Der Einfluss der Geometrie auf Crowd-Desaster-Szenarien: Untersuchungen basierend auf Erweiterung der Open-Source Crowd-Simulation Software PEDSIM um eine neuartige Dichtefunktion

Bachelorarbeit zur Erlangung des Bachelor-Grades
Bachelor of Science im Studiengang Medieninformatik
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Tim Kurtz
Matrikel-Nr.:	011 089 206
	tim.kurtz@smail.th-koeln.de

eingereicht bei:	Prof. Dr. Wolfgang Konen
Zweitgutachter:	Prof. Dr. Lutz Köhler

Gummersbach, 10. Februar 2021

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Rechtsverbindliche Unterschrift

Abstract

Seit dem Unglück bei der Loveparade 2010 sind Crowd-Simulation bei der vorherigen Planung von Massenveranstaltungen besonders wichtig geworden. Sie können helfen, dass ähnliche Unglücke in Zukunft nicht mehr passieren.

Ziel der Arbeit ist es, die Crowd-Simulation PEDSIM um eine neuartige Dichtefunktion zu erweitern und mit dieser ein Crowd-Desaster-Szenario zu simulieren, um herauszufinden unter welchen Umständen gefährliche Dichtewerte entstehen.

PEDSIM wurde um eine Klasse ergänzt, deren Objekte die Agentendichte messen. Die Klasse wurde genutzt um, an mehreren Positionen in einem konstruierten Flaschenhalsszenario, die Dichte zu messen.

Die Arbeit konnte zeigen, dass bei Simulationen mit 1000 Agenten innerhalb des direkten Eingangsbereiches des Flaschenhalses ab einer Flaschenhalshöhe von $f = 45$ gefährliche Dichtewerte entstehen.

Schlagwörter: PEDSIM, Crowd-Simulation, Crowd-Desaster, Multi-Agenten-Simulation

Inhaltsverzeichnis

Erklärung **I**

Kurzfassung/Abstract **II**

Einleitung **1**

1 Grundlagenteil **3**
1.1 Crowd-Desaster . . . 3
1.2 Ursache des Crowd-Desaster . . . 3
1.3 Simulatoren . . . 3
1.3.1 PEDSIM . . . 3
1.3.2 Vadere . . . 4
1.3.3 Hallweger Bachelorarbeit . . . 4

2 Beschreibung von PEDSIM **5**
2.1 Einleitung . . . 5
2.2 Grundlegende Klassen PEDSIM . . . 5
2.3 XML-Szenario . . . 6
2.3.1 Beispielszenario . . . 6
2.3.2 XML-Elemente des Szenarios . . . 7
2.3.3 Beschreibung des Szenarios . . . 8
2.4 Simulationsgrundlage . . . 9
2.4.1 „Social Force Model“ . . . 9
2.4.2 Umsetzung des „Social Force Model“ . . . 9
2.4.3 Quadtree . . . 10
2.5 Erweiterung von PEDSIM mit einer graphische Darstellung der Dichte . . . 10

3 Erstellung einer neuartigen Dichtefunktion **12**
3.1 Betrachtung der Methode getNeighbors() . . . 12
3.2 Erstellung einer eigenen Detektorklasse . . . 12
3.2.1 Grundlagen . . . 12
3.2.2 Konstruktoren . . . 13
3.2.3 Polygon-Suchfelder . . . 13
3.2.4 Detector::count() . . . 13
3.2.5 Detector::draw() . . . 13
3.2.6 Detector::getDensity() . . . 14
3.3 Fazit . . . 14

4 Ausgabe der Dichtewerte **15**
4.1 Ausgabe per GUI und Datei . . . 15
4.2 Fazit . . . 15

5 Erstellung des Testszenarios **16**
5.1 Konstruktion des Flaschenhalses . . . 16

5.2 Positionierung der Agenten und Wegpunkte 17
5.3 Positionierung der Messflächen 19

6 Durchführung 21
6.1 Grundsätzliche Vorgehensweise 21
6.2 Probedurchführungen 21
6.2.1 Beschreibung des Videos DF100.mp4 22
6.2.2 Beschreibung des Videos DF50.mp4 23
6.2.3 Beschreibung des Videos DF25.mp4 24
6.2.4 Fazit der Probedurchführungen 25
6.3 Weitere Vorgehensweise 25

7 Diskussion 26
7.1 Vergleich der Dichtewerte von DF25, DF50 und DF100 26
7.2 Betrachtung der Dichtewerte bei DF25 28
7.3 Betrachtung der Dichtewerte des Eingangsbereiches bei DF25 30
7.4 Zwischenfazit 31
7.5 Vergleich der maximalen Dichtewerte aller Durchführungen 32
7.6 Vergleich der maximalen Dichtewerte für $f = 10$ bis $f = 50$ 34
7.7 Fazit der Diskussion 35

8 Fazit 36

9 Ausblick 37
9.1 Dichtemessung 37
9.2 Testdurchführungen 37

Literaturverzeichnis 38

Anhang 39

Tabellenverzeichnis

Tabelle 1: Agentendichte einfärben 10
Tabelle 2: Grenzwerte für Einfärbung der Dichte 14
Tabelle 3: Koordinaten der Flaschenhalskonstruktion 17
Tabelle 4: Übersicht der Messflächen 57

Abbildungsverzeichnis

Abbildung 1: Screenshot aus Beispielszenario.mp4 8
Abbildung 2: Einfärbung der Quadtrees in Relation zur Agentendichte 11
Abbildung 3: Flaschenhalskonstruktion 16
Abbildung 4: Positionierung von Agentenursprung und Wegpunkte 18
Abbildung 5: Namen der Messflächen 19
Abbildung 6: Deutliche Hufeisenbildung bei Screenshot von DF100.mp4 22
Abbildung 7: Verringerte Hufeisenbildung bei Screenshot von DF50.mp4 . . . 23
Abbildung 8: Screenshot von DF25.mp4 24
Abbildung 9: Plot der Dichte in Relation zu den Timesteps t bei Flaschenhalshöhe 100, 50 und 25 26
Abbildung 10: Plot der Dichte in Relation zu den Timesteps t bei Flaschenhalshöhe 25 28
Abbildung 11: Plot der Dichte des Eingangsbereiches in Relation zu Timesteps t bei Flaschenhalshöhe 25 30
Abbildung 12: Screenshot des Videos CloseUpDF25.mp4 bei $t = 120$ 31
Abbildung 13: Plot der Maximalwerte der Dichte in Relation zur Flaschenhalshöhe 10 bis 100 32
Abbildung 14: Plot der Maximalwerte der Dichte in Relation zur Flaschenhalshöhe 10 bis 50 34
Abbildung 15: Bearbeiteter Screenshot der Ausführung von getNeighbors(0,0,1) 40
Abbildung 16: Screenshot der Ausführung von getNeighbors(0,0,5) 42
Abbildung 17: Von Quadratsform zur Rechtecksform 44
Abbildung 18: Kleinstes umgebendes Quadrat bei $w > h$ 46
Abbildung 19: Bildung polygoner Messfläche per Strahlensatz 55

Listings

Listing 1 Beispielszenario.xml 6
Listing 2 Ausgabe von getNeighbors(0,0,1) 39
Listing 3 Methode getNeighbors() in libpedsim/ped_scene.cpp 40
Listing 4 Geschützte Methode getNeighbors() in libpedsim/ped_scene.cpp . . 41
Listing 5 Methode intersects() in libpedsim/ped_trees.cpp 42
Listing 6 Erstellte Klasse Detector in demoapp/src/detector.cpp 46
Listing 7 Ausschnitt aus demoapp/src/control.cpp 50
Listing 8 Erstellte Deklarationen in demoapp/src/stats.h 51
Listing 9 Erstellte Methode inputStringToUi() in demoapp/src/stats.cpp . . . 51
Listing 10 Erweiterung in demoapp/src/control.cpp 51
Listing 11 Erweiterung in demoapp/src/scene.cpp 51
Listing 12 Erstellte Methode buildOutputFile() in scene.cpp 52
Listing 13 Erweiterung der Methode moveAllAgents() in demoapp/src/scene.cpp 53
Listing 14 Final erstellte Methode outPut() in demoapp/src/scene.cpp 53
Listing 15 Generierte Konstruktoraufrufe in demoapp/src/scene.cpp 58
Listing 16 Generierte XML-Datei 60

Einleitung

Fast genau zehn Jahre nach dem Unglück bei der Loveparade in Duisburg 2010 wurde das langjährige Gerichtsverfahren eingestellt. Damals starben 21 Personen und 652 wurden verletzt [2]. Geblieben ist die Verantwortung, zukünftige Veranstaltungen sicherer zu machen, damit sich solche Unglücke nicht wiederholen. Um derartige Veranstaltungen sicherer zu gestalten, wird vor der Stattfindung mit Simulationsprogrammen gearbeitet.

In dieser Arbeit wird das Demoprogramm der Massensimulationsprogrammbibliothek PEDSIM um eine neuartige Dichtemessung erweitert. Als Crowd-Desaster-Szenario wird beispielhaft ein Flaschenhalsszenario konstruiert und mit verschiedenen Flaschenhalshöhen simuliert. Dabei werden mit der erstellten Dichtemessung Werte erhoben und diese anschließend diskutiert.

Es werden die folgenden Forschungsfragen aufgestellt:

F 1.1 Wie kann die Dichte von Agenten pro Fläche innerhalb von PEDSIM gemessen werden?

F 1.2 Wie können die erhobenen Dichtewerte graphisch in Echtzeit angezeigt werden?

F 1.3 Wie können die erhobenen Dichtewerte numerisch in Echtzeit angezeigt werden?

F 2.1 Wo entstehen bei einem Flaschenhalsszenario hohe bis gefährliche Dichtewerte?

F 2.2 Bei welchen Flaschenhalshöhen entstehen hohe bis gefährliche Dichtewerte?

In Abschnitt 1 wird der Begriff Crowd-Desaster (1.1) und die Dichte als Ursache (1.2) für ein solches erklärt. Des weiteren werden drei verschiedene Simulatoren auf den Aspekt der Dichtemessung verglichen (1.3).

In Abschnitt 2 wird die Funktionalität von PEDSIM beschrieben. Es werden die grundlegenden Klassen (2.2) und der Aufbau eines XML-Szenarios (2.3) erklärt. Danach wird auf das Social-Force-Modell (2.4.1) und Quadtrees (2.4.3) als Simulationsgrundlage eingegangen. Anschließend wird auf die Erweiterung von PEDSIM (2.5), die im Praxisprojekt erstellt wurde, eingegangen.

In Abschnitt 3 wird beschrieben wie PEDSIM um eine neuartige Dichtemessung erweitert wird. Dafür wird zuerst die Methode **getNeighbors()** betrachtet (3.1). Im weiteren Verlauf wird die erstellte Klasse **Detector** beschrieben (3.2.1).

In Abschnitt 4 wird die erstellte Ausgabe der Messwerte per GUI und Datei beschrieben.

In Abschnitt 5 wird ein Testszenario erstellt, dafür wird ein Flaschenhalsszenario konstruiert (5.1), Agenten und deren Wegpunkte (5.2) und Messflächen positioniert (5.3).

In Abschnitt wird das Testszenario durchgeführt. Davor wird die grundsätzliche Vorgehensweise erläutert (6.1), die Videos dreier Probedurchführungen beschrieben und anschließend

die weitere Vorgehensweise erläutert (6.3).

In Abschnitt 7 werden die Daten der Durchführungen geplottet, um diese zu vergleichen und zu diskutieren.

In Abschnitt 8 erfolgt eine Zusammenfassung der Arbeit und ein abschließendes Fazit.

In Abschnitt 9 wird auf mögliche Fortführungen der Arbeit eingegangen.

Anmerkung: Es wurden Teile der, im Rahmen des Studienverlaufsplans erstellter, Praxisprojektdokumentation „Erweiterung der Crowd Simulation PEDSIM mit einer graphischen Darstellung der Dichte von Agenten“ in dieser Arbeit integriert.

1 Grundlagenteil

1.1 Crowd-Desaster

Der Begriff Massenpanik wird häufig für Unglücksfälle bei Großveranstaltungen genutzt. Dabei wird impliziert, dass eine große Menge von Menschen scheinbar unbegründet in Panik gerät und aufgrund dessen ein irrationales Fluchtverhalten annimmt, ähnlich wie Herdentiere. Diese Implikationen sind in mehreren Aspekten falsch [9]:

- Menschen sind weder Herdentiere noch verhalten sie sich wie solche. Zudem ist für ein herdenähnliches Fluchtverhalten in den meisten Fällen kein Platz. Bei der Loveparade standen die Menschen dicht aneinander gedrängt.
- In den seltensten Fällen ist Panik die Ursache für ein Unglücksfall, stattdessen sind Ursache und Wirkung vertauscht. Panik entsteht aus einer lebensbedrohlichen Situation heraus.

Helbing et al. schlagen deshalb die Verwendung des neutraleren Begriffes Crowd Desaster vor [9].

1.2 Ursache des Crowd-Desaster

Helbing et al. und der Gerichtssachverständige Jürgen Gerlach stellen verschiedene Ursachen des Crowd-Desaster bei der Loveparade heraus [9][2]. Ein wichtiger Aspekt sei, dass das Gelände aufgrund der Geometrie nicht für die erwartbare Anzahl an Besuchern geeignet gewesen sei. Es seien zu viele Menschen auf einer zu kleinen Fläche gewesen, es habe eine zu hohe Menschendichte gegeben und tödliche Drucksituationen seien entstanden. Die Loveparade hätte bei einer genaueren Ortsinspektion und Planung nicht stattfinden dürfen.

1.3 Simulatoren

Für die Planung von Großveranstaltungen wird Software genutzt, die das Verhalten und die Bewegung von Menschenmassen an dem geplanten Verstanstaltungsort simuliert. Da eine Mitursache von Crowd-Desaster eine zu hohe Menschendichte ist, soll diese ebenfalls simuliert und gemessen werden können. Die Vergleichsarbeit von Richards bietet einen Überblick von 34 verschiedenen Programmen [11].

Im Praxisprojekt [10] wurde sich für PEDSIM entschieden, nachdem es mit Vadere [1] und dem Simulator von Hallweger [7] verglichen wurde. Es folgt ein erneuter kurzer Vergleich auf den Aspekt der Dichtemessung.

1.3.1 PEDSIM

PEDSIM ist eine Programmbibliothek mit mehreren Beispielprogrammen. Sie hat eine umfassende Dokumentation und wurde durch die nicht mehr aufrufbare Projekt-Website `http://pedsim.silmaril.org/` erweitert. PEDSIM ist eine veröffentliche und vereinfachte Softwareversion von ALPSIM [5][6]. Während in ALPSIM die Dichte gemessen werden kann, gibt es in PEDSIM keine direkte Funktion um die Dichte zu messen oder gar darzustellen. Es gibt jedoch die Methode **getNeighbors()**, die alle Agenten innerhalb eines

vorgegebenen Radius ausgibt [4]. Das Ausmaß der Nutzbarkeit dieser Methode wird in Abschnitt 3.1 beschrieben.

1.3.2 Vadere

In Vadere wird die Dichte mit „EikMesh“ gemessen, indem die Simulationsfläche per Delaunay-Triangulierung unterteilt wird [12][1]. Es gibt es eine graphische Darstellung der Dichte für alle Punkte. Sie scheint aber nicht in Echtzeit generiert zu werden, sondern sich aus bereits durchgeführten Simulation zu ergeben.

1.3.3 Hallweger Bachelorarbeit

Die Bachelorarbeit von Hallweger dokumentiert ein Projekt auf Basis von Unity [7]. Das Simulationsmodel ist von Unity gestellt und nicht einsehbar. Es bleibt unklar, ob Funktionen zur Dichtemessungen implementiert worden sind.

2 Beschreibung von PEDSIM

2.1 Einleitung

Im folgenden Kapitel wird auf die Eigenschaften von PEDSIM eingegangen. PEDSIM ist als eine Programmbibliothek zu verstehen. Diese enthält vier kleinere Beispielprogramme und ein Demoprogramm. Im weiteren Verlauf wird nur auf das Demoprogramm eingegangen, da es eine graphische Oberfläche hat, die eine Steuerung und verschiedene Darstellungen der Simulation ermöglicht. Vorher werden die grundlegenden Funktionen von PEDSIM erläutert.

2.2 Grundlegende Klassen PEDSIM

Die folgenden Klassen stellen die funktionale Grundlage zur Simulation dar [4]:

- **Ped::Tscene:** Ein Objekt dieser Klasse bildet die Welt der Simulation ab. Ihm werden Objekte der Klassen Ped::Tobstacle und Ped::Tagent hinzugefügt.
- **Ped::Tobstacle:** Ein Objekt dieser Klasse ist eine Linie, Start- und Endpunkt werden in Form von Koordinaten übergeben. Mit mehreren Objekten können auf diese Weise beliebige Hindernisse erstellt werden.
- **Ped::Tagent:** Ein Objekt dieser Klasse ist eine Gruppe von Agenten mit beliebiger Anzahl. Sie werden in einem Bereich generiert, der durch Koordinaten definiert ist. Agenten werden Objekte der Klasse **Ped::Twaypoint** hinzugefügt.
- **Ped::Twaypoint:** Ein Objekt dieser Klasse ist ein einzelner Wegpunkt. Es wird durch Koordinaten und einen Passierradius definiert. Die Agenten laufen die Wegpunkte der Reihenfolge nach ab und passieren sie innerhalb des angegebenen Radius.
- **Ped::Tagent- > move():** Durch den Aufruf machen alle Agenten einen Simulationsschritt, auch „timestep“ genannt.

Ein beispielhafter Ablauf wäre [4]:

1. Ein Objekt **Ped::Tscene** erzeugen.
2. Mit Objekten **Ped::Tobstacle** beliebig viele Hindernisse erzeugen.
3. Objekte **Ped::Tobstacle** dem Objekt von **Ped::Tscene** hinzufügen.
4. Mit Objekten **Ped::Tagent** beliebig viele Gruppen von Agenten erzeugen.
5. Mit Objekten **Ped::Twaypoint** beliebig viele Wegpunkte erzeugen.
6. Objekte **Ped::Twaypoint** den Objekten **Ped::Tagent** hinzufügen.
7. Objekte **Ped::Tagent** dem Objekt von **Ped::Tscene** hinzufügen.
8. Simulationsschritte mit dem wiederholten Aufruf von **Ped::Tagent- > move()** erreichen.

2.3 XML-Szenario

Die Initialisierungswerte der Objekte der Klassen **Ped::Tobstacle**, **Ped::Tagent** und **Ped::Twaypoint** werden im Demoprogramm über eine XML-Datei gesetzt. Dadurch können codeunabhängig beliebige Szenarien erstellt werden.

