Erweiterung der Crowd Simulation PEDSIM mit einer graphischen Darstellung der Dichte von Agenten

Praxisprojektdokumentation
im Studiengang Medieninformatik
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Tim Kurtz
Matrikel-Nr.:	011 089 2076
Adresse:	tim.kurtz@smail.th-koeln.de

eingereicht bei:	Prof. Dr. Wolfgang Konen

21.11.2020

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

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Einleitung

Fast genau 10 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 Veranstaltung sicherer zu gestalten, wird vor der Stattfindung mit Simulationsprogrammen gearbeitet.

In dieser Arbeit werden die grundlegenden Funktionen von der Massensimulationsprogrammbibliothek PEDSIM beschrieben und das Demoprogramm, um eine graphische Darstellung der Agentendichte, auf Basis der Quadtreetiefe, erweitert.

In Abschnitt 1 wird der Begriff Crowd Desaster (1.1) und die Ursache (1.2) für ein solches erklärt. Desweiteren werden drei verschiedene Simulationen verglichen (1.3.2).

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. Zudem wird die Oberfläche des Demoprogrammes beschrieben (2.5) und auf Erfahrungen (2.6) damit eingegangen.

In Abschnitt 3 wird beschrieben wie PEDSIM mit einer graphischen Dichtendarstellung erweitert wird. Dazu wird zuerst genauer auf die Funktionalität (3.2) und die graphische Darstellung (3.3) der Quadtrees eingegangen. Im weiteren Verlauf wird der Programmcode erweitert, indem die Quadtrees in Relation zur Tiefe eingefärbt werden (3.4). Anschließenden werden die Beobachtungen mit der Darstellung beschrieben (3.4.5).

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 [10]:

- 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 aneinandergedrängt.
- In den seltesten 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 [10].

1.2 Ursache des Crowd Desaster

Helbing et al. und der Gerichtssachverständige Jürgen Gerlach stellen verschiedene Ursachen des Crowd Desaster bei der Loveparade heraus. Ein wichtiger Aspekt sei, dass das Gelände aufgrund der physikalischen Gegebenheiten nicht für die erwartbare Anzahl an Besuchern geeignet war und so tödliche Drucksituationen entstanden [10][2]. Die Loveparade habe bei einer genaueren Ortsinspektion und Planung nicht stattfinden dürfen [2].

1.3 Simulationen

Für die Planung von Großveranstaltungen wird Software genutzt, die das Verhalten und die Bewegung von Menschenmassen an dem geplanten Verstanstaltungsort simuliert. Die Vergleichsarbeit von Richards bietet einen Überblick von 34 verschiedenen Programmen [14]. Da die Zeit fehlte alle auszutesten, wird im folgenden kurz auf 2 Programme und die Bachelorarbeit von Hallweger eingegangen.

1.3.1 Anforderungen

- einsehbaren, dokumentierten und erweiterbaren Quellcode
- lauffähig auf einem durchschnittlichen Computer mit Betriebssystem Linux
- agentenbasierte Gruppensimulation
- einsehbares Simulationsmodell

1.3.2 Vergleich der Simulationen

Programm	*PEDSIM [5]*	*Vadere [1]*	*Hallweger [8]*
Quellcode	einsehbar, dokumentiert	einsehbar	-
agentenbasiert	ja	ja	ja
Simulationsmodell	Social Force Model	Social Force Model, Optimal Step Model	Unity
lauffähig	ja	teilweise	-
Sprache	C++	Java	C#

Tabelle 1: Vergleich Simulationen

1.3.3 PEDSIM

PEDSIM ist eine Programmbibliothek mit mehreren Beispielprogrammen. Sie hat eine umfassende Dokumentation, dies wird durch die Projekt-Website `http://pedsim.silmaril.org/` erweitert. PEDSIM ist ein Softwaremodul der Arbeit „Distributed intelligence in real world mobility simulations“ [7] und wird momentan nicht weiterentwickelt.

1.3.4 Vadere

Vadere ist ein umfangreiches Softwareprojekt der Hochschule München und ist in der laufenden Entwicklung. Es bietet über 30 Simulationsszenarien, einen Szenarienbaukasten und mindestens zwei Simulationsmodelle. Allerdings ist die Simulation per „Social Force Model“ sehr langsam, die Agenten machen bloß alle drei bis vier Sekunden einen Schritt.

