen soll
584         int r_2=rand()%100;
585         if (propability_arr_dec[r_2]==1) //verifikation: D feld soll sich auflösen
586         {
587             //if /*falls 1 Nachbar frei ist*/ persvec[i].D[x][y+1]==0 || persvec[i].D[x][y-1]==0
588             //|| persvec[i].D[x+1][y]==0 || persvec[i].D[x-1][y]==0 || /*falls 2 Nachbarn frei sind*/
589             //persvec[i].D[x+1][y]==persvec[i].D[x][y+1]==0 || persvec[i].D[x-1][y]==persvec[i].D[x][y+1]==0 ||
590             //persvec[i].D[x-1][y]==persvec[i].D[x][y-1]==0 || persvec[i].D[x+1][y]==persvec[i].D[x][y-1]==0 || /*falls 3
591             //Nachbarn frei sind*/ persvec[i].D[x-1][y]==persvec[i].D[x][y+1]==0 || persvec[i].D[x][y-1]==0 ||
592             //persvec[i].D[x][y-1]==0 || persvec[i].D[x-1][y]==persvec[i].D[x][y+1]==0 ||
593             //persvec[i].D[x][y-1]==0 || persvec[i].D[x+1][y]==persvec[i].D[x][y+1]==0 ||
594             //persvec[i].D[x][y-1]==0 || persvec[i].D[x+1][y]==persvec[i].D[x][y+1]==0)
595             //{
596                 /*if (i==1)
597                 {
598                     cout << "-----" << endl;
599                     cout << "Zerfall findet bei Person " << i << " am Punkt: (x;y) = "
600                     << "(" << persvec[i].x << "," << persvec[i].y << ")" statt."<< endl;
601                     cout << "-----" << endl;
602                 }*/
603                 if(persvec[i].D[x][y] > 0){
604                     persvec[i].D[x][y]--;
605                 }
606                 else{
607                     persvec[i].D[x][y]++;
608                 }
609                 //grid_full=false;
610             //}
611             /*
612             //falls das d feld komplett voll ist, zerfällt es trz iwo am rand
613             if (grid_full==true)
614             {
615                 int x_rand = rand () %grid_width;
616                 int y_rand = rand () %grid_height;
617                 int which_side = rand() %4;
618                 switch (which_side)
619                 {
620                     case 0:
621                         persvec[i].D[0][y_rand]=0;
622                     case 1:
623                         persvec[i].D[x_rand][0]=0;
624                 }
625             }
626         }
627     }
628 }
```

```

616             case 2:
617                 persvec[i].D[grid_width-1][y_rand]=0;
618             case 3:
619                 persvec[i].D[x_rand][grid_height-1]=0;
620             }
621         */
622     }
623 }
624 }
625
626 }
627 }
628 }
629
630 void diffusion_dyn_f(vector <int> &propability_arr_diff, vector <person> &persvec, int xn, int yn,
631 int i, vector<obstacle> &obstvec, vector <int> &propability_arr_dec)
632 {
633     if(diffusion_param == 0){return;}
634     bool grid_full=true;
635     //gehe ganzen grundriss durch und überprüfe ob sich das d feld verteilen soll
636     for (int x=0; x< grid_width; x++)
637     {
638         for (int y=0; y<grid_height; y++)
639         {
640             for (int k=0; k<abs(persvec[i].D[x][y]); k++) //je stärker das d feld desto schneller
kann es sich verteilen
641             {
642                 //Verteilung des D-Felds
643                 //-----
644                 //schreibe wahrscheinlichkeitsarray
645                 for (int k=0; k <propability_arr_diff.size(); k++)
646                 {
647                     propability_arr_diff[k]=0;
648                 }
649                 if (diffusion_param!=0)
650                 {
651                     for (int k=0; k<diffusion_param ; k++)
652                     {
653                         propability_arr_diff[k]=1;
654                     }
655                 }
656                 //generiere zufallszahl und überprüfe ob sich das d feld verteilen soll
657                 int r_1=rand () %100;
658                 if (propability_arr_diff[r_1]==1) //verifikation: D feld soll sich verteilen
659                 {
660                     int which_cell=rand ()%4; //zu welcher zelle propagiert das d-feld?
661
662                     if (which_cell==0)
663                     {
664                         if (x-1!=xn && y!=yn)
665                         {
666                             if (can_d_field_be_here(x-1, y, obstvec)==true)
667                             {
668                                 if (persvec[i].D[x][y]>0)
669                                 {
670                                     persvec[i].D[x-1][y]++;
671                                     //persvec[i].D[x][y]--; //d feld verwischt
672                                 }
673                             else
674                             {
675                                 persvec[i].D[x-1][y]--;
676                                 //persvec[i].D[x][y]++; //d feld verwischt
677                             }
678                         }
679                         grid_full=false;

```