2.3.1 Beispielszenario

```
<scenario>

  <!--quadratisches Hindernis im Zentrum-->
  <obstacle x1="-20" y1="-20" x2="20" y2="-20" />
  <obstacle x1="20" y1="-20" x2="20" y2="20" />
  <obstacle x1="20" y1="20" x2="-20" y2="20" />
  <obstacle x1="-20" y1="20" x2="-20" y2="-20" />

  <!-- Wegpunkte um Rechteck -->
  <waypoint id="w1" x="-30" y="-30" r="5" />
  <waypoint id="w2" x="30" y="-30" r="5" />
  <waypoint id="w3" x="30" y="30" r="5" />
  <waypoint id="w4" x="-30" y="30" r="5" />

  <!-- Agenten links vom Hindernis -->
  <agent x="-40" y="0" n="50" dx="10" dy="10">
     <addwaypoint id="w1" />
     <addwaypoint id="w2" />
     <addwaypoint id="w3" />
     <addwaypoint id="w4" />
  </agent>

  <!-- Agenten rechts vom Hindernis -->
  <agent x="40" y="0" n="50" dx="10" dy="10">
     <addwaypoint id="w2" />
     <addwaypoint id="w1" />
     <addwaypoint id="w4" />
     <addwaypoint id="w3" />
  </agent>

</scenario>
```

Listing 1: Beispielszenario.xml

2.3.2 XML-Elemente des Szenarios

Im folgenden werden die Elemente erläutert [4]:

- **<scenario>:** Die Datei muss ein Wurzelelement haben, der Name kann beliebig sein.
- **<obstacle [...]>:** Ein obstacle-Tag stellt eine Linie dar, die mit den Startpunktkoordinaten (x1|y1) und Endpunktkoordinaten (x2|y2) definiert wird. Mit mehreren Linien können auf diese Weise zweidimensonale Hindernise erstellt werden. Die Reihenfolge der Linien ist dabei beliebig.
- **<waypoint [...]>:** Ein waypoint-Tag stellt einen Wegpunkt dar, der mit einer ID, mit der Positionskoordiante (x|y) und dem Passierradius r definiert ist. Über die ID wird auf einen Wegpunkt referenziert, sodass dieser mehrmals genutzt werden kann.
- **<agent [...]>:** Ein agent-Tag stellt eine Gruppe von Agenten dar, die mit den Ursprungskoordianten (x|y), der Agentenanzahl n und den Verteilungswerten dx und dy definiert werden. Die Agenten werden in einem von Bereich x−dx bis x+dx und y−dy bis y+dy generiert.
- **<addwaypoint [...]>:** Mit dem addwaypoint-Tag werden innerhalb des agent-Elementes per ID Wegpunktreferenzen auf die vorher definierten Wegpunkte gesetzt. Die Reihenfolge der Tags setzt auch die Reihenfolge der Wegpunkte fest. Die Agenten laufen die Wegpunkte nacheinander ab. Wenn der letzte Wegpunkt erreicht wird, laufen die Agenten wieder zum ersten Wegpunkt und beginnen von vorne.

2.3.3 Beschreibung des Szenarios

Das Szenario stellt folgendes dar: Ein Quadrat ist in der Mitte, links und rechts werden zwei Gruppen von jeweils 50 Agenten generiert. Die linke Gruppe bewegt sich im Uhrzeigersinn um das Quadrat, die rechte Gruppe links um das Quadrat. Da beide Gruppen die selben Wegpunkte ablaufen, müssen sie sich gegenseitig ausweichen.

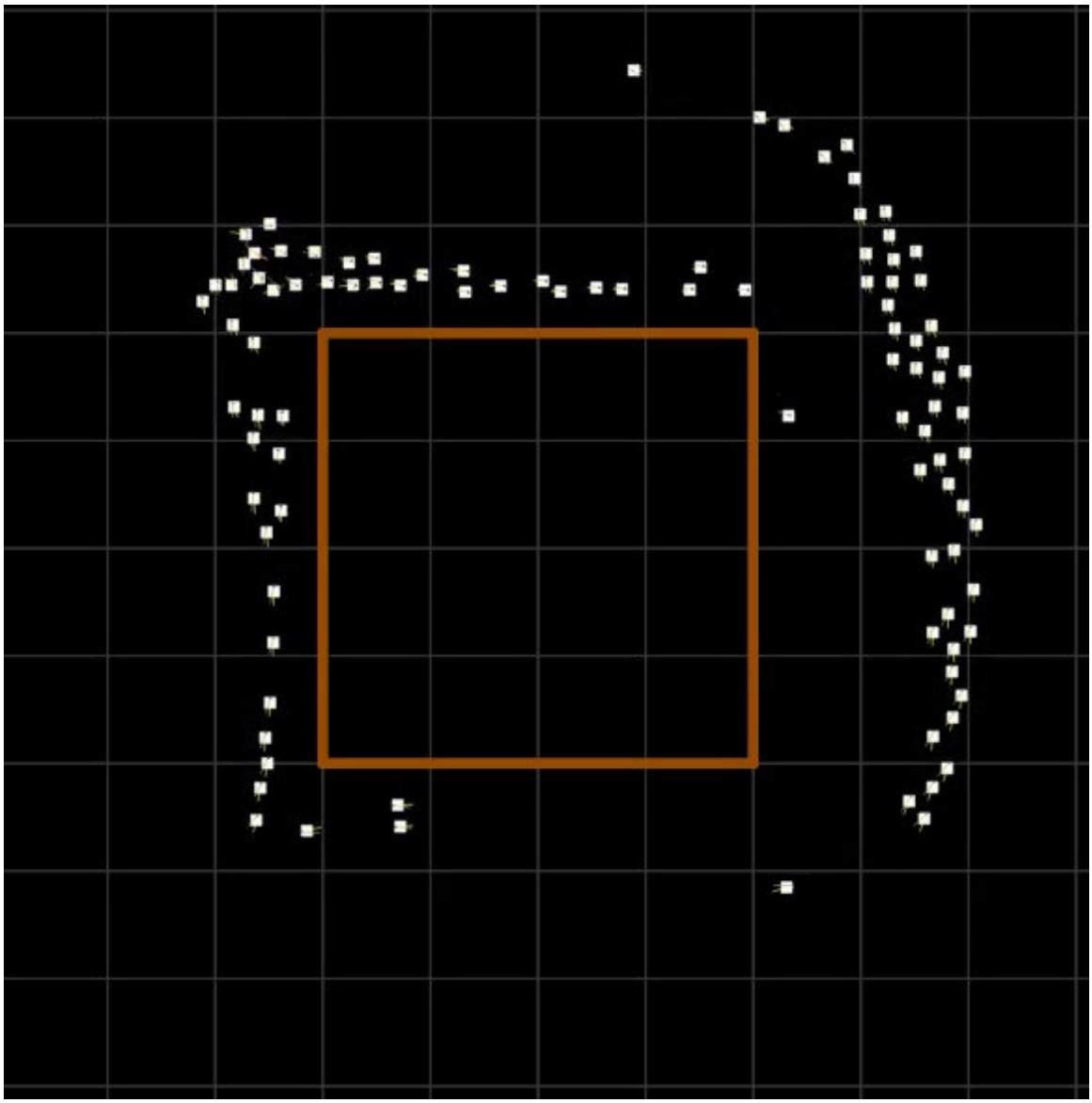

Abbildung 1: Screenshot aus Beispielszenario.mp4

2.4 Simulationsgrundlage

2.4.1 „Social Force Model“

Für die grundlegenden Berechnungen wird das „Social Force Model“ von Helbing genutzt. Es bildet die Bewegung von Personen über eine Summe von drei Kraftvektoren ab [8]:

1. Die gewünschte Geschwindigkeitsänderung pro Strecke, um ein gewünschtes Ziel zu erreichen, als Produkt mit der Masse der Person.

2. Die abstoßende Kraft zwischen Personen untereinander sowie zwischen Personen und Objekten.

3. Die anziehende Kraft für Personen von attraktiven Objekten in der Umgebung.

2.4.2 Umsetzung des „Social Force Model“

In PEDSIM werden die ersten beiden Kraftvektoren genutzt. In der Dokumentation wird folgende Gleichung aufgestellt [4]:

$$m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{\mathbf{v}_i^0 - \mathbf{v}_i}{\tau_i} + \sum_{j \neq i} \mathbf{f}_{ij} + \sum_{W} \mathbf{f}_{iW}$$

$$m_i \frac{d\mathbf{v}_i}{dt}$$

Die Kraft für einen Agenten mit der Masse m_i mit der Geschwindigkeitsänderung $\frac{d\mathbf{v}_i}{dt}$ ergibt sich aus der Summe der Kraftvektoren:

$$m_i \frac{\mathbf{v}_i^0 - \mathbf{v}_i}{\tau_i}$$

m_i die Masse, v_i^0 ist die gewünschte Geschwindigkeit, $\mathbf{v}_i$ die tatsächliche Geschwindigkeit und τ_i das Zeitintervall. Das Ziel eines Agenten ist immer, sich dem nächsten Wegpunkt anzunähern. Sie wird durch **Ped::Tvector Ped::Tagent::desiredForce()** realisiert.

$$\sum_{j \neq i} \mathbf{f}_{ij}$$

Die Summe der abstoßenden Kraft der Agenten untereinander. Dafür wird über alle Agenten in der Szene iteriert und jeweils die Kräfte berechnet. Sie ist als unmittelbare Kollisionsvermeidung angelegt. Sie wird durch **Ped::Tvector Ped::Tagent::socialForce()** realisiert.

$$\sum_{W} \mathbf{f}_{iW}$$

Die Summe der abstoßende Kraft zwischen Agenten und Hindernissen als unmittelbare Kollisionsvermeidung. Es wird über alle Hindernisse des Szenarios iteriert. Sie wird durch **Ped::Tvector Ped::Tagent::obstacleForce()** realisiert.

2.4.3 Quadtree

Die Agenten werden in einer dynamischen Quadtreestruktur gespeichert [4]. Sie unterteilt die Welt rekursiv in vier rechteckige Quadranten. Die Agenten bewegen sich in dieser Welt und werden anhand ihrer aktuellen Position im Quadtree gespeichert. Wenn die maximale Agentenanzahl von acht in einem Quadranten erreicht wird, wird dieser wieder in vier neue Quadranten unterteilt, entsprechend wird der Quadtree um vier Kindsknoten erweitert [3]. Verlassen Agenten den Quadranten, wird dieser entfernt, entsprechend auch die leeren Kindsknoten. Realisiert wird dies durch Objekte der **Klasse Ped::Ttree**, diese werden automatisch durch die Klasse **Ped:Tscene** erstellt.

Die Methode **getNeighors()** der Klasse **Ped::Tagent** nutzt diese Struktur um innerhalb eines vorgegebenen Radius benachbarte Agenten zu finden. Dabei wird über Baumabschnitte iteriert, anstatt über alle vorhandenen Agenten.

2.5 Erweiterung von PEDSIM mit einer graphische Darstellung der Dichte

Im Praxisprojekt wird die Agentendichte in Relation zur Quadtreetiefe gesetzt, da bei größerer Tiefe die Fläche der Quadranten immer kleiner wird [10]. Die Quadtrees werden der Tabelle entsprechend eingefärbt, das Ergebnis der Umsetzung ist in der nachfolgenden Abbildung zu sehen. Die Erreichung von Dichtegrad III ist kritisch, da hier gefährliche Dichtewerte entstehen.

Dichtegrad	**Farbe**	**Tiefe**	**Agentendichte**
I	grün	5	erhöht, aber es besteht kein Risiko
II	gelb	6	deutlich erhöht
III	rot	7	zu hoch, gefährlich

Tabelle 1: Agentendichte einfärben

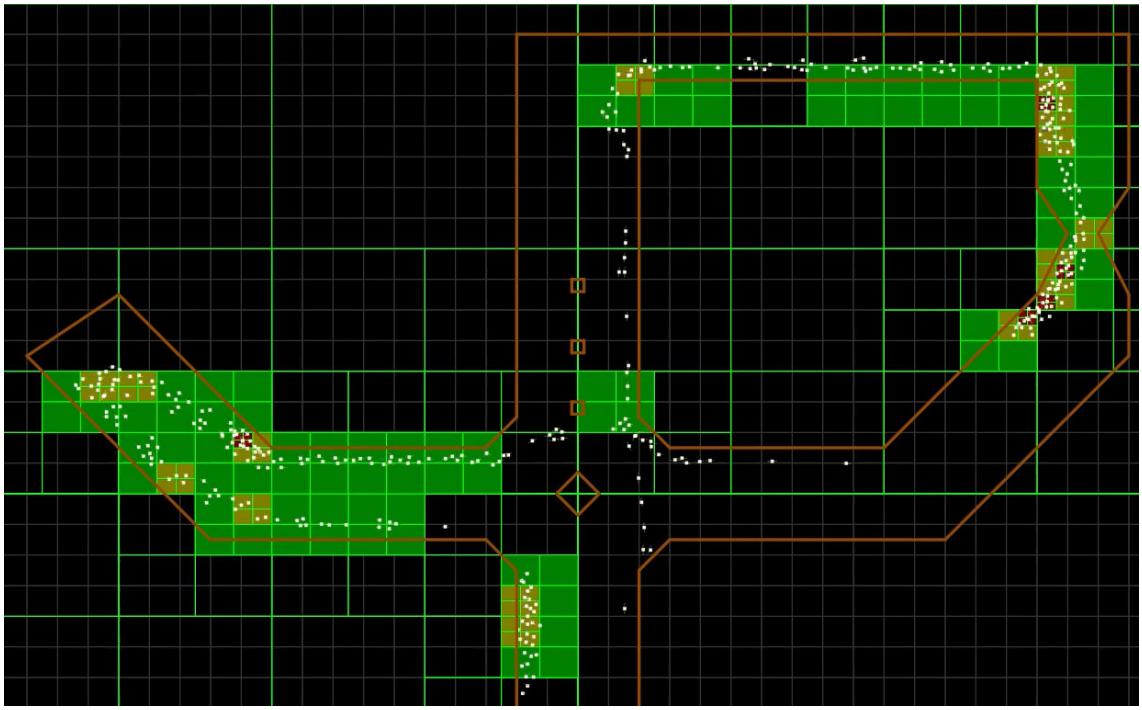

Abbildung 2: Einfärbung der Quadtrees in Relation zur Agentendichte

3 Erstellung einer neuartigen Dichtefunktion

3.1 Betrachtung der Methode getNeighbors()

Um die Dichte von Agenten pro Fläche zu erfassen, müssen in einer vorgegebenen Fläche die Agenten gezählt werden. Dafür wird die Methode **getNeighbors()** genutzt. Laut Dokumentation sollen die Agenten ausgegeben werden, die sich innerhalb des Radius einer Suchdistanz von einem Suchpunkt befinden [4]. Rechnerisch wird damit ein quadratisches Suchfeld gebildet, indem der Suchpunkt den Mittelpunkt darstellt. Jedoch werden die Agenten ausgegeben, die sich im selben Quadtree wie der Suchpunkt befinden. Damit werden auch Agenten ausgegeben, die sich außerhalb des Suchradius befinden. Eine genauere Beschreibung des Umstandes befindet sich im Anhang.

Um also in einer vergebenen Fläche die Agenten zu zählen, wird **getNeighbors()** mit einer zusätzlichen Überprüfung genutzt. Die gefundenen Agenten werden zusätzlich anhand ihrer Koordinaten überprüft, ob sie sich in dem gebildeten Suchfeld befinden.

3.2 Erstellung einer eigenen Detektorklasse

3.2.1 Grundlagen

Die Objekte der Klasse **Detector** zählen die Agenten, die sich innerhalb eines vorgegebenen Suchfeldes befinden.

Die wichtigsten Klassenvariablen sind:

- Name des Detektors
- Anzahl der Agenten
- Fläche des Suchfeldes
- Dichte (Anzahl Agenten pro Fläche)
- Farbe der Linienumrandung des Suchfeldes
- Koordinaten des Suchfeldes als
  - Rechtecksform
  - Quadratsform
  - optional: Eckpunkte für Polygone

Zusätzlich wird das Suchfeld graphisch dargestellt. Die Objektreferenzen werden in einem Queue-Objekt **stats** der externen Helfer-Klasse **Stats** gespeichert, um eine vereinfachte Ausgabe der Werte in der GUI und in einer Ausgabedatei zu ermöglichen. Der Quellcode der Detektorklasse befindet sich im Anhang.

3.2.2 Konstruktoren

Es werden zwei Konstruktoren erstellt, diese unterscheiden sich im Aufruf:

1. Ein quadratisches Suchfeld, erstellt durch die Quadratsform. Es wird mit dem Suchpunkt bzw. Mittelpunkt $(x|y)$ und Suchradius r aufgerufen.
2. Ein rechteckiges Suchfeld, erstellt aus der Rechtecksform. Es wird mit dem Ausgangspunkt oben-links $(x|y)$, der Länge w und der Höhe h aufgerufen.

Aus den Aufrufwerten beider Konstruktoren wird jeweils zusätzlich die Rechtecksform und die Quadratsform gebildet. Die Quadratsform des rechteckigen Suchfeldes wird aus dem kleinsten umgebenden Quadrat gebildet, da das Suchfeld von **getNeighbors()** quadratisch ist. Eine Herleitung der Suchfelder befindet sich im Anhang.

3.2.3 Polygon-Suchfelder

Für Bereiche, wo durch Obstacle Schrägen entstehen, werden polygone Suchfelder benötigt. Anstatt eine Punktüberprüfung für Polygone zu erstellen, wurde aus Zeitmangel eine Zwischenlösung implementiert. Diese funktioniert, wenn die schräge Seite des Suchfeldes direkt an den Obstacle grenzt.

Der Konstruktoraufruf erfolgt mit dem kleinsten umgebenden Rechteck des Polygons. Dieses wird händisch errechnet. Weitere Schritte sind:

- Deklarierung als Polygon.
- Übergabe der händisch errechneten Fläche des Suchfeldes.
- Übergabe aller Eckpunkte in Koordinatenform, um eine graphische Darstellung zu erreichen.

3.2.4 Detector::count()

Die Klassenmethode zählt die Agenten, die innerhalb des Suchfeldes liegen. Dafür werden die, von **getNeighbors()** erfassten Agenten überprüft, ob sie in dem Suchfeld liegen. Die Prüfung findet per Rechtecksform statt. Da für alle Suchfelder eine Rechtecksform gebildet wird, kann die Prüfung für quadratische, rechteckige und polygone Suchfelder genutzt werden.

3.2.5 Detector::draw()

Die Klassenmethode zeichnet die Linien des Suchfeldes bei normalen Objekten mit der Rechtecksform und bei Polygon-Objekten mit den Koordinaten der Eckpunkte. Die Farbe der Linien wird per Konstruktoraufruf angegeben. Die Fläche wird der Dichte entsprechend eingefärbt. Die Dichtewerte und deren farbliche Entsprechung werden der Tabelle aus 2.5 entnommen.

In dem Praxisprojekt wird die Dichte in Relation zur Quadtreetiefe gesetzt [10]. Es werden

Quadtrees mit der Tiefe 5 bis 7 eingefärbt. Davon werden die Flächen ausgegeben und die minimale Dichte $D0min$ bei einem Agenten und die maximale Dichte $D0max$ bei acht Agenten berechnet. Um feste Grenzwerte zu erhalten, werden Mittelwerte aus den sich überschneidenden Werten gebildet:

- $D1min$ der Tiefe n bildet sich aus dem Mittelwert von $D0max$ aus Tiefe $n-1$ und $D0min$ aus Tiefe n.
- $D1max$ der Tiefe n bildet sich aus dem Mittelwert von $D0max$ aus Tiefe n und $D0min$ aus Tiefe $n+1$.

Dichtegrad	**Farbe**	**Tiefe**	**Fläche**	$D0min$	$D0max$	$D1min$	$D1max$
-	transparent	4	500	0,002	0,016	-	0,012
I	grün	5	125	0,008	0,064	0,012	0,048
II	gelb	6	31,25	0,032	0,256	0,048	0,192
III	rot	7	7,8125	0,128	1,024	0,192	0,768
-	-	8	1,953125	0,512	4,096	0,768	-

Tabelle 2: Grenzwerte für Einfärbung der Dichte

3.2.6 Detector::getDensity()

Die Klassenmethode ruft **count()** auf, um die momentane Agentenanzahl des Suchfeldes zu erfassen. Die Dichte wird aus der Division der Agentenanzahl und Fläche des Suchfeldes errechnet und in der Dichte-Klassenvariable gespeichert. Dann wird die Klassenmethode **draw()** aufgerufen, diese färbt, der momentanen Dichte entsprechend, das Suchfeld ein.