1.3.5 Hallweger Bachelorarbeit

Die Bachelorarbeit von Hallweger dokumentiert ein Projekt auf Basis von Unity. Das Simulationsmodel ist von Unity gestellt und nicht einsehbar. Hallwegers Schwerpunkt liegt auf einer ansprechenden visuellen Darstellung, anstatt auf einer realistischen Simulation.

1.3.6 Fazit

Es wurde sich für PEDSIM statt Vadere entschieden, da der Code umfassender dokumentiert ist und eine umfangreichere Programmdokumentation beiliegt. Außerdem traten hier keine Performanceprobleme auf.

2 Beschreibung von PEDSIM

2.1 Einleitung

Nachdem sich in Kapitel 1 für PEDSIM entschieden wurde, wird im folgenden Kapitel auf tiefergehende Beobachtungen und Eigenschaften des Programmes 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 Grundlegenden Klassen PEDSIM

Die folgenden Klassen stellen die funktionale Grundlage zur Simulation dar [5]:

- **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. Ihnen 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 [5]:

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 [5]:

- **<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 rechts um das Quadrat, die rechte Gruppe links um das Quadrat. Da beide Gruppen die selben Wegpunkte ablaufen, müssen sie sich gegenseitig ausweichen.

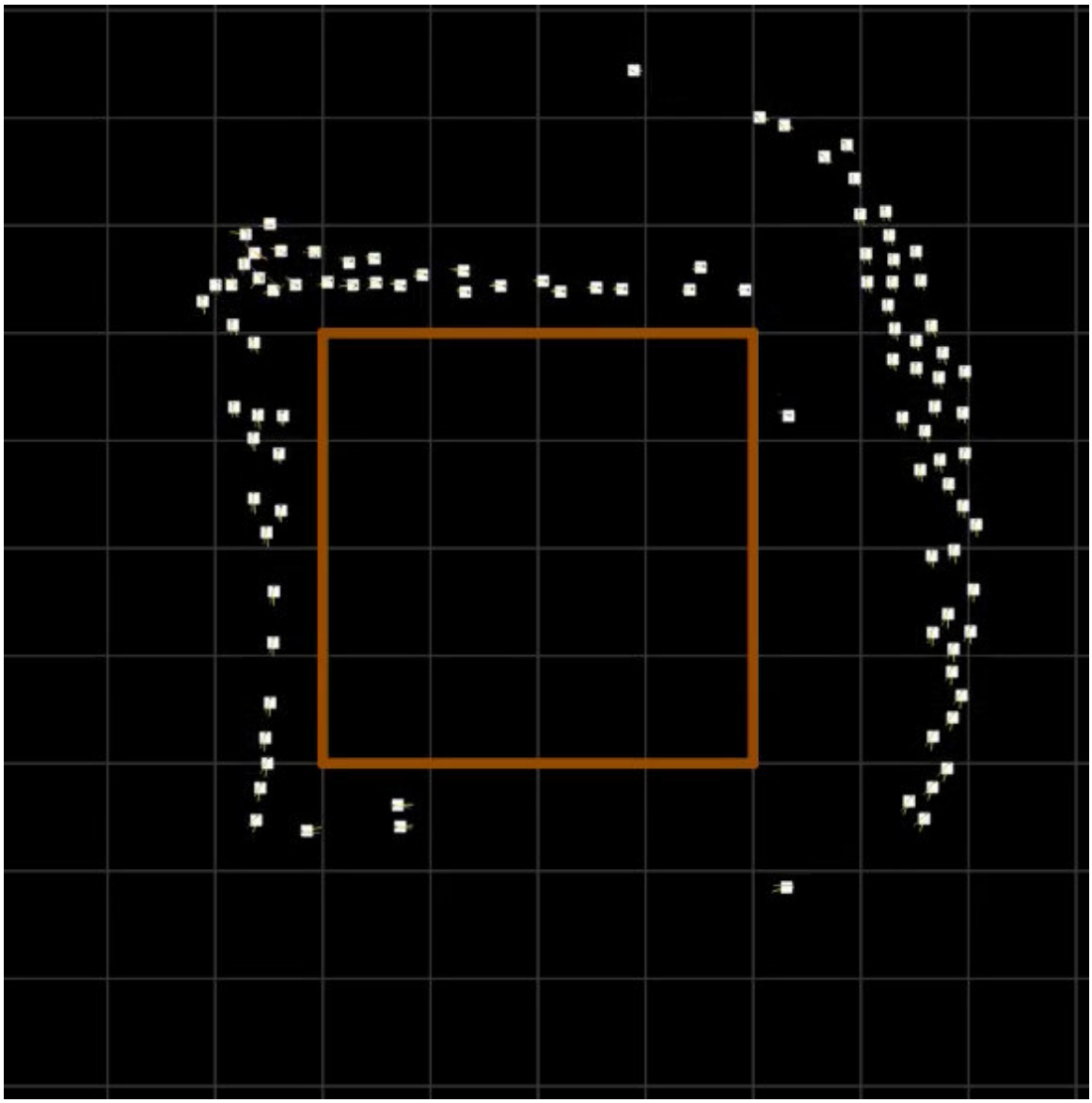

Abbildung 1: Screenshot aus 01-Beispielszenario.mp4

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 [9]:

1. Die gewünschte Geschwindigkeitsänderung pro Zeit, 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 [5]:

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

Mit folgenden Methoden wird das „Social Force Model“ in der Klasse **Ped::Tagent** umgesetzt [5]:

- **Ped::Tvector Ped::Tagent::desiredForce ()**: Realisiert die gewünschte Geschwindigkeitsänderung. Das Ziel eines Agenten ist immer, sich dem nächsten Wegpunkt anzunähern.
- **Ped::Tvector Ped::Tagent::socialForce()**: Realisiert die abstoßende Kraft von Agenten untereinander. Sie ist als unmittelbare Kollisionsvermeidung angelegt. Dafür wird über alle Agenten in der Szene iteriert und jeweils für den am nächsten liegenden Agenten die Kraft berechnet.
- **Ped::Tvector Ped::Tagent::obstacleForce ()**: Realisiert die abstoßende Kraft zwischen Agenten und Hindernissen und ist damit ähnlich wie **Ped::Tagent::socialForce()** als unmittelbare Kollisionsvermeidung angelegt. Sie iteriert über alle Hindernisse und berechnet die abstoßende Kraft für das am nächsten gelegene Hindernis.

2.4.3 Quadtree

Die Agenten werden in einer dynamischen Quadtreestruktur gespeichert [5]. Sie unterteilt die Welt rekursiv in vier rechteckige Quadranten. Die Agenten bewegen sich in der 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 [4]. 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 Baumsabschnitte iteriert, anstatt über über alle vorhandenen Agenten.

2.5 Graphische Oberfläche des Demoprogrammes

Die graphische Oberfläche des Demoprogrammes enthält auf der linken Seite eine Steuerleiste und auf der rechten Seite die graphische Darstellung der Simulation in 2D.