```

680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
}
else if (x+1!=xn && y!=yn)
{
    if (can_d_field_be_here(x+1, y, obstvec)==true)
    {
        if (persvec[i].D[x][y]>0)
        {
            persvec[i].D[x+1][y]++;
            //persvec[i].D[x][y]--; //d feld verwischt
        }
        else
        {
            persvec[i].D[x+1][y]--;
            //persvec[i].D[x][y]++; //d feld verwischt
        }
    }
    grid_full=false;
}
else if (x!=xn && y-1!=yn)
{
    if (can_d_field_be_here(x, y-1, obstvec)==true)
    {
        if (persvec[i].D[x][y]>0)
        {
            persvec[i].D[x][y-1]++;
            //persvec[i].D[x][y]--; //d feld verwischt
        }
        else
        {
            persvec[i].D[x][y-1]--;
            //persvec[i].D[x][y]++; //d feld verwischt
        }
    }
    grid_full=false;
}
else if (x!=xn && y+1!=yn)
{
    if (can_d_field_be_here(x, y+1, obstvec)==true)
    {
        if (persvec[i].D[x][y]>0)
        {
            persvec[i].D[x][y+1]++;
            //persvec[i].D[x][y]--; //d feld verwischt
        }
        else
        {
            persvec[i].D[x][y+1]--;
            //persvec[i].D[x][y]++; //d feld verwischt
        }
    }
    grid_full=false;
}
else if (which_cell==1)
{
    if (x+1!=xn && y!=yn)
    {
        if (can_d_field_be_here(x+1, y, obstvec)==true)
        {
            if (persvec[i].D[x][y]>0)
            {
                persvec[i].D[x+1][y]++;
                //persvec[i].D[x][y]--; //d feld verwischt
            }
            else
            {

```

```

746                     persvec[i].D[x+1][y]--;
747                     //persvec[i].D[x][y]++; //d feld verwischt
748                 }
749             }
750             grid_full=false;
751         }
752     else if (x!=xn && y-1!=yn)
753     {
754         if (can_d_field_be_here(x, y-1, obstvec)==true)
755         {
756             if (persvec[i].D[x][y]>0)
757             {
758                 persvec[i].D[x][y-1]++;
759                 //persvec[i].D[x][y]--; //d feld verwischt
760             }
761         else
762         {
763             persvec[i].D[x][y-1]--;
764             //persvec[i].D[x][y]++; //d feld verwischt
765         }
766     }
767     grid_full=false;
768 }
769 else if (x!=xn && y+1!=yn)
770 {
771     if (can_d_field_be_here(x, y+1, obstvec)==true)
772     {
773         if (persvec[i].D[x][y]>0)
774         {
775             persvec[i].D[x][y+1]++;
776             //persvec[i].D[x][y]--; //d feld verwischt
777         }
778     else
779     {
780         persvec[i].D[x][y+1]--;
781         //persvec[i].D[x][y]++; //d feld verwischt
782     }
783 }
784 grid_full=false;
785 }
786 else if (x-1!=xn && y!=yn)
787 {
788     if (can_d_field_be_here(x-1, y, obstvec)==true)
789     {
790         if (persvec[i].D[x][y]>0)
791         {
792             persvec[i].D[x-1][y]++;
793             //persvec[i].D[x][y]--; //d feld verwischt
794         }
795     else
796     {
797         persvec[i].D[x-1][y]--;
798         //persvec[i].D[x][y]++; //d feld verwischt
799     }
800 }
801 grid_full=false;
802 }
803 }
804 else if (which_cell==2)
805 {
806     if (x!=xn && y-1!=yn)
807     {
808         if (can_d_field_be_here(x, y-1, obstvec)==true)
809         {
810             if (persvec[i].D[x][y]>0)
811             {

```