3.3 Fazit

Mit diesem Abschnitt können die beiden Forschungsfragen **F 1.1** und **F 1.2** beantwortet werden. Die Dichte der Agenten kann durch die Methode **getNeighbors()** mit einer zusätzlichen Koordinatenprüfung gemessen werden. Darauf basierend wurde eine Klasse erstellt, deren Objekte für quadratische, rechteckige und teilweise weitere Messflächen, die Agentendichte messen. Für eine graphische Darstellung der Dichte in Echtzeit werden die Messflächen ihrer Dichte entsprechend eingefärbt.

4 Ausgabe der Dichtewerte

4.1 Ausgabe per GUI und Datei

Simulationsberechnung und Durchführung werden in einer bestimmten Frequenz getätigt. Sie erfolgen mit der Angabe der Simulationsgeschwindigkeit, die die Periodendauer darstellt und der Länge eines Simulationsschrittes. Ein Simulationsschritt wird dabei Timestep genannt.

Die Simulationsgeschwindigkeit wird je nach Performance des Systems gedrosselt, die Drosselung kann gemessen und angezeigt, aber nicht beeinflusst werden. Da die Simulationsgeschwindigkeit die zeitliche Dauer einer Simulationsdurchführung beeinflusst, werden die durchgeführten Timesteps betrachtet, diese sind unabhängig von der Geschwindigkeit.

Mit Aufruf der Demoapp startet gleichzeitig die Simulation, es werden sofort Timesteps durchgeführt. Damit die Simulation optisch pausiert ist, wird für den Aufruf der Demoapp die Schrittlänge auf 0 gesetzt. Es werden Timesteps mit der Schrittlänge 0 durchgeführt, also werden die Agenten nicht bewegt. Mit dem Klick auf einen erstellten Button wird die eigentliche Simulationsdurchführung gestartet: Die Schrittlänge wird auf 1 gesetzt und eine Variable zählt die Timesteps, die ab diesem Zeitpunkt durchgeführt werden.

Die Ausgabe der Simulationswerte soll per GUI und per Dateiausgabe, hier im CSV-Format, erfolgen. Benötigt werden der Wert der durchgeführten Timesteps seit Start der Simulationsdurchführung sowie die Namen und Dichtewerte der jeweiligen Messfelder. Per Timestepswert können im Nachgang mögliche Bildschirmvideos mit der Dateiausgabe synchronisiert werden. Da die Objektreferenzen aller Messfelder in der Queue **stats** gespeichert sind (siehe 3.2.1), erfolgt die Ausgabe per Iteration durch **stats**. Mit der Ausgabe erfolgt auch der Aufruf von **getDensity()** (siehe 3.2.6), der die Dichte misst.

Eine ausführliche Beschreibung der benötigten Schritte zur Erstellung der Ausgabe befindet sich im Anhang.

4.2 Fazit

Mit diesem Abschnitt kann die Forschungsfrage **F 1.3** beantwortet werden. Die Dichtewerte der Messfelder und die Anzahl der durchgeführten Timesteps werden in Echtzeit mit der Helferklasse **Stats** an die GUI übergeben. Desweiteren werden alle benötigen Werte und Eigenschaften der Messfelder in eine CSV-Datei geschrieben.

5 Erstellung des Testszenarios

Es soll ein Testszenario erstellt werden, in dem die Agenten zu einer Engstelle streben und diese passieren. Währenddessen sollen Dichtewerte erfasst werden. Dafür werden an kritischen Stellen des Szenarios Messflächen positioniert.

5.1 Konstruktion des Flaschenhalses

Um eine Engstelle zu erhalten, wird eine Flaschenhalskonstruktion erstellt. Die Höhe des Flaschenhalses soll variiert werden können, um zu testen, bei welchen Höhen kritische Dichtewerte entstehen. Die Konstruktion ist zur vertikalen und horizontalen Achse symmetrisch. Beide Achsen laufen durch den Ursprung $(0|0)$. Daher kann die Konstruktion anhand zweier Koordinaten und der Flaschenhalshöhe f konstruiert werden.

Ein Obstacle in PEDSIM setzt sich durch die Linienverbindung eines Start- und Endpunktes zusammen. Durch Spiegelung der Punkte S_1 und S_2 zur vertikalen Achse entstehen die Punkte S_3 und S_4. Alle vier Punkte werden zur horizontalen Achse gespiegelt. Es werden jeweils S_1 bis S_4 und S'_1 bis S'_4 verbunden.

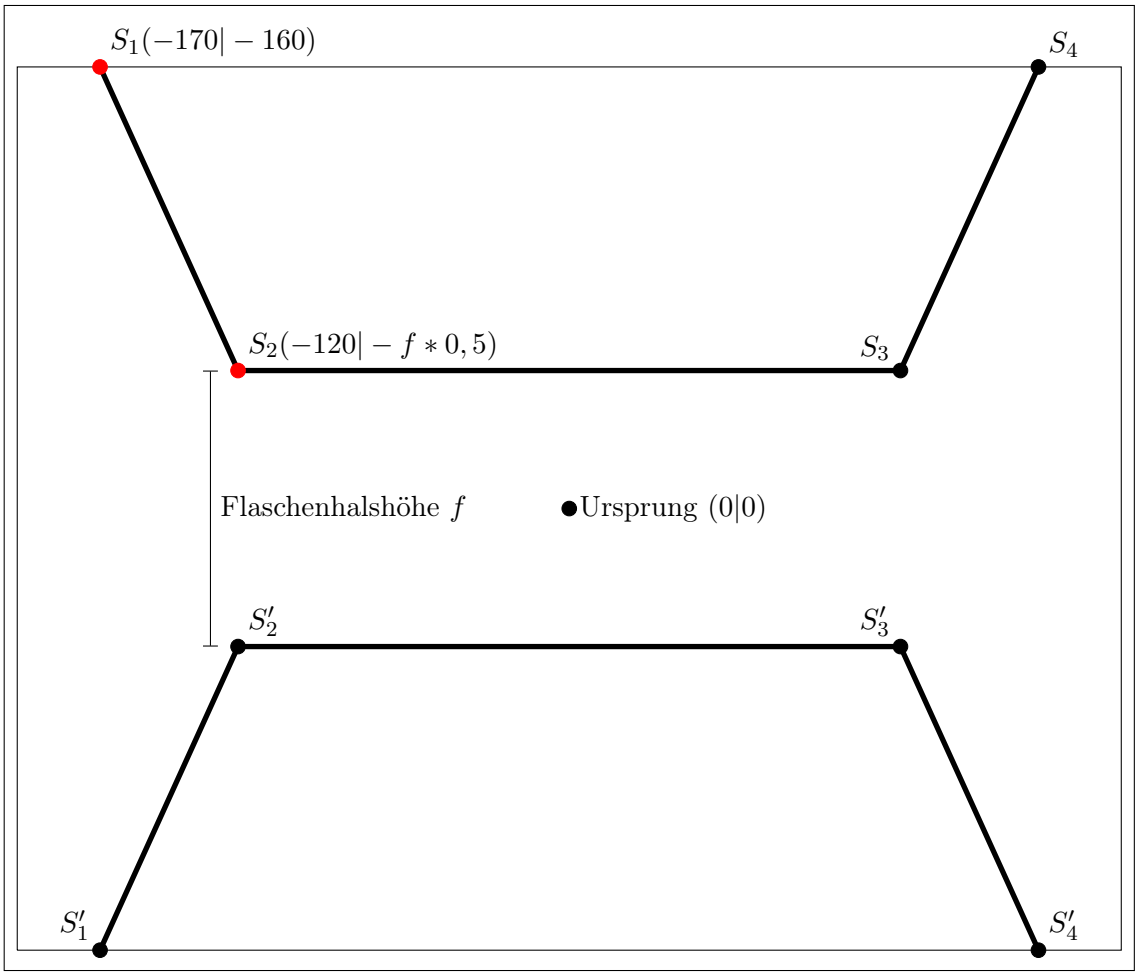

Abbildung 3: Flaschenhalskonstruktion

Punkt	x	y	**gespiegelt von**
S_1	-170	-160	-
S_2	-120	$-f * 0,5$	-
S_3	$x_{S_2} * -1$	y_{S_2}	S_2
S_4	$x_{S_1} * -1$	y_{S_1}	S_1
S'_1	x_{S_1}	$y_{S_1} * -1$	S_1
S'_2	x_{S_2}	$y_{S_2} * -1$	S_2
S'_3	x_{S_3}	$y_{S_3} * -1$	S_3
S'_4	x_{S_4}	$y_{S_4} * -1$	S_4

Tabelle 3: Koordinaten der Flaschenhalskonstruktion

5.2 Positionierung der Agenten und Wegpunkte

Der Ursprungsbereich der Agenten, in der nachfolgenden Abbildung dargestellt durch die graue Fläche, liegt am linken Szenariorand. Er geht mit vertikaler Ausbreitung von $dy = 320$ über die gesamte Szenariohöhe und hat eine horizontale Ausbreitung von $dx = 30$.

Agenten sollen sich vom Ursprungsbereich links nach rechts durch den Flaschenhals bewegen. Dafür werden Wegpunkte gesetzt, diese haben einen kreisförmigen Passierbereich. Sobald Agenten diesen erreicht haben, streben sie den nächsten Wegpunkt an.

Damit die Agenten gleichmäßig in den Flaschenhalseingang strömen, wird Wegpunkt W_1 genau in der Mitte des Flaschenhalses positioniert, der Durchmesser des Passierbereiches entspricht dabei der Flaschenhalshöhe f.

Wegpunkt W_2 wird an Punkt $(100200|0)$ gesetzt. Sein Passierbereich beginnt genau an der rechten Szenariogrenze. Der Radius des Passierbereiches ist mit 100000 dabei so groß, das dort seine Kreissehne und sein Kreisbogen annähernd die selbe Länge wie die Szenariohöhe haben. Da der Passierbereich von W_2 genau an der Szenariogrenze liegt, können Agenten diesen nie erreichen. Dies wird hingenommen, weil der Schwerpunkt der Beobachtung auf dem Flaschenhalses liegt.

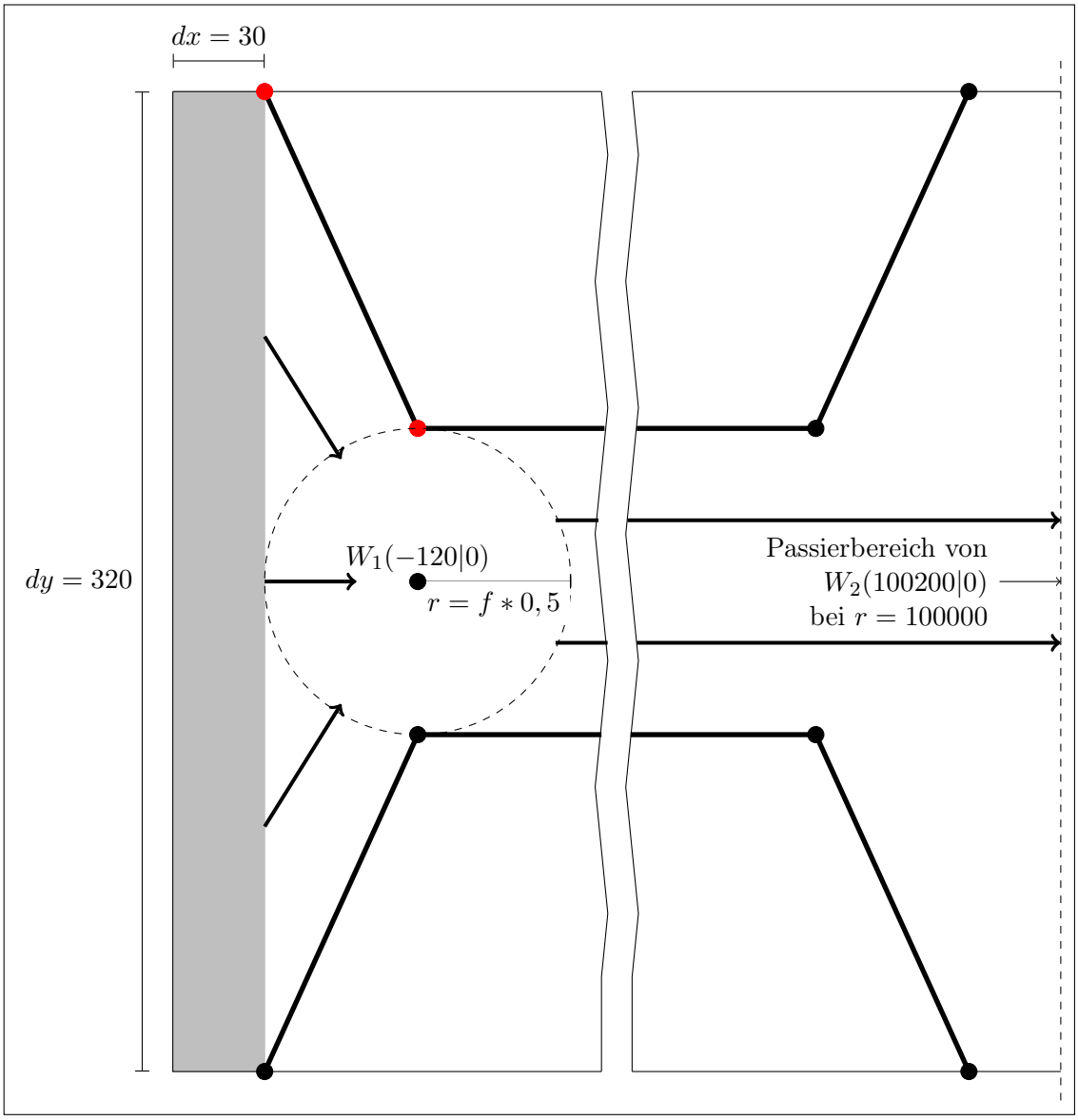

Abbildung 4: Positionierung von Agentenursprung und Wegpunkte

5.3 Positionierung der Messflächen

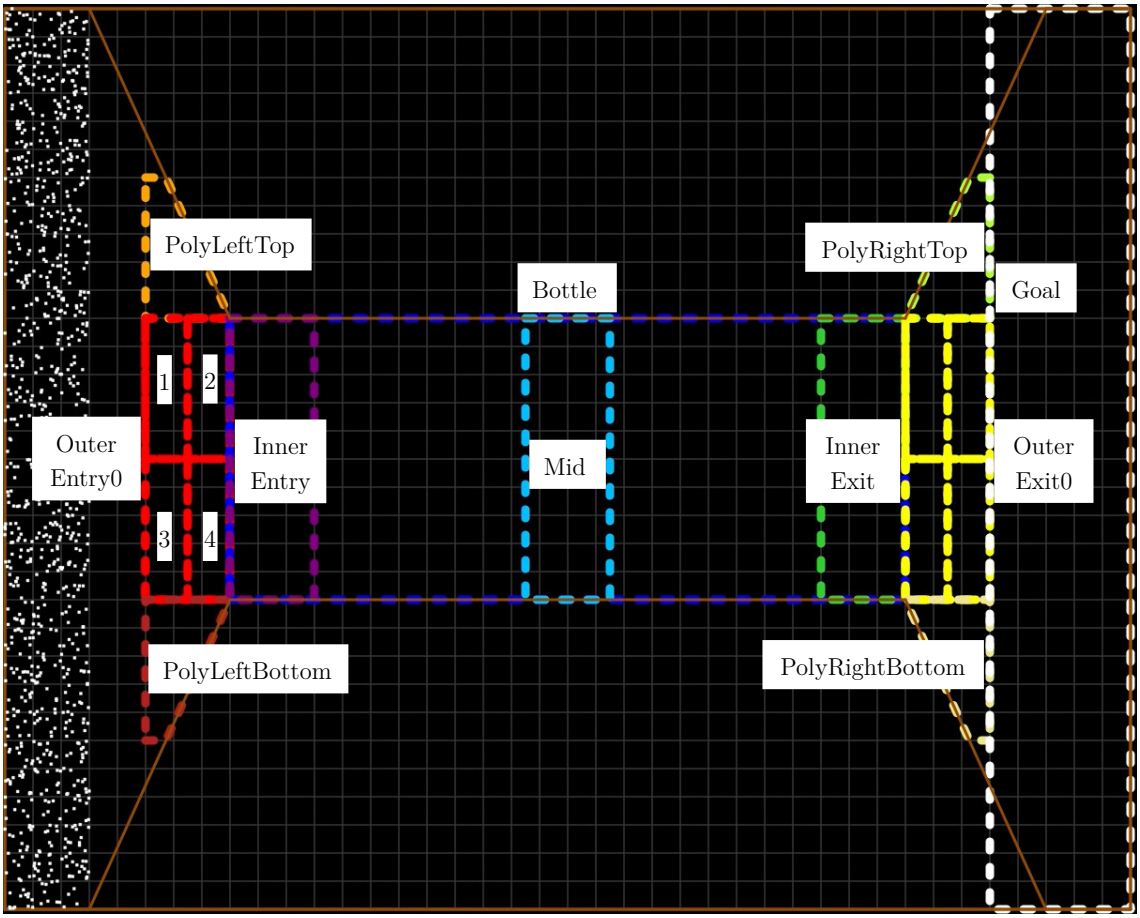

Abbildung 5: Namen der Messflächen

Gemessen werden soll an Stellen, wo vermutlich hohe Dichtewerte auftreten. Es werden Messflächen im Eingangsbereich des Flaschenhalses, im Flaschenhals selber und im Ausgangsbereich des Flaschenhalses positioniert.

Je größer die Messfläche ist, desto ungenauer kann die Messung sein, da sich Agenten in bestimmten Bereichen der Messfläche sammeln können. Dort entstehen punktuell höhere Dichtewerte, die aber von einer größeren Messfläche nicht erfasst werden würden. Deshalb wird **OuterEntry0** zusätzlich in die vier Messflächen **OuterEntry1** bis **OuterEntry4** unterteilt. Der gesamte Flaschenhals wird mit **Bottle** gemessen, zusätzlich am Eingang mit **InnerEntry**, in der Mitte mit **Mid** und am Ausgang mit **InnerExit**. An den Schrägen des Eingangsbereich sind oben **PolyLeftTop** und unten **PolyLeftBottom** positioniert.

Bei einem Testdurchlauf wurde festgestellt, dass die Agenten **PolyRightTop** und **PolyRightBottom** nicht durchlaufen, aufgrund der Positionierung des Wegpunktes $W2$. Ebenso ist eine Unterteilung von **OuterEntry0** nicht notwendig, da die Agenten diese gleichmäßig passieren. Deshalb wird im weiteren Verlauf der Arbeit für den äußeren Ausgangsbereich nur **OuterExit0** betrachtet.

Zusätzlich wird die Messfläche **Goal** erstellt, deren Aufgabe das Zählen der Agenten ist. Sie wird hinter dem **OuterExit0** positioniert. Ein Durchlauf wird als beendet gesehen,

wenn sich 99% der Agenten innerhalb von **Goal** befinden. Zum Verständnis ist **Goal** in der Abbildung eingezeichnet, da sie jedoch nicht zur Dichtemessung genutzt wird, wird sie bei den Durchführungen im nachfolgenden Abschnitt ausgeblendet. Die Herleitung für Messflächen mit Schrägen und eine Übersicht der Koordinaten der relevanten Messflächen befindet sich im Anhang.

6 Durchführung

6.1 Grundsätzliche Vorgehensweise

Für die Konstruktion des Testszenarios werden alle Punkte der Obstacle und die Positionen der Messflächen relativ zur Flaschenhalshöhe f gesetzt. In einer Tabellenkalkulation werden für den Wert der angegeben Flaschenhalshöhe f die Tags für die XML-Datei und die Konstruktoraufrufe der Messflächen generiert. Im Anhang befinden sich beispielhaft die generierten Dateien für $f = 50$. Die Namen der Durchführungen sind „DF“ und Wert der Flaschenhalshöhe f. Die Agentenanzahl beträgt bei jeder Durchführung 1000.