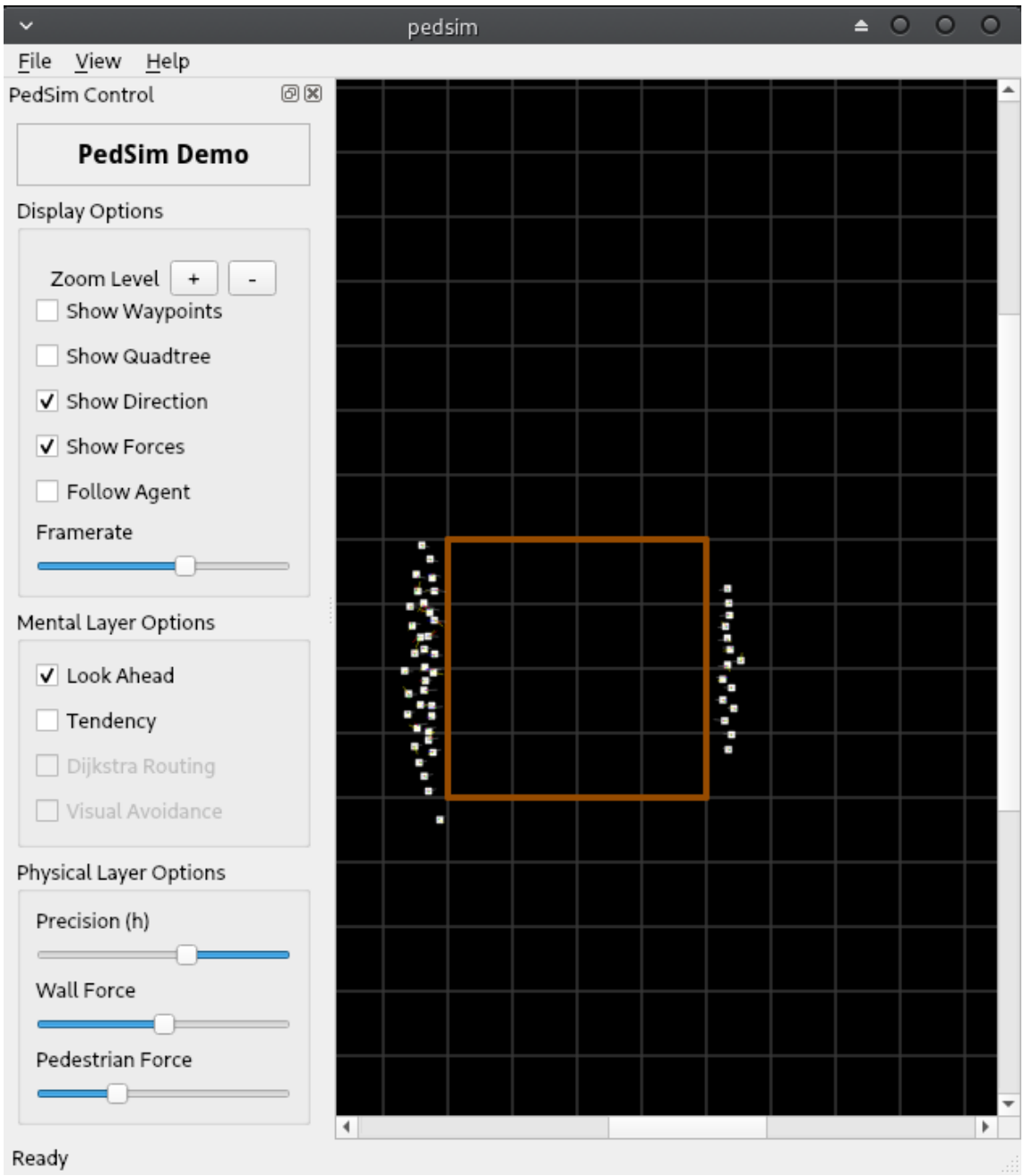

Abbildung 2: Graphische Oberfläche Demoprogramm

2.5.1 Steuerleiste

Displayoptions

Zoom Level Per Plus-Button und Minus-Button kann die Simulationsdarstellung verlustfrei skaliert werden. Die Skalierung geht von unendlich klein bis unendlich groß.

Show Waypoints Per Checkbox sollte die Sichtbarkeit der Wegpunkte ausgewählt werden. Dies scheint nicht zu funktioneren, die Wegpunkte waren zu keinem Zeitpunkt sichtbar. Da die Wegpunkte auch für den weiteren Verlauf der Arbeit keine besondere Relevanz haben, wird dieser Fehler ignoriert.

Show Quadtree Per Checkbox kann die Sichtbarkeit des Quadtrees ausgewählt werden. Sie ist durch den geringen Kontrast der dunkelroten Linien zu dem schwarzen Hintergrund zwar niedrig, trotzdem lässt sich die Dynamik der Speicherung erkennen. Aus diesem Grund wurde die Farbe später in Abschnitt 3.4.1 in grün geändert.

Show Direction Per Checkbox kann die Sichtbarkeit des gelben Pfeiles ausgewählt werden, der die Bewegungsrichtung des Agentens anzeigt. Sie wird im Abschnitt 2.4.2 durch **desiredForce()** beschrieben.

Show Forces Per Checkbox kann die Sichtbarkeit der Pfeile ausgewählt werden, die die auf den Agenten wirkenden Kräfte (Abschnitt 2.4.2) anzeigen.