```

812                     persvec[i].D[x][y-1]++;
813                     //persvec[i].D[x][y]--; //d feld verwischt
814                 }
815             else
816             {
817                 persvec[i].D[x][y-1]--;
818                 //persvec[i].D[x][y]++; //d feld verwischt
819             }
820         }
821         grid_full=false;
822     }
823     else if (x!=xn && y+1!=yn)
824     {
825         if (can_d_field_be_here(x, y+1, obstvec)==true)
826         {
827             if (persvec[i].D[x][y]>0)
828             {
829                 persvec[i].D[x][y+1]++;
830                 //persvec[i].D[x][y]--; //d feld verwischt
831             }
832             else
833             {
834                 persvec[i].D[x][y+1]--;
835                 //persvec[i].D[x][y]++; //d feld verwischt
836             }
837         }
838         grid_full=false;
839     }
840     else if (x-1!=xn && y!=yn)
841     {
842         if (can_d_field_be_here(x-1, y, obstvec)==true)
843         {
844             if (persvec[i].D[x][y]>0)
845             {
846                 persvec[i].D[x-1][y]++;
847                 //persvec[i].D[x][y]--; //d feld verwischt
848             }
849             else
850             {
851                 persvec[i].D[x-1][y]--;
852                 //persvec[i].D[x][y]++; //d feld verwischt
853             }
854         }
855         grid_full=false;
856     }
857     else if (x+1!=xn && y!=yn)
858     {
859         if (can_d_field_be_here(x+1, y, obstvec)==true)
860         {
861             if (persvec[i].D[x][y]>0)
862             {
863                 persvec[i].D[x+1][y]++;
864                 //persvec[i].D[x][y]--; //d feld verwischt
865             }
866             else
867             {
868                 persvec[i].D[x+1][y]--;
869                 //persvec[i].D[x][y]++; //d feld verwischt
870             }
871         }
872         grid_full=false;
873     }
874 }
875 else if (which_cell==3)
876 {
877     if (x!=xn && y+1!=yn)

```

```

878
879     {
880         if (can_d_field_be_here(x, y+1, obstvec)==true)
881         {
882             if (persvec[i].D[x][y]>0)
883             {
884                 persvec[i].D[x][y+1]++;
885                 //persvec[i].D[x][y]--; //d feld verwischt
886             }
887             else
888             {
889                 persvec[i].D[x][y+1]--;
890                 //persvec[i].D[x][y]++; //d feld verwischt
891             }
892             grid_full=false;
893         }
894     else if (x-1!=xn && y!=yn)
895     {
896         if (can_d_field_be_here(x-1, y, obstvec)==true)
897         {
898             if (persvec[i].D[x][y]>0)
899             {
900                 persvec[i].D[x-1][y]++;
901                 //persvec[i].D[x][y]--; //d feld verwischt
902             }
903             else
904             {
905                 persvec[i].D[x-1][y]--;
906                 //persvec[i].D[x][y]++; //d feld verwischt
907             }
908             grid_full=false;
909         }
910     else if (x+1!=xn && y!=yn)
911     {
912         if (can_d_field_be_here(x+1, y, obstvec)==true)
913         {
914             if (persvec[i].D[x][y]>0)
915             {
916                 persvec[i].D[x+1][y]++;
917                 //persvec[i].D[x][y]--; //d feld verwischt
918             }
919             else
920             {
921                 persvec[i].D[x+1][y]--;
922                 //persvec[i].D[x][y]++; //d feld verwischt
923             }
924             grid_full=false;
925         }
926     else if (x!=xn && y-1!=yn)
927     {
928         if (can_d_field_be_here(x, y-1, obstvec)==true)
929         {
930             if (persvec[i].D[x][y]>0)
931             {
932                 persvec[i].D[x][y-1]++;
933                 //persvec[i].D[x][y]--; //d feld verwischt
934             }
935             else
936             {
937                 persvec[i].D[x][y-1]--;
938                 //persvec[i].D[x][y]++; //d feld verwischt
939             }
940         }
941     }
942     grid_full=false;

```

```

944         }
945     }
946   }
947 }
948 }
949 }
950 }
951 if (grid_full==true && diffusion_param !=0) //sollte nicht passieren, vermeidet fehler
952 {
953   decay_dyn_f(propability_arr_dec, persvec, i);
954 }
955 }
956 }
957 bool can_d_field_be_here (int qx, int qy, vector<obstacle> &obstvec)
958 {
959   //delivers true, if the d field could stay at (x,y)/ could move to this cell
960
961   //###cells out of borders arent available for d feld:
962   if ((qy >= grid_height) || (qx >= grid_width))
963   {
964     return false;
965   }
966   if ((qy < 0) || (qx < 0))
967   {
968     return false;
969   }
970
971   //###cells filled by an obstacle arent available for d feld:
972   for(int i = 0; i < quantity_obstacles; i++)
973   {
974     if((obstvec[i].x == qx) && (obstvec[i].y == qy))
975     {
976       return false;
977     }
978   }
979   return true;
980 }
981
982
983
984 void set_panic_par(vector <person> &persvec, int i, int iteration)
985 {
986   if(panik_aktiviert == false){return;} // wenn der Panik Parameter nicht beachtet werden soll, wird
diese Funktion abgebrochen
987   if (iteration==0)
988   {
989     persvec[i].iterat_val=0;
990   }
991
992   //rising panic parameter
993   if (persvec[i].had_a_conflict==true)
994   {
995     persvec[i].panic_par++;
996     persvec[i].iterat_val=persvec[i].iteration;
997     /*r=0;
998     g=255;
999     b=0;*/
1000   }
1001
1002   //zerfall vom panikparameter
1003   if (persvec[i].iterat_val+2 <= persvec[i].iteration && persvec[i].had_a_conflict==false && persvec
[i].evacuated==false && persvec[i].panic_par >= 0)
1004   {
1005     persvec[i].panic_par--;
1006     if(panic_par < panik_schwelle && r==255){
1007       r=0;

```