Die Durchführungen der Simulationen wurden gefilmt und in der Nachbearbeitung die Geschwindigkeit der Videos um Faktor 5 erhöht. Falls Nahaufnahmen benötigt wurden, wurde die Simulation ein weiteres Mal durchgeführt.

6.2 Probedurchführungen

Im ersten Schritt wurden probeweise mit den Flaschenhalshöhen $f = 100$, $f = 50$ und $f = 25$ durchgeführt und beschrieben. Besonderer Wert wurde dabei auf die farbliche Einfärbung der Messflächen und deren entsprechenden Dichtegrade gelegt.

6.2.1 Beschreibung des Videos DF100.mp4

Im Eingangsbereich tritt teilweise Dichtegrad II auf, **OuterEntry0** und **InnerEntry** werden gelb, **PolyLeft** und **PolyRight** bleiben grün. Innerhalb des Flaschenhalses ist insgesamt Dichtegrad I, **Bottle** wird grün, vereinzelte Bereiche erreichen aber auch Dichtegrad II, **Mid** wird grün.

Nachdem das erste Drittel der Agenten den Eingangsbereich passiert hat, passieren die Agenten von den weiter entfernten Startpositionen diesen ebenfalls. Dabei bewegen sich die Agenten hauptsächlich an den inneren Wänden des Flaschenhalses entlang, sodass eine Hufeisenform entsteht.

Wenn das erste Drittel der Agenten den Ausgangsbereich passiert, entsteht Dichtegrad II, **OuterEntry0** und **InnerExit** werden gelb, danach sinkt der Dichtegrad auf I.

Insgesamt bleibt die Dichte hier moderat, es werden Grad I und Grad II erreicht. Die entstehende Hufeisenform lässt darauf schließen, dass die Flaschenhalshöhe noch deutlich verringert werden kann.

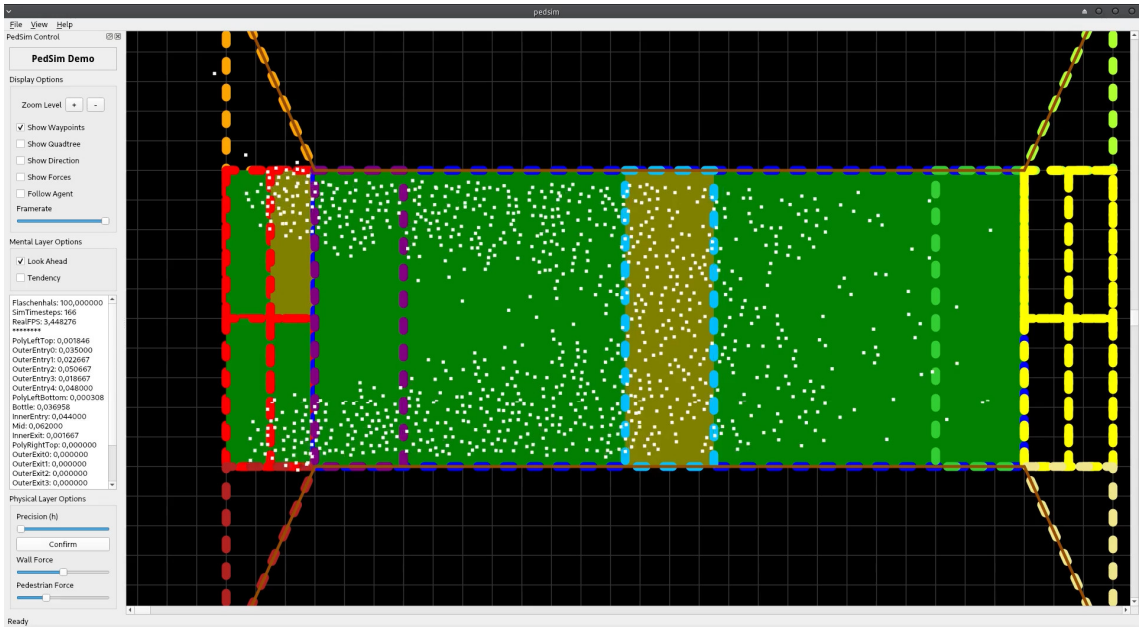

Abbildung 6: Deutliche Hufeisenbildung bei Screenshot von DF100.mp4

6.2.2 Beschreibung des Videos DF50.mp4

Im Eingangsbereich tritt komplett Dichtegrad II auf,**OuterEntry0** und **InnerEntry** sowie **PolyLeftTop** und **PolyLeftBottom** werden gelb. Ebenso wird innerhalb des Flaschenhalses Dichtegrad II erreicht, **Bottle** ist die meiste Zeit gelb.

Im Gegensatz zum Durchlauf mit $f = 100$ entsteht hier fast gar keine Hufeisenform mehr.

Wenn die Agenten den Ausgangsbereich passieren, entsteht die meiste Zeit Dichtegrad II, **OuterExit0** und **InnerExit** werden gelb.

Insgesamt ist die Dichte deutlich höher als bei DF100, es wird durchgehend Grad II erreicht.

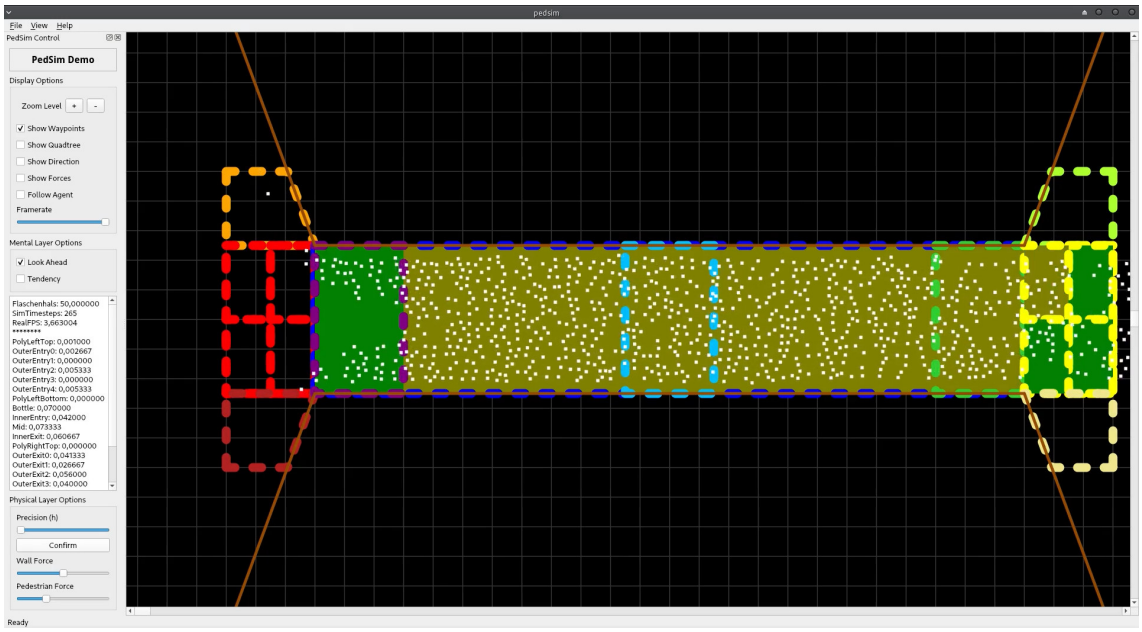

Abbildung 7: Verringerte Hufeisenbildung bei Screenshot von DF50.mp4

6.2.3 Beschreibung des Videos DF25.mp4

Im Eingangsbereich tritt komplett Dichtegrad III auf, **OuterEntry0** bis **4** sowie **PolyLeftTop** und **PolyLeftBottom** werden rot. Innerhalb des Flaschenhalses wird maximal Dichtegrad II erreicht, **Bottle** ist die meiste Zeit gelb, ebenso **InnerEntry** und **Mid**.

Eine Hufeisenform lässt sich nicht beobachten.

Im Ausgangsbereich ist die meiste Zeit Dichtegrad II, **InnerExit** und **OuterExit0** werden gelb.

Wie bei DF50 wird durchgehend Dichtegrad II erreicht. Zusätzlich wird im äußeren Eingangsbereich Dichtegrad III erreicht.

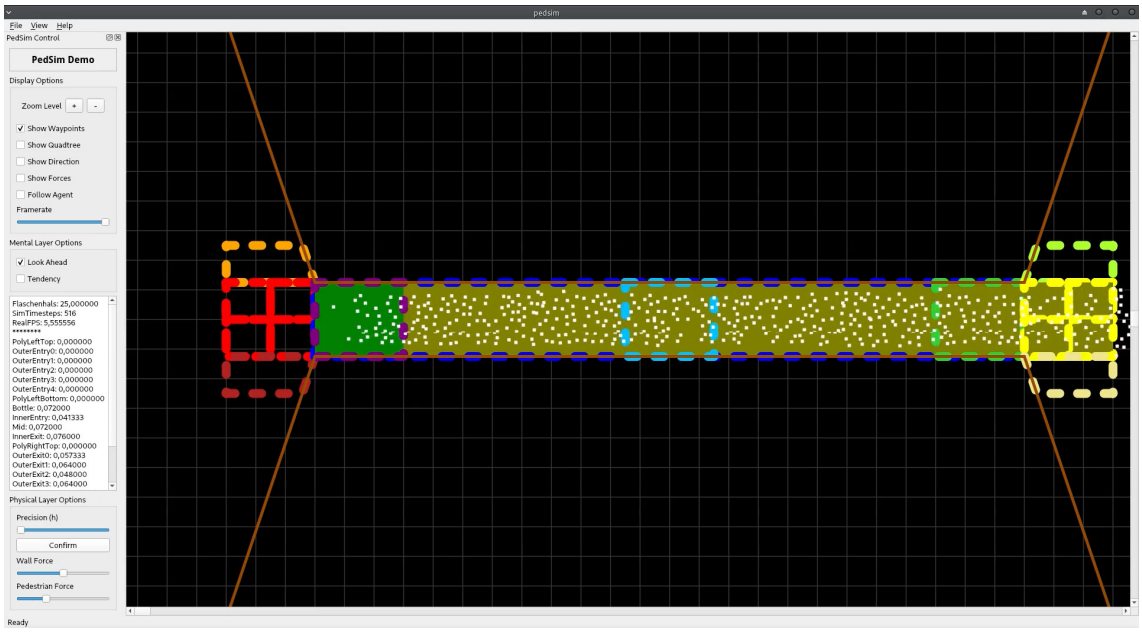

Abbildung 8: Screenshot von DF25.mp4

6.2.4 Fazit der Probedurchführungen

Die Probedurchführungen haben gezeigt, dass im Eingangsbereich immer ein höherer Dichtegrad entsteht, als im eigentlichen Flaschenhals und im Ausgangsbereich. Um die genauen Dichtewerte aller Messflächen abzulesen, müssen die Werte weiterer Durchgänge erhoben und geplottet werden.

6.3 Weitere Vorgehensweise

Anschließend wurden weitere Durchführungen mit $f = 100$ bis $f = 10$ in fünf-er Schritten abfallend getätigt. Die geringste ist $f = 10$, da hier Artefakte entstehen: Die Agenten prallen im Flaschenhals zwischen den gegenüberliegenden Wänden hin und zurück, gut sichtbar in dem Video AusschnittDF10.mp4. Innerhalb des Flaschenhalses stößt die in Abschnitt 2.4.1 beschriebene Wall Force die Agenten von einer Wand ab. Durch die geringe Höhe werden die Agenten bis zur gegenüberliegenden Wand gestoßen und von dort wieder zurück.

Es gibt also Datensätze für 19 Durchführungen. In einem weiteren Datensatz wurden von allen Durchführungen die Maximalwerte aller Messflächen erhoben. Zusätzlich wurde die Durchlaufzeit in durchgeführten Timesteps erhoben. Wie in Abschnitt 5.3 beschrieben, ist ein Durchlauf beendet, wenn sich 99% der Agenten im Zielbereich befinden, markiert durch das Suchfeld **Goal**. Aus den Datensätzen können folgende Plots erstellt werden:

- Von jeder Durchführung die Entwicklung der Dichtewerte in Relation zu den durchgeführten Timesteps.
- Ein Plot mit den Maximalwerten der Dichte und der Durchlaufzeit in Relation zur Flaschenhalshöhe.

Die Farben der Graphen entsprechen den jeweiligen Linienfarben der Messflächen, soweit nicht anders angegeben.

7 Diskussion

7.1 Vergleich der Dichtewerte von DF25, DF50 und DF100

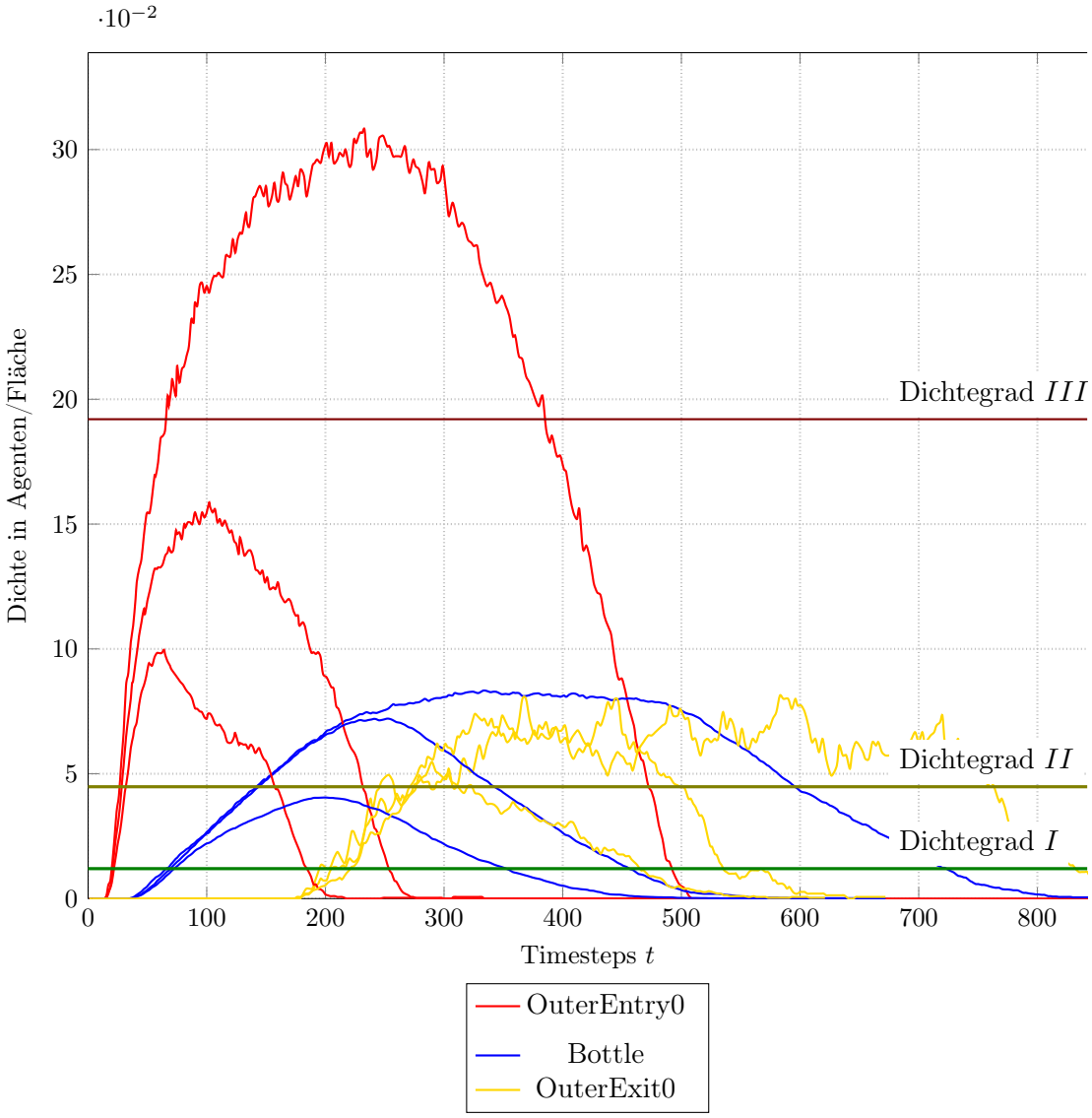

Abbildung 9: Plot der Dichte in Relation zu den Timesteps t bei Flaschenhalshöhe 100, 50 und 25

Im Plot sind von den drei Probedurchführungen DF100, DF50 und DF25 die Graphen der allgemeineren Messflächen **OuterEntry0**, **Bottle** und **OuterExit0** zu sehen. **OuterEntry0** setzt sich aus vier weiteren Messflächen und **Bottle** aus drei weiteren Messflächen zusammen. Die Graphen dieser Messflächen werden zwecks Übersichtlichkeit nicht abgebildet.

Die höchsten Dichtewerte und benötigten Timesteps erreichen jeweils die Graphen von DF25, die mittleren Werte DF50 und die niedrigsten DF100. Die für einen Durchlauf benötigten Timesteps sind bei DF100 $t = 500$, bei DF50 $t = 600$ und bei DF25 $t = 800$.

In allen Durchführungen erreichen die Graphen von **OuterEntry0** wesentlich höhere Wert als die anderen Graphen und überschreiten den Dichtegrad II, bei DF25 wird mit einem

Dichtewert von 0,3 der Dichtegrad III stark überschritten. Die benötigten Timesteps für den Durchlauf von **OuterEntry0** sind bei DF100 $t = 200$, bei DF50 $t = 300$ und bei DF25 $t = 500$. Je höher die Dichtewerte sind, für desto mehr Timesteps werden sie beibehalten.

Die Graphen von **Bottle** DF25 und DF50 überschreiten den Dichtegrad II, dabei erreichen sie ungefähr die gleichen maximalen Dichtewerte von ca. 0,07 bis 0,08. Bei DF50 wird dieser für 10 Timesteps beibehalten, bei DF25 für über 200 Timesteps. Der Graph von **Bottle** von DF100 bleibt knapp unter Dichtegrad I.

Die Graphen von **OuterExit0** überschreiten bei allen DF den Dichtegrad II. Dabei entstehen bei DF25 und DF50 ähnliche maximale Dichtewerte wie bei dem jeweiligen Graph von **Bottle**. Bei DF100 überschreitet der Graph knapp Dichtegrad II und erreicht somit höhere Werte als der Graph von **Bottle**. Bei erneuter Videodurchsicht lässt sich erkennen, dass sich bei $t = 150$ bis $t = 200$ alle Agenten innerhalb der Messfläche **Bottle** befinden, die Verteilung aber nicht gleichmäßig über die gesamte Fläche ist. Deshalb tritt bei kleineren Flächen wie **OuterExit0** eine höhere Dichte auf.

Insgesamt wird am Plot deutlich, dass die höchsten Dichtewerte im Eingangsbereich entstehen, bei DF25 sogar Dichtegrad III überschritten wird. Für den nächsten Schritt müssen deshalb die Werte der Messflächen von DF25 genauer betrachtet werden.