- blau: Wall Force, in 2.4.2 durch **obstacleForce()** beschrieben.
- rot: Pedestrian Force, in 2.4.2 durch **socialForce()** beschrieben.
- Grau: Look Ahead Force, laut Dokumentation sollte sie magenta sein [5].

Follow Agent Per Checkbox kann ausgewählt werden, ob der Bildausschnitt der Simulation einem zufälligen Agenten folgt oder starr ist.

Framerate Per Schieberegler kann die Framerate gesetzt werden, der Wert wird hierbei nicht angezeigt. PEDSIM reguliert zudem die Framerate, sodass das Programm nicht das System überlastet [5]. Im Quellcode **demoappp/scr/config.cpp** lässt sich ablesen, das der Default-Wert 30 FPS beträgt.

Mental Layer Options Dieser Bereich enthält die nicht funktionalen Checkboxen für „Dijkstra Routing“ und „Visual Avoidance“. Diese Funktionen sind nicht implementiert [5].

Look Ahead Per Checkbox kann die Funktion „Look Ahead“ ausgewählt werden. Für jeden Agenten werden die entgegenkommenden Agenten gezählt, um auf die Seite auszuweichen, wo weniger Agenten sind [5].

Tendency Per Checkbox kann die Funktion „Tendency“ ausgewählt werden. Dadurch weichen die Agenten untereinander früher aus und verringern ihre Geschwindigkeit. Leider wird diese Funktion in der PEDSIM-Dokumentation nicht erklärt.

2.5.2 Physical Layer Options

Precision Per Schieberegler kann die „Precision (h)“ von 1 bis 0, in Zwischenschritten von 0,01, gesetzt werden. „h“ gibt die zeitliche Länge eines Simulationsschrittes an. Je mehr „h“ gegen 0 geht, desto kleiner wird ein Simulationsschritt und desto langsamer bewegen sich die Agenten. Je mehr „h“ gegen 1 geht, desto größer wird ein Simulationsschritt und desto schneller bewegen sich die Agenten. Zudem bedeuten kleinere Simulationsschritte eine höhere Genauigkeit der Simulationsberechnung [5].

Wall Force Per Schieberegler kann die „Wall Force“ von 0 bis 100, in Zwischenschritten von 0,1, gesetzt werden. Sie wird in 2.4.2 durch **obstacleForce()** beschrieben.

Pedestrian Force Per Schieberegler kann die „Pedestrian Force“ von 0 bis 100, mit Zwischenschritten von 0,1, gesetzt werden. Sie wird in 2.4.2 durch **socialForce()** beschrieben.

2.6 Beobachtung

Das Demoprogramm wurde mit dem enthaltenen Szenario **scene.xml** und sichtbarem Quadtree ausgeführt. Die Agentenanzahl wurde von 200 auf 400 verdoppelt. Nun lassen sich folgende Beobachtungen machen:

- Die Darstellung der Simulations wird stockender.
- Es enstehen mehrere Stellen mit augenscheinlich hoher Agentendichte, vor allem an Engstellen wie Flaschenhälse.
- Hierbei entstehen sehr kleine Quadtrees, diese fassen maximal vier Agenten.