```

1008         g=250;
1009         b=0;
1010     }
1011 }
1012 }
1013
1014 /////panikschwelle -> diffusion
1015 if (persvec[i].panic_par>=panik_schwelle)
1016 {
1017     for (int k=0; k<persvec.size(); k++)
1018     {
1019         if (persvec[k].x==persvec[i].x-1 && persvec[k].y==persvec[i].y)
1020         {
1021             persvec[k].panic_par++;
1022         }
1023         if (persvec[k].x==persvec[i].x+1 && persvec[k].y==persvec[i].y)
1024         {
1025             persvec[k].panic_par++;
1026         }
1027         if (persvec[k].x==persvec[i].x && persvec[k].y==persvec[i].y-1)
1028         {
1029             persvec[k].panic_par++;
1030         }
1031         if (persvec[k].x==persvec[i].x-1 && persvec[k].y==persvec[i].y+1)
1032         {
1033             persvec[k].panic_par++;
1034         }
1035     r=255;
1036     g=0;
1037     b=0;
1038 }
1039 }
1040 }
1041
1042 void print_D()
1043 {
1044     cout << "-----" << endl;
1045     for(int j = 0; j < grid_height; j++){
1046         for(int i = 0; i < grid_width; i++){
1047             if (D[i][j] > 9999){cout << " " << (int)D[i][j] << ";" ;}
1048             else if (D[i][j] > 999){cout << " " << (int)D[i][j] << ";" ;}
1049             else if (D[i][j] > 99){cout << " " << (int)D[i][j] << ";" ;}
1050             else if (D[i][j] > 9){cout << " " << (int)D[i][j] << ";" ;}
1051             else {cout << " " << D[i][j] << ";" ;}
1052         }
1053     cout << endl;
1054     }
1055 }
1056 }
1057
1058
1059 // ##### Static field S
1060 double k_S = 1;
1061 vector <double> w_S;// wie sehr kennt die Person die verschiedenen Eingänge; Eintrag ist zwischen 0,1
1062 double S[grid_width][grid_height];
1063
1064 void set_w_S(double w){// der Wissenstand der Personen wird für alle Ausgänge gleich groß gewählt (so
groß wie w)
1065     w_S.resize(quantity_destinations);
1066     for(int i = 0; i < quantity_destinations; i++){
1067         w_S[i] = w;
1068     }
1069 }
1070 void set_w_S(int quantity_known_dest, bool previously_set = false){//legt den anfänglicher
Wissensstand der Person über die Ausgänge fest, Parameter w_S wird für eine Anzahl(quantity_known_dest) von
Zielen zufällig gewählt, wenn previously set == false ist (Ausnahmefall ist, wenn quantity_known_dest == 1 ist,

```