7.2 Betrachtung der Dichtewerte bei DF25

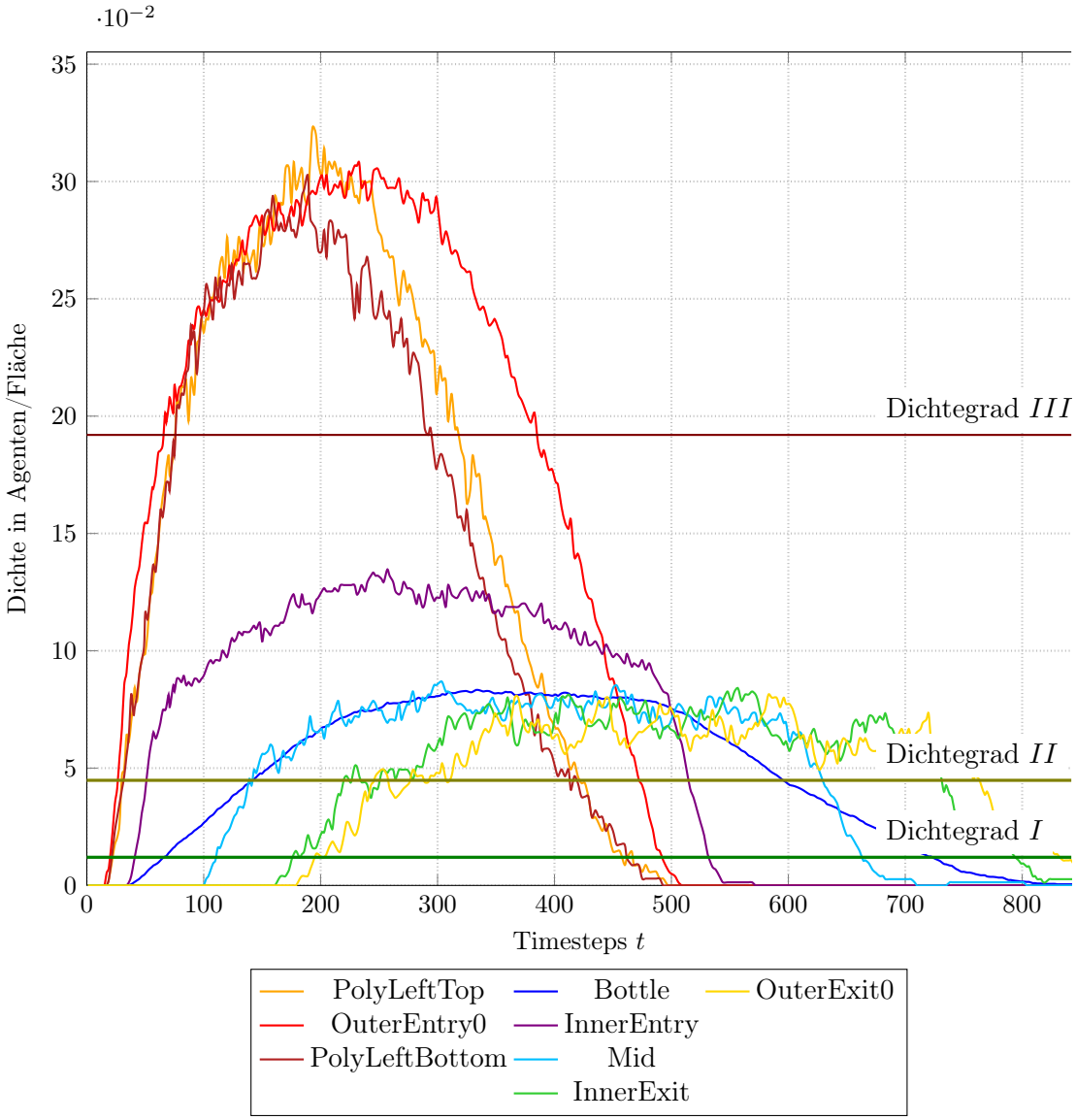

Abbildung 10: Plot der Dichte in Relation zu den Timesteps t bei Flaschenhalshöhe 25

Im Plot sind von der Probedurchführung DF25 die Graphen der Messflächen des äußeren Eingangbereiches **PolyLeftTop**, **OuterEntry0** und **PolyLeftBottom**, die des Flaschenhalses **Bottle**, **InnerEntry**, **Mid** und **InnerExit**, und des äußeren Ausgangsbereiches **OuterExit0** abgebildet. Die in **OuterEntry0** enthaltenen vier weiteren Messflächen werden, wieder zwecks Übersichtlichkeit, ausgelassen.

Wie schon im vorherigen Plot wird deutlich, dass im Eingangsbereich die höchsten Dichtewerte entstehen. Die zusätzlich abgebildeten Graphen von **PolyLeftTop** und **PolyLeftBottom** erreichen punktuell höhere Werte, wobei **PolyLeftTop** auch den Maximaldichtewert von **OuterEntry0** 0,305 mit 0,32 übertrifft. Unklar bleibt, warum **PolyLeftTop** bei einem symmetrischen Szenario höhere Werte als **PolyLeftBottom** erreicht.

Der Graph von **InnerEntry** ist von $t = 110$ bis $t = 430$ über einen Dichtewert von 0,1 und erreicht ein maximalen Werte von 0,13. Die Graphen der weiteren Messflächen des Fla-

schenhalses **Bottle**, **Mid** und **InnerExit** erreichen alle maximale Werte von 0,08. Auch der Graph von **OuterExit0** erreicht diesen Wert.

Durch den Plot wird deutlich, dass Dichtehöchstwerte im Eingangsbereich sind und sie sich im weiteren Verlauf des Flaschenhalses verringern. **InnerExit** hat bereits mit Höchstwerten von 0,13 eine geringere Dichte als die des äußeren Eingangbereiches von **OuterEntry0**, **PolyLeftTop** und **PolyLeftBottom** mit über 0,3. Der Flaschenhals im Gesamten und der äußere Ausgangsbereich haben nur noch Höchstwerte von 0,08.

Für den nächsten Schritt müssen die Messwerte des inneren und äußeren Eingangbereiches genauer untersucht werden.

7.3 Betrachtung der Dichtewerte des Eingangsbereiches bei DF25

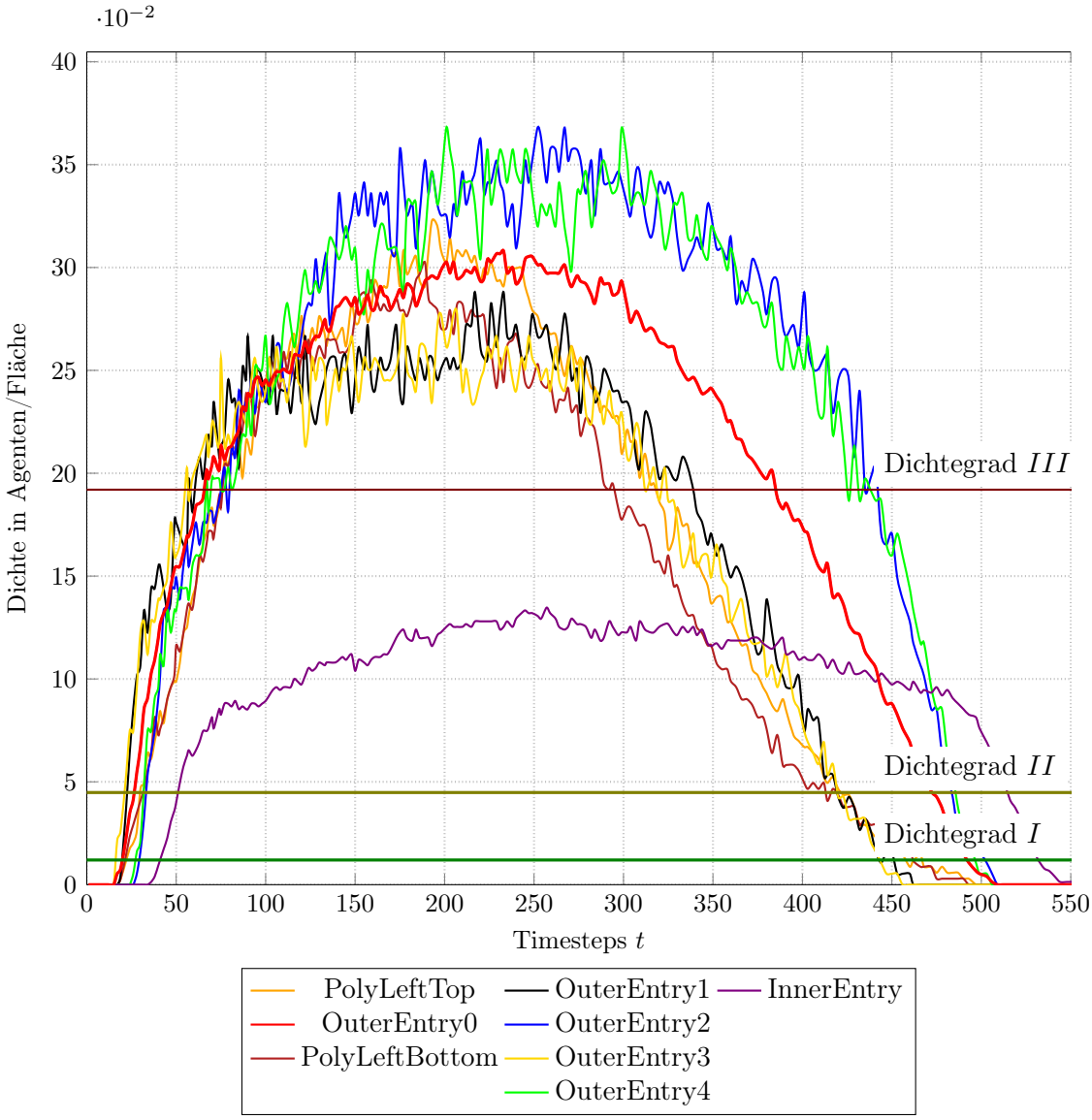

Abbildung 11: Plot der Dichte des Eingangsbereiches in Relation zu Timesteps t bei Flaschenhalshöhe 25

Im Plot sind von der Probedurchführung DF25 die Graphen aller Messflächen des äußeren Eingangsbereiches **PolyLeftTop**, **OuterExit0** bis **OuterExit4** und **PolyLeftBottom** sowie der Graph der Messfläche des inneren Eingangbereiches **InnerEntry** für $t = 0$ bis $t = 550$ abgebildet. Die Farben von **OuterEntry1** bis **OuterEntry4** entsprechen zwecks Übersichtlichkeit nicht dem Rot der Messflächenumrandung.

In den vorherigen Plots wurden die Graphen der inneren Messflächen von **OuterEntry0** nicht abgebildet. Nun wird sichtbar, dass die, unmittelbar vor dem Eingang liegenden, Messflächen **OuterEntry2** und **OuterEntry4** mit Dichtewerte von über 0,35 höhere Werte als die, links davor liegenden, Messflächen **OuterEntry1** und **OuterEntry3**, mit Werten von 0,27, haben.

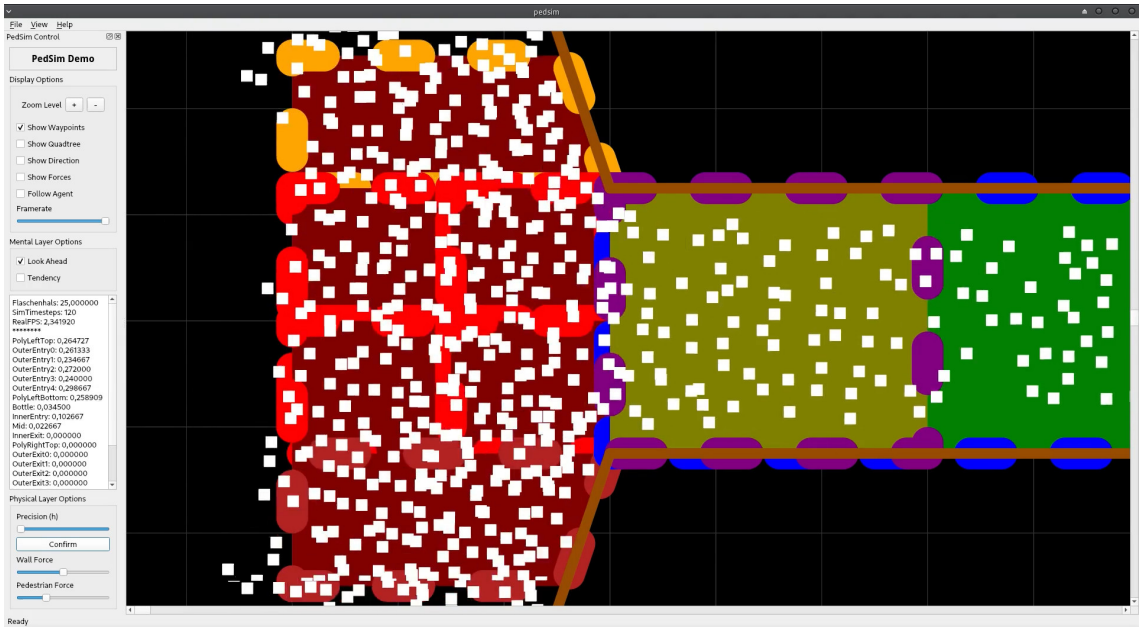

Abbildung 12: Screenshot des Videos CloseUpDF25.mp4 bei $t = 120$

7.4 Zwischenfazit

Für DF25 lassen sich also folgende Bereiche nach Maximalwerte ordnen, die den kritischen Dichtegrad III überschreiten:

1. der Bereich direkt am Eingang des Flaschenhalses mit **OuterEntry2** und **OuterEntry4**,
2. der Bereich an den Schrägen mit **PolyLeftTop** und **PolyLeftBottom** und
3. der Bereich kurz hinter dem Eingang des Flaschenhalses mit **OuterEntry1** und **OuterEntry3**.

Im nächsten Schritt wird untersucht, wie sich die Maximalwerte im Vergleich bei unterschiedlichen Flaschenhalshöhen verhalten.

7.5 Vergleich der maximalen Dichtewerte aller Durchführungen

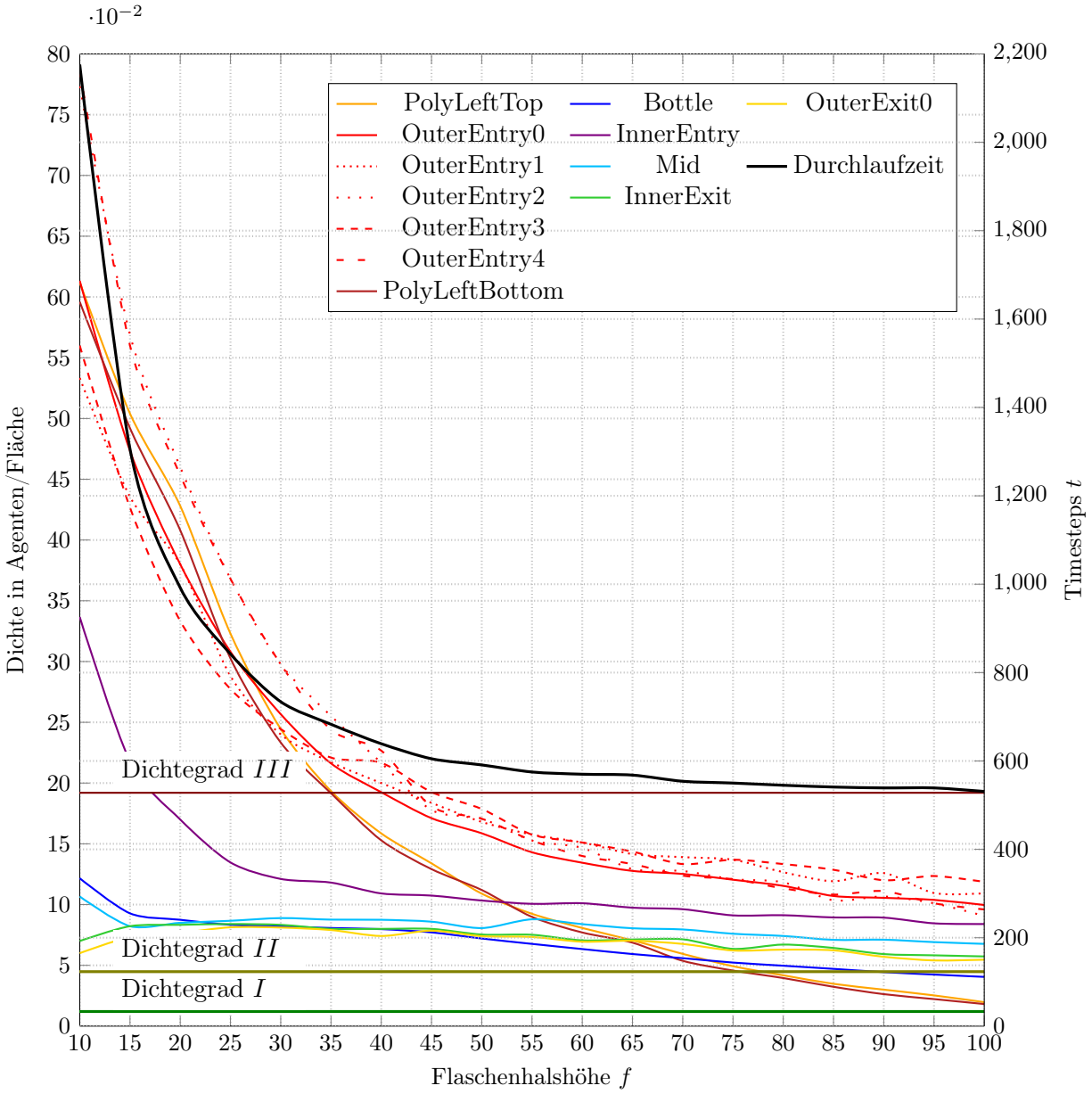

Abbildung 13: Plot der Maximalwerte der Dichte in Relation zur Flaschenhalshöhe 10 bis 100

Der Plot stellt die Graphen der Maximalwerte der Dichte (linke y-Achse) aller Messflächen und die Laufzeit (rechte y-Achse) in Relation zur Flaschenhalshöhe dar.

Dichte und Laufzeit fallen in der ersten Hälfte stark ab, in der zweiten Hälfte nur noch gering. Die Laufzeit fällt von $f = 10$ bis $f = 50$ mit $t = 2200$ bis $t = 600$ stark ab. Von $f = 50$ bis $f = 100$ nähert sie sich auf $t = 500$ an und sinkt somit kaum noch.

Außer **PolyLeftTop** und **PolyLeftBottom** bleiben alle Dichtewerte mindestens im Grad II.

Von $f = 15$ bis $f = 100$ sinken die Dichte von **Bottle**, **Mid**, **InnerExit** und **OuterExit0** von 0,09 langsam auf 0,05 ab. Dagegen fällt **InnerEntry** bei $f = 10$ bis $f = 15$ von 0,35

stark ab, unterschreitet Dichtegrad III und fällt bis $f = 30$ auf 0,12 ab. Von $f = 30$ bis $f = 100$ sinkt die Dichte langsam auf 0,11 ab und bleibt somit höher als die anderen Dichtewerte des Flaschenhalses.

Von $f = 10$ bis $f = 35$ sind die Dichtewerte des äußeren Eingangsbereiches alle im Dichtegrad III, sie fallen dabei von 0,8 auf 0,2 stark ab. Von $f = 35$ bis $f = 45$ wird der Dichtegrad III unterschritten, **OuterEntry0** bis **OuterEntry4** sinken auf 0,1 bis 0,12 ab, **PolygonLeftTop** und **PolygonLeftBottom** sinken stärker und unterschreiten ab $f = 75$ Dichtegrad I. Dies liegt vermutlich daran, dass bei kleineren f ein Rückstau der Agenten, vom direkten Eingangsbereich bis zu den Bereich der Schrägen, entsteht.

Im nächsten Schritt werden die Graphen des Eingangsbereiches von $f = 10$ bis $f = 50$ geplottet, da dort die Werte im Dichtegrad III liegen.