```

dann ist w_S = 1; previously_set = true wird nur intern benutzt
1071     w_S.resize(quantity_destinations);
1072     for(int i = 0; i < quantity_destinations; i++){
1073         if(previously_set == false)
1074             w_S[i] = 0;
1075     }
1076     //Fehlervermeidung:
1077     if(quantity_known_dest > quantity_destinations){
1078         cout << "Fehler in set_w_S(int) Anzahl der bekannten Ziele ist größer, als die Anzahl der
Ziele!!" << endl;
1079         return;
1080     }
1081     //Füllt vector selected_dest mit Nummern, die den Zielen zugeordnet werden, die schon einen Wert
w_S erhalten haben:
1082     vector<int> selected_dest;
1083     for(int i = 0; i < quantity_known_dest; i++){
1084         int r = (rand() % quantity_destinations);
1085         //jede Nummer darf nur ein mal vorkommen:
1086         bool used = false;
1087         for(int j = 0; j < selected_dest.size(); j++){
1088             if(selected_dest[j] == r){
1089                 used = true;
1090             }
1091         }
1092         if(used == false && (previous_set == false || (previous_set == true && w_S[r] == 0))){
1093             //Füllen von w_S zum ausgewählten Ziel mit einer Zufälligen Zahl von (0 bis 1], außer, wenn
nur ein Ausgang bekannt ist (in diesem Fall ist w_S immer 1)
1094             if(quantity_known_dest == 1){
1095                 w_S[r] = 1.0;
1096             }
1097             else{
1098                 w_S[r] = (double) (rand() % 10) / 10 + 0.1;
1099             }
1100             //cout << "w_S[r] ist:" << w_S[r] << " mit r =" << r << endl;
1101             selected_dest.push_back(r);
1102         }
1103         else{
1104             i--;//könnte eine Endlosschleife verursachen -> nicht so schön ; aber geht
1105         }
1106     }
1107 }
1108 void set_w_S(bool prefer_a_dest, int quantity_preferred_dest, int *preferred_dest, int
quantity_known_dest){//legt den anfänglichen Wissensstand der Person über die Ausgänge fest; preferierte Ziele
werden bevorzugt nach der Festlegung angesteuert, qpd ist die Anzahl der übergebenen Ziele
1109     w_S.resize(quantity_destinations);
1110     // set all values of w_S = 0:
1111     for(int i = 0; i < quantity_destinations; i++){
1112         w_S[i] = 0;
1113     }
1114     if (prefer_a_dest == true){
1115         if (quantity_known_dest < quantity_preferred_dest){//Fehlervermeidung
1116             cout << "Fehler in set_w_S; die Anzahl der Ziele, die nicht 0 sein sollen ist zu klein!"<< endl;
1117             return;
1118         }
1119         for(int i = 0; i < quantity_preferred_dest; i++){
1120             w_S[preferred_dest[i]] = 2;
1121         }
1122         set_w_S(quantity_known_dest-quantity_preferred_dest, true);
1123     }
1124 }
1125 }
1126 else{
1127     set_w_S(quantity_known_dest);
1128 }
1129

```

```

1130
1131
1132
1133     }
1134     void renew_w_S_and_S(vector<destination> &destvec, bool foreign_call){// erneuert die Einträge von
w_S, wenn bestimmte Umstände eintreten
1135         //wenn sich die Person sehr nahe an einem Ausgang befindet bekommt der Wert w_S, der für das
Wissen über diesen Ausgang steht, einen sehr hohen Wert, da die Person den Ausgang sieht o.Ä.
1136         if(foreign_call == false){//Bedingung, da sonst die Analyse verfälscht wird
1137             int r_influence_sphere = 5;// legt fest, ab wann die Person den Ausgang sehen kann
1138             for(int i = 0; i < quantity_destinations; i++){
1139                 if(destvec[i].x > x - r_influence_sphere && destvec[i].x < x + r_influence_sphere){
1140                     if(destvec[i].y > y - r_influence_sphere && destvec[i].y < y + r_influence_sphere){
1141                         w_S[i] = 2;
1142                         panic_par = 0;
1143                     }
1144                 }
1145             }
1146         }
1147         if((foreign_call == false || take_which_exit == "near") && corridor_conditions == false){
1148             set_S_normal(destvec);
1149         }
1150     }
1151 }
1152 void print_w_S(){
1153     cout << endl << "print_w_S:" << endl;
1154     for(int i = 0; i < quantity_destinations; i++){
1155         cout << w_S[i] << ",";
1156     }
1157     cout << endl;
1158 }
1159 void set_S_normal(vector<destination> &destvec){//Addiere alle S_k Arrays der einzelnen destinations
zum S Array hinzu; dies verläuft nach Gewichtung
1160     double max_S = 0;
1161     //setze alle Einträge von S auf 0:
1162     for(int xi = 0; xi < grid_width; xi++){
1163         for(int yi = 0; yi < grid_height; yi++){
1164             S[xi][yi] = 0;
1165         }
1166     }
1167
1168     //Füllt die Einträge von S
1169     for(int l = 0; l < quantity_destinations; l++){
1170         for(int xi = 0; xi < grid_width; xi++){
1171             for(int yi = 0; yi < grid_height; yi++){
1172                 S[xi][yi] = S[xi][yi] + destvec[l].get_S_k(xi,yi) * w_S[l];
1173                 if(S[xi][yi] > max_S){
1174                     max_S = S[xi][yi];
1175                 }
1176             }
1177         }
1178     }
1179 }
1180 void set_S_corridor( vector<person> &persvec, vector<destination> &destvec, vector<obstacle> &obstvec
){
1181     //setze alle Einträge von S auf 0:
1182     for(int xi = 0; xi < grid_width; xi++){
1183         for(int yi = 0; yi < grid_height; yi++){
1184             S[xi][yi] = 0;
1185         }
1186     }
1187     //Wenn diese Funktion ausgeführt wird ein waagerechter Korridor simuliert, der mehrere Ziele am
Anfang und am Ende des Korridors besitzt
1188     //Geht die Ziele durch und merkt sich die größte x Koordinate
1189     int max_x = 0;
1190     for(int i = 0; i < destvec.size(); i++){

```

```

1191         if(destvec[i].x > max_x){
1192             max_x = destvec[i].x;
1193         }
1194     }
1195
1196     if(destvec[numb_selected_dest].x == max_x){//Dies ist das rechte Ziel, also ist die Formel S[x][y]
1197     =
1198         // Aendert Farbe der Personen für eine bessere Übersicht
1199         b = 0;
1200         g = 50;
1201         r = 255;
1202         for(int i = 0; i < max_x; i++){
1203             for(int j = 0; j < grid_height; j++){
1204                 if(could_I_go_to(i,j,obstvec,persvec) || is_there_a_person_on(i,j,persvec)){//Abfrage
1205                     ob eine Person auf das Feld gehen könnte
1206                         S[i][j] = i;
1207                     }
1208                 }
1209             else{//Es bleibt nur das linke Ziel übrig, also ist die Formel S[x][y] = x_max - x
1210                 b = 255;
1211                 g = 0;
1212                 for(int i = 0; i < max_x; i++){
1213                     for(int j = 0; j < grid_height; j++){
1214                         if(could_I_go_to(i,j,obstvec,persvec) || is_there_a_person_on(i,j,persvec)){//Abfrage
1215                             ob eine Person auf das Feld gehen könnte
1216                             S[i][j] = max_x - i;
1217                         }
1218                     }
1219                 }
1220             }
1221         }
1222     void print_S(int width = grid_width, int height = grid_height){
1223         cout << "-----" << endl;
1224         for(int j = 0; j < height; j++){
1225             for(int i = 0; i < width; i++){
1226                 if (S[i][j] > 9999){cout << " " << (int)S[i][j] << ";" ;}
1227                 else if (S[i][j] > 999){cout << " " << (int)S[i][j] << ";" ;}
1228                 else if (S[i][j] > 99){cout << " " << (int)S[i][j] << ";" ;}
1229                 else if (S[i][j] > 9){cout << " " << (int)S[i][j] << ";" ;}
1230                 else {cout << " " << S[i][j] << ";" ;}
1231             }
1232         cout << endl;
1233     }
1234 }
1235 }
1236
1237 // ##### Transition matrix
1238 long double T[3][3];
1239
1240 void set_T(vector<obstacle> &obstvec,vector<person> &persvec, char movement_mode = 's'){
1241     //in der Funktion wird zwischen dem Erstellen des T Arrays für eine parallele Update-Regel und einer
1242     //sequentiellen Update-Regel unterschieden
1243     //Bei der parallelen Update-Regel wird nicht abgefragt ob sich eine Person auf dem angefragten Fehld
1244     befindet
1245     //füllt Einträge:
1246     //Alle Felder bekommen Wert 0; nicht benutzte Felder (nach Neumann Nachbarschaft) bleiben 0:
1247     for (int i = 0; i < 3; i++){
1248         for(int j = 0; j < 3; j++){
1249             T[i][j] = 0;
1250         }
1251     }
1252     //cout << "hier müsste alles null sein:" << endl;
1253     //print_T();

```