7.6 Vergleich der maximalen Dichtewerte für $f = 10$ bis $f = 50$

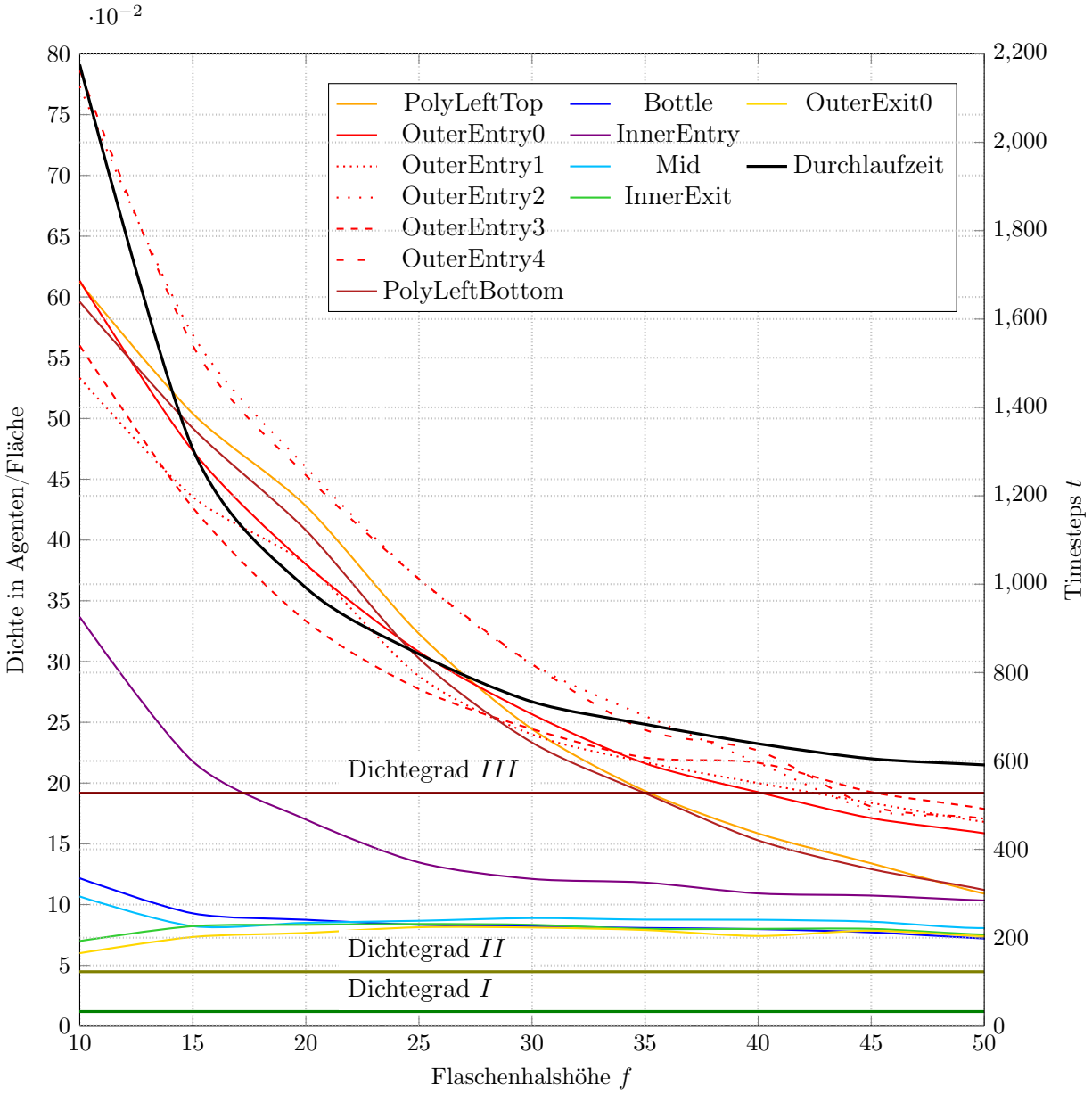

Abbildung 14: Plot der Maximalwerte der Dichte in Relation zur Flaschenhalshöhe 10 bis 50

In dem Plot sind die selben Graphen wie im vorherigen Plot abgebildet, allerdings für den Bereich von $f = 10$ bis $f = 50$.

Alle Dichtewerte sinken im Verlauf, außer die von **InnerExit** und **OuterExit0**, diese steigen von $f = 10$ bis $f = 15$ von 0,06 auf 0,08 an. Gleichzeitig sinken **Bottle** und **Mid** von 0,12 auf 0,08 ab, während sie im weiteren Verlauf von $f = 15$ bis $f = 50$ auf 0,08 bleiben. Dies lässt sich mit dem in Abschnitt 6.3 erwähnten Artefakt erklären: durch das hin und her prallen an den gegenüberliegenden Wänden des Flaschenhalses benötigen die Agenten mehr Timesteps bis sie **InnerExit** und **OuterExit0** erreichen. Da gleichzeitig weitere Agenten in den Flaschenhals gelangen, entsteht eine höhere Dichte. Zudem können weniger Agenten gleichzeitig den Flaschenhals verlassen, weswegen bei **InnerExit** und

OuterExit0 die Maximaldichte mit 0,07 geringer ist, als die Maximaldichte von **Bottle** und **Mid** mit 0,12.

Die höchsten Werte erreichen **OuterEntry2** und **OuterEntry4** bei $f = 10$ mit 0,8, **OuterEntry0**, **PolyLeftTop** und **PolyLeftBottom** erreichen 0,6, **OuterEntry1** und **OuterEntry3** erreichen 0,55. Alle unterschreiten bis $f = 45$ Dichtegrad III, jedoch mit kleineren Unterschieden: **OuterEntry2** und **OuterEntry4** sinken von 0,8 auf 0,2 am steilsten, **OuterEntry0** in der ersten Hälfte um bis zu 0,05 mehr als **PolyLeftTop** und **PolyLeftBottom**, in der zweiten Hälfte sinken **PolyLeftTop** und **PolyLeftBottom** dagegen schneller und unterschreiten bei $f = 35$ Dichtegrad III. Bei $f = 40$ bis $f = 45$ unterschreiten **OuterEntry0** bis **OuterEntry4** Dichtegrad III.

Auffällig ist, dass trotz des symmetrischen Szenarios bei den symmetrischen Messflächen **PolyLeftTop** und **PolyLeftBottom**, **OuterEntry1** und **OuterEntry3** sowie **OuterEntry2** und **OuterEntry4** im Bereich $f = 15$ bis $f = 25$ die jeweils obere Messfläche Abweichungen von bis zu 0,05 hat.

Besonders deutlich wird an dem Plot, dass, je geringer die Flaschenhalshöhe ist und desto näher die Messfläche im Eingangsbereich liegt, umso höhere Dichten entstehen. Desweiteren scheint die Dichte vom Eingangsbereich bis zur Mitte des Flaschenhalses stark zu sinken, **InnerEntry** bleibt ab $f = 20$ im Dichtegrad II, **Bottle**, **Mid**, **InnerExit** und **OuterExit0** sind mit meistens 0,08 im unteren Bereich des Dichtegrades II. Zudem wird bei $f = 10$ bis $f = 45$ der kritische Dichtegrad III teilweise extrem überschritten.

7.7 Fazit der Diskussion

Mit diesem Abschnitt können die beiden Forschungsfragen **F 2.1** und **F 2.2** beantwortet werden: Für $f = 10$ bis $f = 100$ entstehen im direkten Eingangsbereiche des Flaschenhalses die höchsten Werte, gemessen durch **OuterEntry1** und **OuterEntry3**. Die zweit höchsten Werte entstehen direkt dahinter und an den zulaufenden Schrägen des Flaschenhalses: Für $f = 10$ bis $f = 25$ an den zulaufenden Schrägen des Flaschenhalses, gemessen durch **PolyLeftTop** und **PolyLeftBottom** und von $f = 30$ bis $f = 100$ hinter dem direkten Eingangsbereich, gemessen durch **OuterEntry2** und **OuterEntry4**.

Um für das erstellte Flaschenhalszenario unterhalb des kritische Dichtegrades III zu bleiben, muss die Flaschenhalshöhe mindestens $f = 45$ betragen, geringere Höhen erzeugen extrem hohe Dichtewerte.

8 Fazit

Ziel der Arbeit war es aufzeigen, wo in einem Crowd-Desaster-Szenario die höchsten Dichtewerte auftreten. Die dafür aufgestellten Forschungsfragen konnten beantwortet werden.

F 1.1 Wie kann die Dichte von Agenten pro Fläche innerhalb von PEDSIM gemessen werden?

Dafür wurde PEDSIM um eine neuartige Dichtefunktion erweitert, die die Methode **getNeighbors()** mit einer zusätzlich erstellten Koordinatenprüfung nutzt. Auf Basis dieser Dichtefunktion wurde die Klasse **Detector** erstellt, dessen Objekte bei vorgegebenen Flächen die Dichte messen. Mögliche Messflächen sind quadratisch, rechteckig und in Sonderfällen auch trapezförmig.

F 1.2 Wie können die erhobenen Dichtewerte graphisch in Echtzeit angezeigt werden?

Um die Dichte in Echtzeit graphisch darzustellen, werden die Messflächen ihrer Dichte entsprechend eingefärbt. Die Werte für die Einfärbungen wurden dem Praxisprojekt entnommen, wo die Dichte in Relation zur Quadtreetiefe gesetzt wurde.

F 1.3 Wie können die erhobenen Dichtewerte graphisch in Echtzeit angezeigt werden?

Für die numerische Darstellung wurde eine Ausgabe der Dichtewerte per GUI und per CSV-Datei realisiert.

F 2.1 und **F 2.2** Wo und wann entstehen bei einem Flaschenhalsszenario hohe bis gefährliche Dichtewerte?

Es wurde zunächst ein Flaschenhalsszenario konstruiert. In dem Flaschenhalsszenario wurden die Messflächen der erstellten Klasse **Detector** im Eingangsbereich des Flaschenhalses, im Flaschenhals selber und im Ausgangsbereich des Flaschenhalses positioniert.

Das Szenarios wurde mit 1000 Agenten in 19 Durchläufen mit Flaschenhalshöhen von $f = 100$ bis $f = 10$ durchgeführt. Die Durchführungen wurden dabei gefilmt und die erstellten Daten geplottet. Bei einem Vergleich der Dichtewerte konnte gezeigt werden, dass die höchsten Dichtewerte im direkten Eingangsbereich des Flaschenhalses und an den Schrägen im erweiterten Eingangsbereich des Flaschenhalses entstehen. Ab einer Flaschenhalshöhe von $f = 45$ entstehen gefährliche Dichtewerte im Grad III.

9 Ausblick

Anknüpfend an die Arbeit, könnten Vertiefungen im Bereich Dichtemessung und Testdurchführung durchgeführt werden.

9.1 Dichtemessung

- Die Zwischenlösung in der Klasse **Detector** für polygone Messflächen, die Schrägen enthalten, könnte um eine Punktprüfung für polygone Flächen erweitert werden.
- Es könnte eine Erzeugung der Messflächen per XML-Szenario erstellt werden, ähnlich, wie die der Agenten, Wegpunkte und Obstacle
- Anstatt die Dichte über stationäre Messflächen zu messen, könnte eine Dichtemessung für ausgewählte Agenten, innerhalb eines gewissen Radius um diese herum, erstellt werden.

9.2 Testdurchführungen

Für das Flaschenhalsszenario wurde betrachtet, wie die Flaschenhalshöhe die Dichte beeinflusst. Desweiteren könnte betrachtet werden, inwiefern die folgende Eigenschaften des Szenarios die Dichte beeinflussen:

- die Entfernung vom Startbereich der Agenten bis zum Flaschenhalseingang,
- die Agentenanzahl,
- die Agentengeschwindigkeit und
- entgegen laufende Agenten.

Literatur

[1] Benedikt Kleinmeier and Benedikt Zönnchen and Marion Gödel and Gerta Köster. Vadere: An open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4, 2019.

[2] Jürgen Gerlach. Fachliche Aufbereitung von Ursachen der tragischen Ereignisse bei der Loveparade Duisburg 2010. `https://www.svpt.uni-wuppertal.de/fileadmin/bauing/svpt/Loveparade_2010/Loveparade_Aufarbeitung_Gerlach_vorl_Fassung.pdf`. Vorläufige Fassung, Stand 10. Juli 2020. Zugriff am 31. Dezember 2020.

[3] Christian Gloor. libPEDSIM Documentation: ped_tree.cpp. `http://pedsim.silmaril.org/documentation/libpedsim/latest/ped__tree_8cpp_source.html`. Zugriff am 31. Dezember 2020.

[4] Christian Gloor. PEDSIM. A Pedestrian Crowd Simulation System .Motivation, Usage, Installation and Library Documentation. `http://pedsim.silmaril.org/documentation/libpedsim/libpedsim.pdf`. Zugriff am 31. Dezember 2020.

[5] Christian Gloor. *DISTRIBUTED INTELLIGENCE IN REAL WORLD MOBILITY SIMULATIONS*. PhD thesis, ETH Zürich, 2005.

[6] Christian Gloor, Pascal Stucki, and Kai Nagel. Hybrid techniques for pedestrian simulations. *International Conference on Cellular Automata*, pages 581–590, 2004.

[7] Marvin Nicholas Hallweger. Konzeption und Evaluation eines Job-basierten Entity-Component-Systems anhand einer Massenpaniksimulation auf Basis von Unity DOTS. Bachelorthesis, TH Köln, 2020.

[8] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Phys. Rev. E*, 51:4282–4286, May 1995.

[9] Dirk Helbing and Pratik Mukerji. Crowd disasters as systemic failures: Analysis of the love parade disaster. *EPJ Data Science*, 1, 06 2012.

[10] Tim Kurtz. Erweiterung der Crowd Simulation PEDSIM mit einer graphischen Darstellung der Dichte von Agenten. Praxisprojekt, TH Köln, 2020.

[11] Thomas Richards. A REVIEW OF SOFTWARE FOR CROWD SIMULATION. `https://urban-analytics.github.io/dust/docs/ped_sim_review.pdf`, March 2020. Zugriff am 31. Dezember 2020.

[12] Zönnchen, Benedikt and Köster, Gerta. A parallel generator for sparse unstructured meshes to solve the eikonal equation. *Journal of Computational Science*, 32:141–147, 2019.

Anhang

Videos

- Webordner mit allen Durchführungen: `https://th-koeln.sciebo.de/s/oXCuFqfcN5jt7sq`
- Alle Durchführungen in einem Gesamtvideo: DF100-10.mp4: `https://th-koeln.sciebo.de/s/gwdN1zdbqWcsCx4`
- Beispielszenario.mp4: `https://th-koeln.sciebo.de/s/usxe9PwKnOkgJ51`
- DF100.mp4: `https://th-koeln.sciebo.de/s/f4iJGWFIzrSekvB`
- DF50.mp4: `https://th-koeln.sciebo.de/s/INLadUDu8bHNqcq`
- DF25.mp4: `https://th-koeln.sciebo.de/s/Zrdw7RtnYUOOUvp`
- AusschnittDF10.mp4:`https://th-koeln.sciebo.de/s/EcRhODKlkziA3dN`
- DF25CloseUp.mp4: `https://th-koeln.sciebo.de/s/6ig5ZdNv2YYjIlF`

Ergänzung zu Abschnitt 3.1: Test und Untersuchung von getNeighbors()

Test von getNeighbors()

Um die Funktionsweise der Methode von getNeighbors() zu testen, wird ausgegeben:

- Per **getNeighbors(0,0,1).size()** die Anzahl der erfassten Agenten innerhalb des Punktes (0|0) mit dem Suchradius 1.
- Per **getAllAgents.size()** die Gesamtanzahl der Agenten.
- Per Iterator die ID und Positionskoordinaten aller Agenten, die von **getNeighbors(0,0,1)** erfasst wurden.

Die Ausgabe lautet:

```
getNeighbors(0,0,1).size(): 6 von 100
86: 38.4726|0.546168
27: -41.8968|2.70858
68: 43.9095|1.03808
14: -37.9381|3.29708
11: -35.5116|3.93746
74: 33.4746|1.23933
```

Listing 2: Ausgabe von getNeighbors(0,0,1)

Der erstellte Screenshot dazu, die Agenten der Ausgabe wurden rot makiert.
Die Koordinaten des Hindernisses sind:

- oben-links: $(-20|-20)$
- oben-rechts: $(-20|20)$

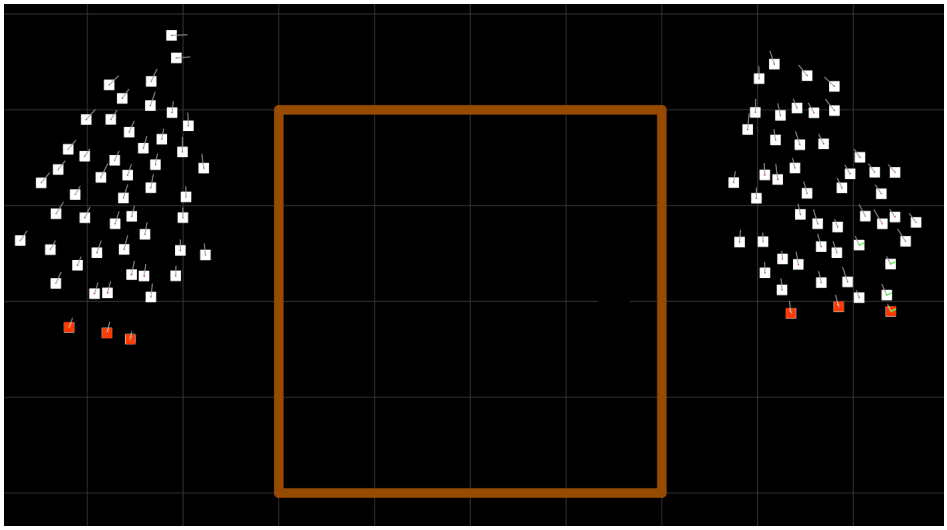

Abbildung 15: Bearbeiteter Screenshot der Ausführung von getNeighbors(0,0,1)

- unten-links: $(20| - 20)$
- unten-rechts: $(20|20)$

Anhand der Koordinaten und des Screenshot wird deutlich, dass der Aufruf **getNeighbors(0,0,1)** Agenten erfasst, die außerhalb des Suchradius 1 liegen. Die Methode **getNeighbors()** kann also nicht ohne weiteres genutzt werden.

Untersuchung von getNeighbors()

Um den Grund für die fehlerhafte Agentenerfassung der Methode **getNeighbors()** zu verstehen, wird diese untersucht.

```
set<const Ped::Tagent*> Ped::Tscene::getNeighbors(double x, double y,
    ↪ double dist) const {
    // if there is no tree, return all agents
    if (tree == NULL)
        return set<const Ped::Tagent*>(agents.begin(), agents.end());

    // create the output list
    list<const Ped::Tagent*> neighborList;
    getNeighbors(neighborList, x, y, dist);

    // copy the neighbors to a set
    return set<const Ped::Tagent*>(neighborList.begin(), neighborList.end()
        ↪ );
}
```

Listing 3: Methode getNeighbors() in libpedsim/ped_scene.cpp

- Die Methode **getNeighbors()** wird mit den Koordinaten **x**, **y** und dem Suchradius **dist** aufgerufen.
- Wenn kein Baum vorhanden ist, werden alle Agenten der Szene als Set zurückgegeben.
- Die leere Agentenliste **neighborList** wird erstellt.
- Die namensgleiche, aber geschützte, Methode **getNeighors()** wird mit den Parametern **neighborList**, **x**, **y** und **dist** aufgerufen.
- **neighborList** wird in ein Set kopiert, das Set ist der Rückgabewert.

```
void Ped::Tscene::getNeighbors(list<const Ped::Tagent*>& neighborList,
    ↪ double x, double y, double dist) const {
    stack<Ped::Ttree*> treestack;

    treestack.push(tree);
    while(!treestack.empty()) {
        Ped::Ttree *t = treestack.top();
        treestack.pop();
        if (t->isleaf) {
            t->getAgents(neighborList);
            //ausgeben der Baumwerte...
        }
        else {

            if (t->tree1->intersects(x, y, dist)) treestack.push(t->tree1);
            if (t->tree2->intersects(x, y, dist)) treestack.push(t->tree2);
            if (t->tree3->intersects(x, y, dist)) treestack.push(t->tree3);
            if (t->tree4->intersects(x, y, dist)) treestack.push(t->tree4);
        }
    }
}
```

Listing 4: Geschützte Methode getNeighbors() in libpedsim/ped_scene.cpp

- Die geschützte Methode **getNeighbors()** wird mit **neighborList**, **x**, **y** und **dist** aufgerufen.
- Der **Ped::Ttree**-Stack **treestack** wird erstellt.
- Die while-Schleife läuft, solange **treestack** nicht leer ist. Das erste Element von **treestack** wird in **t** gespeichert und aus dem Stack entfernt.
- Wenn **t** ein leaf ist, also keine Kindsknoten hat, werden alle Referenzen der Agenten von **t** in **neigborList** gespeichert.