```

1252     //Eintrag oben:
1253     //cout << "oben ?" << could_I_go_to(x,y - 1,obstvec) << endl;
1254     if(could_I_go_to(x,y - 1,obstvec,persvec)){ // entweder sequentieller Ablauf: dann could I go to:
bei parallelen ist es egal ob auf dem Feld gerade eine andere Person steht
1255         T[1][0] = expl(k_S * S[x][y - 1] + (k_D+panic_par) * D[x][y - 1]);
1256     }
1257     //Eintrag rechts:
1258     //cout << "rechts ?" << could_I_go_to(x + 1,y,obstvec) << endl;
1259     if(could_I_go_to(x + 1,y,obstvec,persvec)){
1260         T[2][1] = expl(k_S * S[x + 1][y] + (k_D+panic_par) * D[x + 1][y]);
1261     }
1262     //Eintrag unten:
1263     //cout << "unten ?" << could_I_go_to(x,y+1,obstvec) << endl;
1264     if(could_I_go_to(x,y + 1,obstvec,persvec)){
1265         T[1][2] = expl(k_S * S[x][y + 1] + (k_D+panic_par) * D[x][y + 1]);
1266     }
1267     //Eintrag links:
1268     //cout << "unten ?" << could_I_go_to(x,y+1,obstvec) << endl;
1269     if(could_I_go_to(x - 1,y,obstvec,persvec)){
1270         T[0][1] = expl(k_S * S[x - 1][y] + (k_D+panic_par) * D[x - 1][y]);
1271     }
1272     //mitte:
1273     //cout << "hier bleiben ?" << could_I_go_to(x,y,obstvec) << endl;
1274     T[1][1] = expl(k_S * S[x][y] + (k_D+panic_par) * D[x][y]);
1275
1276
1277     //Überprüfung ob die Einträge des Feldes zu groß sind und deswegen T fehlerhaft erstellt wird:
1278     for (int i = 0; i < 3; i++){
1279         for(int j = 0; j < 3; j++){
1280             if (isinf(T[i][j])){
1281                 cout << "Fehler beim Erstellen der Matrix T aufgetreten ! Die Groesse der Eintraege uebersteigt die maximale Groesse des long double Zahlentyps." << endl << "Versuchen sie die Simulation mit einem kleineren Feld zu wiederholen." << endl;
1282             }
1283         }
1284     }
1285
1286     //cout << "T befor normalization:" << endl;
1287     //print_T();
1288     //Normalisierung der T-Matrix:
1289     //Finden der Summe der Einträge von T:
1290     long double sum_T_entries = 0;
1291     for (int i = 0; i < 3; i++){
1292         for(int j = 0; j < 3; j++){
1293             sum_T_entries = sum_T_entries + T[i][j];
1294         }
1295     }
1296     //Normalisierung durchführen:
1297     for(int i = 0; i < 3; i++){
1298         for(int j = 0; j < 3; j++){
1299             T[i][j] = T[i][j] / sum_T_entries;
1300         }
1301     }
1302     //cout << "T after normalization:" << endl;
1303     //print_T();
1304 }
1305 void print_T(){
1306     cout << "-----" << endl;
1307     cout << "Koordinaten:" ;
1308     print_coords();
1309     cout << "transition matrix T:" << endl;
1310     for (int i = 0; i < 3; i++){
1311         for(int j = 0; j < 3; j++){
1312             cout << T[i][j] << " ";
1313         }
1314     cout << endl;

```

```

1315         }
1316         cout << "-----" << endl;
1317     }
1318     double get_T(int qx, int qy){ // gibt Eintrag der 3x3 Matrix aus
1319         if(qx > 2 || qy > 2 || qx < 0 || qy < 0){
1320             cout << "Fehler aufgetreten bei get_T!!" << endl;
1321         }
1322         return T[qx][qy];
1323     }
1324
1325 // ##### desired coordinates; are used, when update rule is parallel
1326     int desired_x;
1327     int desired_y;
1328     char desired_direction;
1329     int number_of_conflicts;
1330     bool wins_conflict;
1331     bool had_a_conflict; //wird benutzt um herauszufinden ob bei einer Bewegung der "friction" Parameter
angewendet werden muss
1332
1333 // ##### time measurement for the analysis of movement of the person
1334     double time_start;
1335     double time_end;
1336     double evacuation_time = 0;
1337     int iteration;
1338     int iteration_when_evacuated;
1339     bool evacuated;
1340
1341     void start_time_measurement(){
1342         time_start = clock()/CLOCKS_PER_SEC;
1343     }
1344     void end_time_measurement(){
1345         time_end = clock()/CLOCKS_PER_SEC;
1346         evacuation_time = time_end - time_start;
1347     }
1348
1349 // ##### wenn "reject_other_D_fields" aktiviert ist, werden folgende Variablen benutzt:
1350     int numb_selected_dest; // gibt das Ziel an welches die Person kennt und ansteuern wird
1351     //last_movement_direction wird benutzt und steht bei den Koordinaten
1352
1353
1354
1355
1356
1357 private:
1358     bool followed_the_pers_my_S(person qpers, int iterations_done){ //ist wahr, wenn die gefragte Person
mit ihrem letzten(iterations_done == 1) bzw. vorletzten(iterations_done == 2) Schritt dem eigenen Potentialfeld
gefollgt ist und in die Richtung des eigenen Ziels geht.
1359         int nx, ny, ax, ay;
1360         if(iterations_done == 1){
1361             nx = qpers.x;
1362             ny = qpers.y;
1363             ax = qpers.ax;
1364             ay = qpers.ay;
1365         }
1366         else if(iterations_done == 2){
1367             nx = qpers.ax;
1368             ny = qpers.ay;
1369             ax = qpers.aax;
1370             ay = qpers.aay;
1371         }
1372         else{
1373             cout << "Fehler in followed_the_pers_my_S()" << endl;
1374         }
1375         if((k_S * S[nx][ny]) >= (k_S * S[ax][ay]) && qpers.evacuated == false && iteration > 1)
1376         {
1377             if ((iterations_done==2 && followed_the_pers_my_S(qpers, 1)==true) || iterations_done==1)

```

```

1378         {
1379             return true;
1380         }
1381     }
1382     return false;
1383 }
1384 };
1385
1386
1387 class conflict{
1388 public:
1389     conflict(){
1390     }
1391     conflict(int nx, int ny, vector<int> &cp, vector<person> &persvec){
1392         x = nx;
1393         y = ny;
1394         conflict_partner = cp;
1395         rise_number_of_conflicts_of_persons(persvec);
1396         set_C(persvec);
1397         number_of_winner = who_winnns_conflict();
1398     }
1399 }
1400
1401 //Ort des Konfliktes:
1402 int x;
1403 int y;
1404
1405 //Nummern der Personen im Konflikt:
1406 vector<int> conflict_partner;
1407
1408 //Gewinner des Konfliktes:
1409 int number_of_winner;
1410
1411 //Konfliktmatrix
1412 double C[3][3];
1413
1414 //Hilfsmatrix, in der die Nummern der Personen stehen, die am Konflikt teilnehmen (mit relativer
Position zum Ort des Konflikts
1415 double C_pers_numb[3][3];
1416
1417
1418
1419 void print_C(){
1420     cout << "-----C-----" << endl;
1421     cout << "An der Stelle: " << x << ";" << y << endl;
1422     for(int i = 0; i < 3; i++){
1423         for(int j = 0; j < 3; j++){
1424             cout << C[j][i] << " , ";
1425         }
1426         cout << endl;
1427     }
1428     cout << "-----C-----" << endl;
1429 }
1430 void print_C_pers_numb(){
1431     cout << "-----Cpers-----" << endl;
1432     cout << "An der Stelle: " << x << ";" << y << endl;
1433     for(int i = 0; i < 3; i++){
1434         for(int j = 0; j < 3; j++){
1435             cout << C_pers_numb[j][i] << " , ";
1436         }
1437         cout << endl;
1438     }
1439     cout << "-----Cpers-----" << endl;
1440 }
1441 double get_C(int x, int y){
1442     return C[x][y];

```

```

1443     }
1444     double get_C_pers_numb(int x, int y){
1445         return C_pers_numb[x][y];
1446     }
1447
1448 private:
1449     void set_C(vector<person> &persvec){
1450
1451         //Setzt Einträge der Matrizen auf 0
1452         for(int k = 0; k < 3; k++){
1453             for(int l = 0; l < 3; l++){
1454                 C[k][l] = 0;
1455                 C_pers_numb[k][l] = 0;
1456             }
1457         }
1458
1459         //Füllt Einträge von C, die Werte von der Transitionmatrix, die dazu geführt haben, dass sich die
1460         Person auf das Feld Conflict.x,Conflict.y bewegen will wird in die Konfliktmatrix C eingetragen
1461         for(int i = 0; i < conflict_partner.size(); i++){
1462             C[persvec[conflict_partner[i]].x - x + 1][persvec[conflict_partner[i]].y - y + 1] = persvec[
1463             conflict_partner[i]].get_T(x - persvec[conflict_partner[i]].x + 1, y - persvec[conflict_partner[i]].y + 1);
1464             C_pers_numb[persvec[conflict_partner[i]].x - x + 1][persvec[conflict_partner[i]].y - y + 1] =
1465             conflict_partner[i];
1466         }
1467         //Normalisierung der C-Matrix:
1468         //Finden der Summe der Einträge von C:
1469         double sum_C_entries = 0;
1470         for (int i = 0; i < 3; i++){
1471             for(int j = 0; j < 3; j++){
1472                 sum_C_entries = sum_C_entries + C[i][j];
1473             }
1474         }
1475         //Normalisierung durchführen:
1476         for(int i = 0; i < 3; i++){
1477             for(int j = 0; j < 3; j++){
1478                 C[i][j] = C[i][j] / sum_C_entries;
1479             }
1480         }
1481         int who_wins_conflict(){
1482             double r = rand() % 1000 / 1000.;
1483             if(r < get_C(1,0)){
1484                 return get_C_pers_numb(1,0);
1485             }
1486             else if(r < (get_C(1,0) + get_C(2,1))){
1487                 return get_C_pers_numb(2,1);
1488             }
1489             else if(r < (get_C(1,0) + get_C(2,1) + get_C(1,2))){
1490                 return get_C_pers_numb(1,2);
1491             }
1492             else if(r < (get_C(1,0) + get_C(2,1) + get_C(1,2) + get_C(0,1))){
1493                 return get_C_pers_numb(0,1);
1494             }
1495             else{
1496                 return get_C_pers_numb(1,1);
1497             }
1498         void rise_number_of_conflicts_of_persons(vector<person> &persvec){
1499             for(int i = 0; i < conflict_partner.size(); i++){
1500                 persvec[conflict_partner[i]].number_of_conflicts++;
1501                 persvec[conflict_partner[i]].had_a_conflict = true; // für die richtige Anwendung des friction
Parameters
1502             }
1503         };
1504

```