- Sonst werden jeweils alle vier Teilbäume von **t** mit **intersects(x, y, dist)** geprüft. Ist die Prüfung wahr, wird die Referenz des Teilbaumes nach **treestack** kopiert.

```
bool Ped::Ttree::intersects(double px, double py, double pr) const {
    if (((px+pr) > x) && ((px-pr) < (x+w)) && ((py+pr) > y) && ((py-pr) < (
        ↪ y+h))) {
        return true; // x+-r/y+-r is inside
    } else
        return false;
}
```

Listing 5: Methode intersects() in libpedsim/ped_trees.cpp

- Die Methode **intersects()** wird mit den Parametern **px**, **py** und **dist** aufgerufen.
- **x** und **y** sind die Koordinaten des Ausgangspunktes, **w** die Länge und **b** die Breite des Rechteckes, die der Tree behandelt.
- Es wird geprüft, ob der Punkt $(px|py)$ mit Suchradius r in dem Rechteck liegt, das der Tree behandelt.

Durch den Aufruf von **getNeighbors(x,y,dist)** werden alle Agenten ausgegeben, die in dem Tree liegen, der den Punkt $(x|y)$ mit dem Suchradius *dist* enthält. Es findet keine genauere Überprüfung der Agentenkoordinaten statt.

Deutlich wird der Umstand an folgender Abbildung:

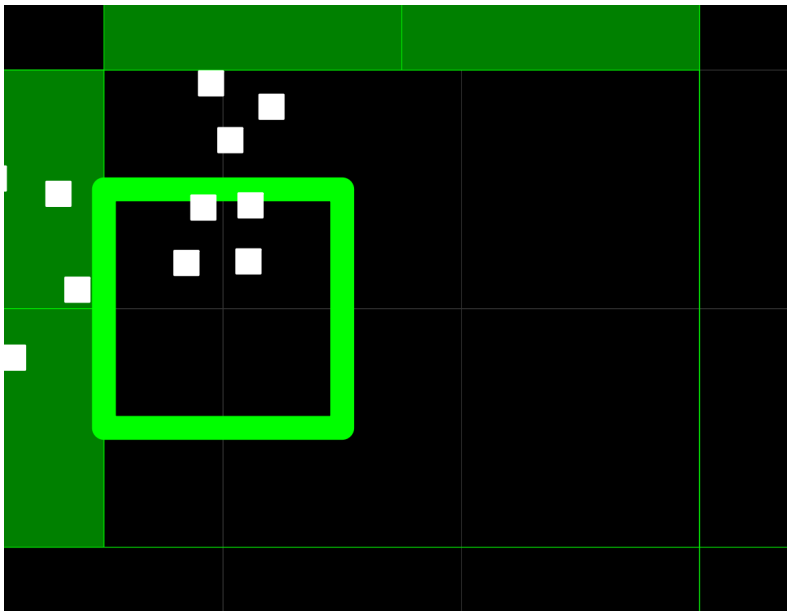

Abbildung 16: Screenshot der Ausführung von getNeighbors(0,0,5)

- Das Quadrat mit der grünen Umrandung stellt das Suchfeld von **getNeighbors(0,0,5)** dar.
- Das darum liegende Rechteck stellt den Quadtree dar.
- Innerhalb des Quadtree sind sieben Agenten.
- Innherhalb des Suchfeldes sind vier Agenten.
- Die Ausgabe von **getNeighbors(0,0,5).size()** beträgt sieben.

Die Methode **getNeighbors()** liefert nur ein Teilergebnis. Dies muss für eine genaue Zählung der Agenten nochmals, auf den Suchbereich hin, überprüft werden.

Ergänzung zu Abschnitt 3.2.1: Herleitung der Suchfelder der Detektorklasse

Herleitung Quadratsform zur Rechtecksform

Die Quadratsform besteht aus dem Suchpunkt $(mx|my)$ mit dem Suchradius r. Daraus ergibt sich ein Suchfeld:

- oben-links: $(mx - r|my - r)$
- oben-rechts: $(mx + r|my - r)$
- unten-links: $(mx - r|my + r)$
- unten-rechts: $(mx + r|my + r)$
- Mittelpunkt: $(mx|my)$

Mit diesen lassen sich die zwei Strecken beschreiben:

- von oben-links nach oben-rechts:

$$\begin{pmatrix} (mx + r) - (mx - r) \\ (my - r) - (my - r) \end{pmatrix} = \begin{pmatrix} 2r \\ 0 \end{pmatrix}$$

- von oben-links nach unten-links:

$$\begin{pmatrix} (mx - r) - (mx - r) \\ (my + r) - (my - r) \end{pmatrix} = \begin{pmatrix} 0 \\ 2r \end{pmatrix}$$

Die Rechtecksform besteht aus:

- dem Ausgangspunkt oben-links: $(x|y)$,

- dem Wert w, die Länge der Strecke von oben-links nach oben-rechts und
- dem Wert h, die Länge der Strecke von oben-links nach unten-links.

Nun kann die Rechtecksform gebildet werden:

- oben links: $(x|y) = (mx - r|my - r)$
- Länge: $w = 2r$
- Breite: $h = 2r$

Hieraus können die Punkte des Suchfeldes gebildet werden:

- oben-links: $(x|y)$
- oben-recht: $(x + w|y)$
- unten-links: $(x + h|y)$
- unten-rechts: $(x + h|y + w)$

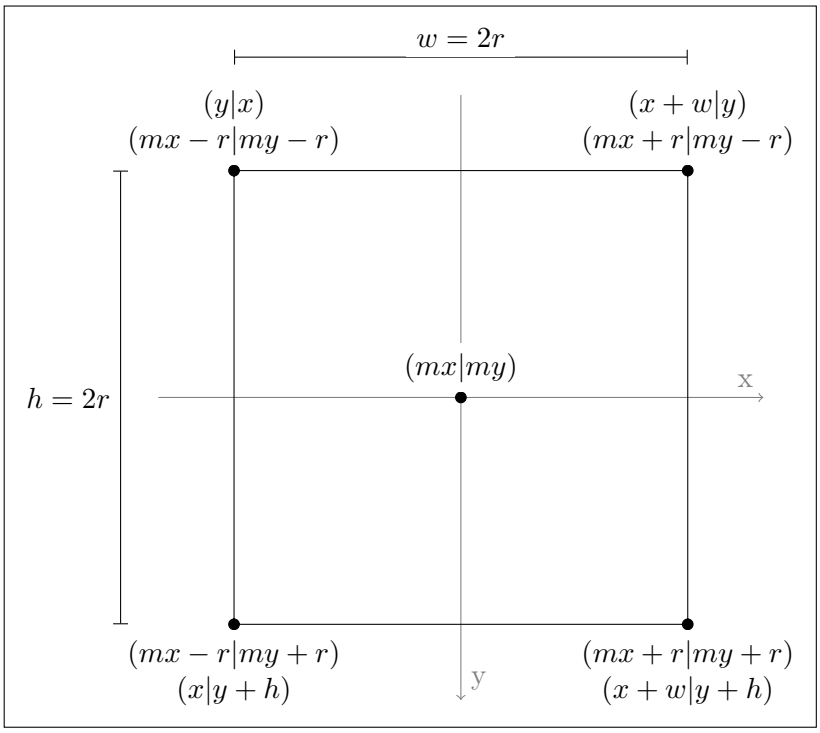

Abbildung 17: Von Quadratsform zur Rechtecksform

Herleitung Rechtecksform zur Quadratsform

Um aus der Rechtsecksform die Quadratsform zu bilden, wird das kleinste umgebende Quadrat des Rechtecks gebildet.

Die Rechtecksform besteht aus:

- dem Ausgangspunkt oben links: $(x|y)$
- dem Wert w, die Länge der Strecke von oben-links nach oben-rechts
- dem Wert h, die Länge der Strecke von oben-links nach unten-links

Daraus werden die Punkte des Suchfeldes gebildet:

- oben-links: $(x|y)$
- oben-rechts: $(x + w|y)$
- unten-links: $(x|y + h)$
- unten-rechts: $(x + w|y + h)$

Aus den Punkten wird das kleinste umgebende Quadrat gebildet:

- Die größere Seitenlänge des Rechtecks wird die Seitenlänge des Quadrates.
- Die Länge des Suchradius ist die Hälfte der Seitenlänge des Quadrates, also die Hälfte der größeren Seitenlänge des Rechtecks:

$$r = \begin{cases} \frac{w}{2} & \text{if } w > h \\ \frac{h}{2} & \text{if } h > w \end{cases}$$

- Mittelpunkt $(mx|my) = (x + r|y + r)$

Die nachfolgende Abbildung stellt den Fall $w > h$ dar.

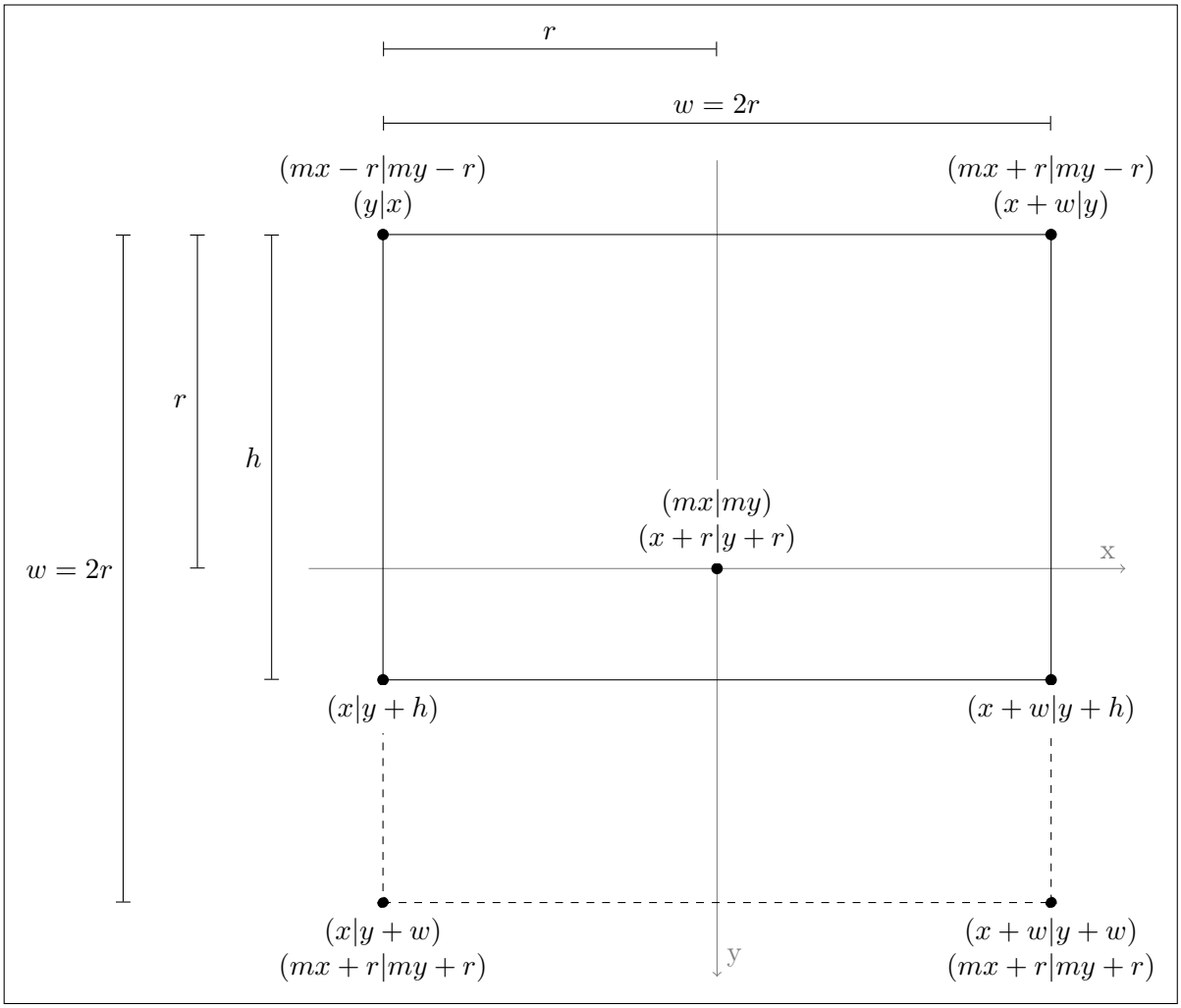

Abbildung 18: Kleinstes umgebendes Quadrat bei $w > h$

Ergänzung zu Abschnitt 3.2.1: Quellcode der erstellten Detektorklasse

```
extern Config config;
extern Stats stats;

Detector::Detector(string name, QColor color, double dmx, double dmy,
    ↪ double dr, QGraphicsScene *gscene, Ped::Tscene *pscene){
        graphicsscene = gscene;
        scene = pscene;
        this->mx = dmx;
        this->my = dmy;
        this->r = dr;
        this->x = mx-r;
        this->y = my-r;
        this->w = 2*r;
        this->h = 2*r;
```

```
        this->area = w*h;
        numberOfAgents = 0;
        density = 0;
        this->color = color;
        this->name = name;

        //quadratwerte in rechteckform x,y,w,h umwandeln, da draw in
            ↪ Rechteckform zeichnet

        //Detektoren werden in einer Deque gespeichert, weil FIFO und
            ↪ Iterator
        stats.DetectorQueue.push_back(this);
        //if (isVisible) this->draw();
}


Detector::Detector(string name, QColor color, double dx, double dy, double
    ↪ dw, double dh, QGraphicsScene *gscene, Ped::Tscene *pscene) {
        graphicsscene = gscene;
        scene = pscene;

        this->x = dx;
        this->y = dy;
        this->w = dw;
        this->h = dh;

        if (dw > dh) {
                this->mx = dx+0.5*dw;
                this->my = dy+0.5*dw;
                this->r = dw*0.5;
        } else {
                this->mx = dx+0.5*dh;
                this->my = dy+0.5*dh;
                this->r = dh*0.5;
        }

        this->area = w*h;

        /*Suchqadrat fuer Quadtree bilden, dafuer kleinstes umgebendes
            ↪ Rechteck bilden
         * die laengere Rechteckseite/2 wird der Suchradius
         * daraus Mittelpunkt und des Suchqadrat bilden
```

```
         */

        this->name = name;
        this->color = color;

        numberOfAgents = 0;
        density = 0;
        //Detektoren werden in einer Queque gespeichert, weil FIFO und
            ↪ Iterator
        stats.DetectorQueue.push_back(this);
}

int Detector::count() {
        int counter;
        for (Ped::Tagent const* agent : scene->getNeighbors(mx, my, r)) {

                double px = agent->getPosition().x;
                double py = agent->getPosition().y;

                if(((x) < px) && (px < (x+w)) && ((y) < py) && (py < (y+h)))
                    ↪ {
                        counter++;
                }
        }
        numberOfAgents = counter;

        return numberOfAgents;

        /* Suche mit getneighbors per Succhquadrat, wurde fuer rechteckige
            ↪ Suchfelder im Konstruktor umgewandelt
         * Ergebnisse werden per Rechteck ueberprueft, wurde fuer
             ↪ quadratische Suchfelder im Konstruktor umgewandelt
         */
}

void Detector::draw() {
        //loescht bestehndes Graphicitem und zeichnet neues, beim
            ↪ allerersten Durchlauf gibt es keins, QT catcht das
            ↪ automatisch, statt jedes Mal ne Pruefung durchzufuehren
        graphicsscene->removeItem(rect);
        graphicsscene->removeItem(poly);

        QBrush b = QBrush();
```

```
        if (density < 0.012) { b = QBrush(Qt::transparent);}
        else if ((density > 0.012) && (density < 0.048)) { b = QBrush(Qt::
            ↪ darkGreen);}
        else if (density > 0.048 && density < 0.192) { b = QBrush(Qt::
            ↪ darkYellow);}
        else if (density > 0.192 && density < 0.768) { b = QBrush(Qt::
            ↪ darkRed);}

        if (isPolygon) {
                poly = graphicsscene->addPolygon(polygon, QPen(this->color,
                    ↪ 3, Qt::DotLine, Qt::RoundCap, Qt::RoundJoin),b);
                poly->setZValue(-80);

        } else {
                //rect = graphicsscene->addRect(QRectF(this->x,this->y,this->
                    ↪ w,this->h), QPen(Qt::blue, 2, Qt::DotLine, Qt::
                    ↪ RoundCap, Qt::RoundJoin), b);
                rect = graphicsscene->addRect(QRectF(this->x,this->y,this->w,
                    ↪ this->h), QPen(this->color, 3, Qt::DotLine, Qt::
                    ↪ RoundCap, Qt::RoundJoin), b);
                rect->setVisible(true);
                rect->setZValue(-80); // Quadtreefuellung liegt bei -100 bis
                    ↪ -90, wenn hier groesserer Wert ist der Detektor
                    ↪ ueberhalb von Quadtrees
        }
}

void Detector::cleaner() {
        graphicsscene->removeItem(rect);
}

double Detector::getDensity() {
        this->count();
        this->density = numberOfAgents/area;
        if (isVisible) draw();
        return density;
}
```

Listing 6: Erstellte Klasse Detector in demoapp/src/detector.cpp

Ergänzung zu Abschnitt 4: Erstellung der Ausgabe

Um Bildschirmvideos und Ausgabedaten synchron zu halten, müssen der Timesteps-Wert t in der GUI und in der Ausgabedatei augegeben werden. Dafür wird die bisherige GUI-

Ausgabe betrachtet.

Bisherige GUI-Ausgabe

```
Control::Control(QWidget *parent, bool admin) : QDockWidget(parent) {
[...]
    connect(ui.simhSlider, SIGNAL(valueChanged(int)), this, SLOT(simh(int))
        ↪ );
[...]
}
[...]
void Control::simh(int value) {
    config.simh = 0.01f*value;
    ui.simhLabel->setText(QString("Precision␣(h=%1)").arg(config.simh));
}
[...]
```

Listing 7: Ausschnitt aus demoapp/src/control.cpp

Mit der Klasse **Control** werden Elemente der GUI manipuliert, beispielsweise Labeltexte:

- Durch die Verschiebung des Sliders **simhSlider** wird das Signal **valueChanged(int)**, mit dem Wert der veränderten Sliderposition, emittiert.
- **SLOT** ruft die Methode **simh()** mit dem geänderten Wert auf.
- In **simh()** wird Wert in **config.simh** und als QString im Text von **simhLabel** gespeichert.

Hinleitung

Da der Timesteps-Wert und die erfassten Werte der Detektoren direkt in der GUI ausgegegeben werden sollen, wird ebenfalls mit **SIGNAL-SLOT** gearbeitet. Dazu wird eine Objektrefenz benötigt, auf die, von Klasse **Control** und Klasse **Scene**, zugegriffen werden kann, da:

- die Detektorobjekte innerhalb der Klasse **Scene** erstellt werden und von dort die Dichte erfassen,
- die Klasse **Control** die Werte an GUI ausgibt.

Daher wird ein globales Objekt zum Transfer benötigt. Dafür wird die Klasse **Stats** erstellt. Das globale **Stats**-Objekt **stats** wird mit dem Befehl **extern Stats stats** in der Klasse **Main** erstellt.

inputStringToUi() der Klasse Stats

```
public:
        void inputStringToUi(string input);
signals:
        void valueChanged(QString uiText);
```

Listing 8: Erstellte Deklarationen in demoapp/src/stats.h

```
void Stats::inputStringToUi(string input) {
        uiText = QString::fromStdString(input);
        emit valueChanged(uiText);
}
```

Listing 9: Erstellte Methode inputStringToUi() in demoapp/src/stats.cpp

- Mit der Methode **inputStringToUi()** wird ein String eingelesen, in ein **QString**-Objekt umgewandelt und in der Klassenvariable **uiText** gespeichert.
- Das Signal **valueChange()** wird mit der veränderten **uiText** emittiert.

Erweiterung in der Klasse Control

```
Control::Control(QWidget *parent, bool admin) : QDockWidget(parent) {
   [...]
    connect(&stats, SIGNAL(valueChanged(QString)), this, SLOT(time(QString)
        ↪ ));
}
[...]
void Control::time(QString text) {
        ui.textBrowser->setPlainText(text);
}
```

Listing 10: Erweiterung in demoapp/src/control.cpp

- Wenn das globale Objekt **stats** ein **valueChanged**-Signal emittiert, wird durch **SLOT** die Methode **time(QString)** aufgerufen.
- In **time(QString text)** wird der Inhalt von **text** in den Textbrowser **ui.textBrowser** gesetzt.

Erweiterung in der Klasse Scene

```
extern Stats stats;
Detector *detR;
```

```
Detector *decQ;
char outputFileName[] = "./output/";
[...]
Scene::Scene(QGraphicsScene *pscene) : Tscene(), scene(pscene) {
[...]
  outputtimer = new QTimer();
  QObject::connect(outputtimer, SIGNAL(timeout()), this, SLOT(outPut()));
  outputtimer->start(50);

  detQ = new Detector("Quadrat", 30, 30, 5, pscene, this);
  detR = new Detector("Rechteck", -40, 25, 20, 10, pscene, this);

   buildOutputFile();
}
```

Listing 11: Erweiterung in demoapp/src/scene.cpp

- Das globale **Stats**-Objekt **stats** wird eingebunden.
- Für die Dateiausgabe wird das Char-Arry **outputFileName** erzeugt, sie soll innerhalb des Projektordners, in den Ordner output, erfolgen.
- **outputtimer** wird mit einer Intervalldauer von $50ms$ verwendet.
- Es werden beispielhaft die Detektoren **detR** und **detQ** erzeugt.
- Der Aufruf **buildOutputFile()** initialisiert die Ausgabedatei.

```
void Scene::buildOutputFile() {
       time_t now = time(0);
       tm * date = localtime(&now);
       char temp [100];
       strftime(temp,100,"%Y-%m-%d_%H:%M:%S",date);
       strcat(outputFileName, temp);

       ofstream myfile;
       myfile.open(outputFileName);
       myfile<<"Zeit;DetQ;DetR"<<endl;
       myfile.close();
}
```

Listing 12: Erstellte Methode buildOutputFile() in scene.cpp

- Es wird die momentane Zeit erfasst und in der Form `JJJJ.MM.TT\_hh:mm:ss` in der Variable **outputFileName** gespeichert.

- Die Ausgabe erfolgt im CSV-Format, die Spaltentrennung erfolgt per Semikolon. Die Spaltennamen `Zeit`, `DetQ` und `DetE` werden in die Ausgabedatei geschrieben.

```
void Scene::moveAllAgents() {
[...]
        //bei start ist Simulationsschrittgroesse = 0, wenn > 0 soll der
            ↪ Timer starten
        if ((config.simh > 0) && !(simtimer.isValid())) {
                simtimer.start();
                diffsimtimestep = timestep;
        }

        if (config.simh > 0) {
                simtimestep = timestep - diffsimtimestep;
        }

        //hier rein weil, Periodendauer benoetigt wird
        fpsdiff = fpstimer.elapsed() - fpsnow;
        realfps = (double) 1000/fpsdiff;
        fpsnow = fpstimer.elapsed();
}
```

Listing 13: Erweiterung der Methode moveAllAgents() in demoapp/src/scene.cpp

```
allAgents = getAllAgents().size();
        // fileOutput fuer die Dateiausgabe: Eine Zeile besteht aus
            ↪ folgenden Spalten Zeit, 1-Detektor-Wert....
        // guiOutput fuer die Guiausgabe: wird zusammengesetzt aus
            ↪ Simulationswerten und DetektorWerte
        string fileOutput;
        string guiOutput;

        //fileOutput wird gefuellt wenn simtime>0, also erst ab
            ↪ Simulationsstart
        if (simtime > 0) fileOutput = to_string(simtimestep).append(";");

        guiOutput.append("Flaschenhals:␣" + to_string(bottleNeck) + "\n");
        guiOutput.append("SimTimesteps:␣" + to_string(simtimestep) + "\n");
        guiOutput.append("RealFPS:␣" + to_string(realfps) + "\n");
        guiOutput.append("********\n");

        //die DetectorQueue wird einmal komplett durchgegangen, alle
            ↪ Detektoren werden so abgewickelt
```

```
for (Detector *item : stats.DetectorQueue) {

        //beliebige Detektorwerte fuer Dateiausgabe, gleichzeitig
            ↪ wird hier das aktulle item durch Detector::getDensity
            ↪ () getriggert die Dichte zu erfassen
        //fileOutput wird gefuellt wenn simtime>0, also erst ab
            ↪ Simulationsstart
        if (simtime > 0) {
                fileOutput.append(to_string(item->getDensity())).
                    ↪ append(";");
                fileOutput.append(to_string(item->numberOfAgents)).
                    ↪ append(";");

        }

        //Detektorwerte fuer Gui-Ausgabe

        guiOutput.append(item->name + ":␣");
        guiOutput.append(to_string(item->density) + "\n");

}

//Dateiausgabe wenn simtime>0, also erst ab Durchfuehrungsstart
if (simtime > 0) {
        fileOutput.append(to_string(allAgents) + ";");
        fileOutput.append(to_string(bottleNeck) + ";");
        fileOutput.append(to_string(config.simh) + ";");
        //Die Zeile wird wird \n abgeschlossen
        fileOutput.append("\n");

        //Die Ausgabe fileOutput wird in die Datei geschrieben
        ofstream myfile;
        myfile.open(outputFileName, ios::app);
        myfile<<fileOutput;
        myfile.close();
}

//guiOutput wird an Stats gepackt, das triggert die Ausgabe
stats.inputStringToUi(guiOutput);
```

Listing 14: Final erstellte Methode outPut() in demoapp/src/scene.cpp

- Der String **fileOutput** ist für die Ausgabedatei. In ihn wird die Zeit geschrieben.
- Der String **guiOutput** ist für die GUI-Ausgaben. In ihn werden mehrere Simulati-

onswerte geschrieben,

- Die Queue **stats.DetectorQueue**, die alle vorhandenen Detektoren enthält, wird mit **iterator** durchgegegangen. Dabei werden in **fileOutput** und **guiOutput** die jeweils benötigten Werte geschrieben, im Code beispielhaft die Dichte und die Detektornamen.
- Mit **myfile«fileOutput** wird der erzeugt Inhalt in die Ausgabedatei geschrieben.
- Die Methode **stats.inputStringToUi(guiOutput)** wird aufgerufen. Dadurch wird die GUI-Ausgabe ausgelöst.

Ergänzung zu Abschnitt 5.3: Koordinaten der Messflächen

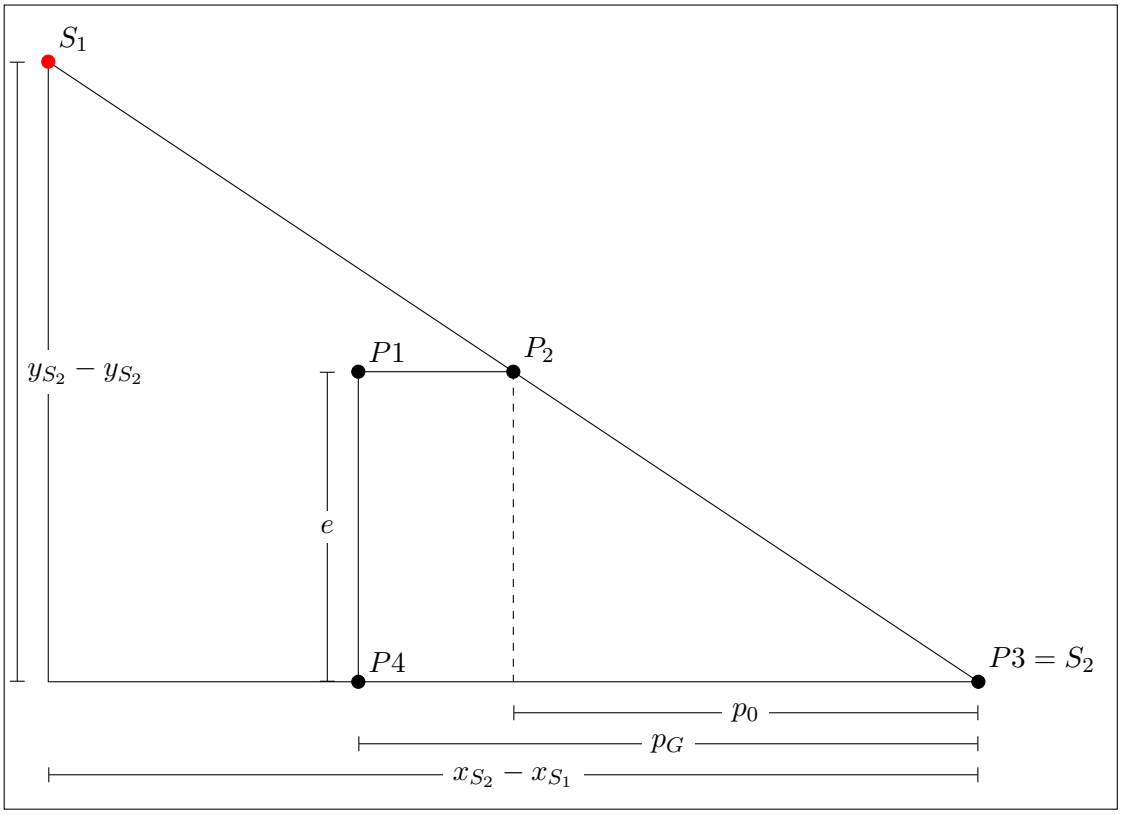

Abbildung 19: Bildung polygoner Messfläche per Strahlensatz

Zusätzlich zu S_1 und S_2 sind p_G und e gegeben. Für ein Polygon, das mit einer Schrägen genau an der abfallenden Schräge des Hindernis liegt, ergeben sich die Punkte P_1, P_3 und P_4. Für die Positionierung von P_2 werden die Koordinaten $x_{P2} = x_{S_3} - p_0$ und $y_{P_2} = x_{S_3} - e$ gebildet. Per Strahlensatz wird mit $p_0 = e * \frac{x_{S_2} - x_{S_1}}{y_{S_2} - y_{S_1}}$ berechnet. Der Flächeninhalt beträgt $A = e * p_0 * 0,5 + e * p_G$. Das Polygon kann jeweils horizontal und vertikal gespiegelt werden, sodass sich die Koordinaten für weitere Polygone ergeben.

Eine Übersicht der Koordinaten der relevanten Messflächen bietet die nachfolgende Tabelle. Enthalten sind:

- Name
- Koordinaten der Polygonpunkte zur Polygon-Objektübergabe
- Werte für den Konstruktoraufruf:
  - xy
  - y
  - w
  - h
- Form

Name	**Punkt**	**x**	**y**	**w**	**h**	**Form**
PolyLeftTop						
Aufruf	P_{LT1}	$x_{S_2} - p_G$	$y_{S_2} - e$	p_G	e	
	P_{LT2}	$x_{S_2} - p_0$	$y_{S_2} - e$			
	P_{LT3}	x_{S_2}	y_{S_2}			
	P_{LT4}	$x_{S_2} - p_G$	y_{S_2}			
OuterEntry						
Aufruf 0	P_{LT4}	$x_{S_2} - p_G$	y_{S_2}	p_G	f	
Aufruf 1		$x_{S_2} - p_G$	y_{S_2}	$p_G * 0,5$	$f * 0,5$	
Aufruf 2		$x_{S_2} - p_G * 0,5$	y_{S_2}	$p_G * 0,5$	$f * 0,5$	
Aufruf 3		$x_{S_2} - p_G$	$y_{S_2} + f * 0,5$	$p_G * 0,5$	$f * 0,5$	
Aufruf 4		$x_{S_2} - p_G * 0,5$	$y_{S_2} + f * 0,5$	$p_G * 0,5$	$f * 0,5$	
PolyLeftBottom						
Aufruf	P_{LB1}	$x_{S'_2} - p_G$	$y_{S'_2}$	p_G	e	
	P_{LB2}	$x_{S'_2}$	$y_{S'_2}$			
	P_{LB3}	$x_{S'_2} - p_0$	$y_{S'_2} + e$			
	P_{LB4}	$x_{S'_2} - p_G$	$y_{S'_2} + e$			
Bottle						
Aufruf		x_{S_2}	y_{S_2}	$x_{S_3} - x_{S_2}$	f	
InnerEntry		x_{S_2}	y_{S_2}	30	f	
Mid		-15	y_{S_2}	30	f	
InnerExit		$x_{S_2} - 30$	y_{S_2}	30	f	
OuterExit0						
Aufruf		x_{S_3}	y_{S_3}	30	f	

Tabelle 4: Übersicht der Messflächen

Ergänzung zu Abschnitt 6.1: Generierte Dateien für DF50

Generierte Konstruktoraufrufe

```
detPolyLeftTop = new Detector("PolyLeftTop",orange,-150,-50,30,25,pscene,
    ↪ this);
polyLeftTop << QPointF(-150,-50) << QPointF(-129.259259259259,-50) <<
    ↪ QPointF(-120,-25) << QPointF(-150,-25) << QPointF(-150,-50);
detPolyLeftTop->isPolygon = true;
detPolyLeftTop->polygon = polyLeftTop;
detPolyLeftTop->area =1000;

detOuterEntry0 = new Detector("OuterEntry0",red,-150,-25,30,50,pscene,this)
    ↪ ;
detOuterEntry1 = new Detector("OuterEntry1",red,-150,-25,15,25,pscene,this)
    ↪ ;
detOuterEntry2 = new Detector("OuterEntry2",red,-135,-25,15,25,pscene,this)
    ↪ ;
detOuterEntry3 = new Detector("OuterEntry3",red,-150,0,15,25,pscene,this);
detOuterEntry4 = new Detector("OuterEntry4",red,-135,0,15,25,pscene,this);

detPolyLeftBottom = new Detector("PolyLeftBottom",firebrick,-150,25,30,25,
    ↪ pscene,this);
polyLeftBottom << QPointF(-150,50) << QPointF(-129.259259259259,50) <<
    ↪ QPointF(-120,25) << QPointF(-150,25) << QPointF(-150,50);
detPolyLeftBottom->isPolygon = true;
detPolyLeftBottom->polygon = polyLeftBottom;
detPolyLeftBottom->area =1000;

detBottle = new Detector("Bottle",blue,-120,-25,240,50,pscene,this);
detInnerEntry = new Detector("InnerEntry",purple,-120,-25,30,50,pscene,this
    ↪ );
detMid = new Detector("Mid",deepskyblue,-15,-25,30,50,pscene,this);
detInnerExit = new Detector("InnerExit",limegreen,90,-25,30,50,pscene,this)
    ↪ ;

detPolyRightTop = new Detector("PolyRightTop",greenyellow,95,-50,30,25,
    ↪ pscene,this);
polyRightTop << QPointF(150,-50) << QPointF(129.259259259259,-50) <<
    ↪ QPointF(120,-25) << QPointF(150,-25) << QPointF(150,-50);
detPolyRightTop->isPolygon = true;
detPolyRightTop->polygon = polyRightTop;
detPolyRightTop->area =1000;
```

```
detOuterExit0 = new Detector("OuterExit0",yellow,120,-25,30,50,pscene,this)
    ↪ ;
detOuterExit1 = new Detector("OuterExit1",yellow,135,-25,15,25,pscene,this)
    ↪ ;
detOuterExit2 = new Detector("OuterExit2",yellow,120,-25,15,25,pscene,this)
    ↪ ;
detOuterExit3 = new Detector("OuterExit3",yellow,135,0,15,25,pscene,this);
detOuterExit4 = new Detector("OuterExit4",yellow,120,0,15,25,pscene,this);

detPolyRightBottom = new Detector("PolyRightBottom",khaki,120,25,30,25,
    ↪ pscene,this);
polyRightBottom << QPointF(150,50) << QPointF(129.259259259259,50) <<
    ↪ QPointF(120,25) << QPointF(150,25) << QPointF(150,50);
detPolyRightBottom->isPolygon = true;
detPolyRightBottom->polygon = polyRightBottom;
detPolyRightBottom->area =1000;

detGoal = new Detector("Goal",white,150,-160,50,320,pscene,this);
detGoal->isVisible = false;

for (Detector *item : stats.DetectorQueue) if (item->isVisible) item->draw
    ↪ ();
```

Listing 15: Generierte Konstruktoraufrufe in demoapp/src/scene.cpp

Generierte XML-Datei

```
<scenario>
        <!-- Flaschenhals oben-links nach oben-rechts -->
        <obstacle x1='-170' y1='-160' x2='-120' y2 ='-25' />
        <obstacle x1='-120' y1='-25' x2='120' y2 ='-25' />
        <obstacle x1='120' y1='-25' x2='170' y2 ='-160' />
        <obstacle x1='170' y1='-160' x2='200' y2 ='-160' />
        <obstacle x1='170' y1='-160' x2='200' y2 ='-160' />

        <!-- Flaschenhals unten-links nach unten-rechts -->
        <obstacle x1='-170' y1='160' x2='-120' y2 ='25' />
        <obstacle x1='-120' y1='25' x2='120' y2 ='25' />
        <obstacle x1='120' y1='25' x2='170' y2 ='160' />
        <obstacle x1='170' y1='160' x2='200' y2 ='160' />

        <!--Wegpunkt W1 im Flaschenhalseingang-->
        <waypoint id='w1' x='-120' y='0' r='25' />

        <!-- Simulationsbegrenzung-->
        <obstacle x1='-200' y1='-160' x2='200' y2 ='-160' />
        <obstacle x1='200' y1='-160' x2='200' y2 ='160' />
        <obstacle x1='-200' y1='160' x2='200' y2 ='160' />
        <obstacle x1='-200' y1='160' x2='-200' y2 ='-160' />

        <!-- Wegpunkt W2-->
        <waypoint id='w2' x='100200' y='0' r='100000' />

        <!-- Agenten-->
        <agent x='-185' y='0' n='1000' dx='30' dy='320' >
                <addwaypoint id='w1' />
                <addwaypoint id='w2' />
        </agent >
</scenario>
```

Listing 16: Generierte XML-Datei