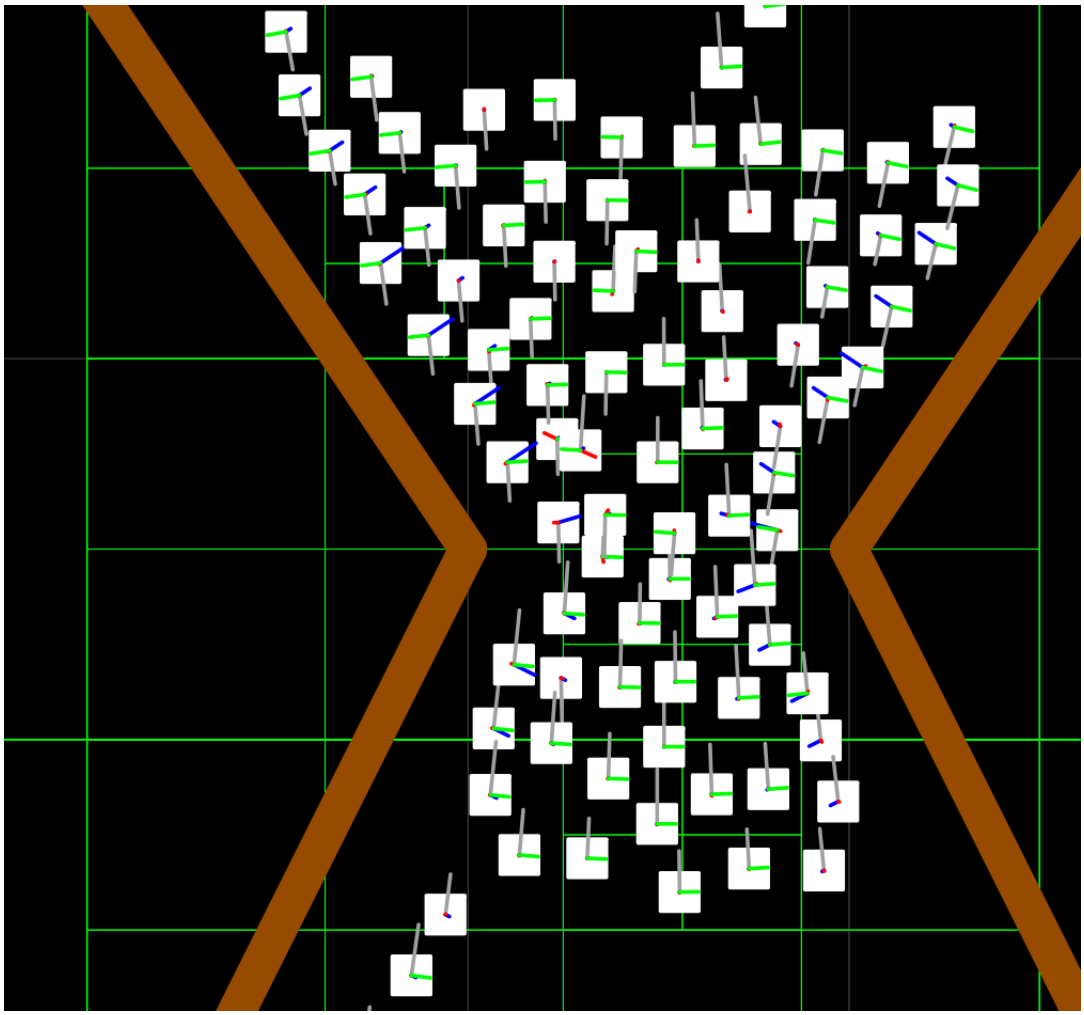

Abbildung 3: Hohe Agentendichte bei Flaschenhals

3 Erweiterungen von PEDSIM mit graphischer Dichtendarstellung

3.1 Einleitung

In Abschnitt 1.2 wurde deutlich, das eine zu hohe Personendichte zu dem Crowd Desaster der Loveparade geführt hat. Weil PEDSIM aber keine graphische Darstellung der Agentendichte bereitstellt, bot sich an, diese zu entwickeln. Dafür müssten punktuell Agenten gezählt und in Relation zur Fläche gesetzt werden. Da in der Dokumentation keine Angaben zum Simulationsmaßstab gemacht werden, wird vorerst nach dem Sichtbaren gegangen. Hierbei geht es um eine grundsätzliche Machbarkeit und nicht um die genauen Werte.

Wie bereits in Abschnitt 2.4.3 beschrieben, werden die Agenten in Quadtrees gespeichert, wobei ein Quadrant maximal acht Agenten enthält. In Abschnitt 2.6 wurde die Beobachtung beschrieben, dass bei augenscheinlich hoher Dichte der kleinste gesichtete Quadrant maximal vier Agenten fasst. Hier wird vermutlich schon eine zu hohe Agentendichte erreicht. Ein erster Ansatz ist deshalb, die Agentendichte in Relation zur Tiefe des Quadtrees zu setzen.

3.2 Tiefe des Quadtrees

Der Quadtree ist ein Objekt der **Ped::Ttree** - Klasse [3], die graphische Darstellung ist ein Objekt der davon abgeleiteten Klasse **Ped::Ttree:Tree** [6]. Beide enthalten jeweils die geschützte Klassenvariable **depth**. Es kann angenommen werden, dass sie die Tiefe des jeweiligen Kindsknoten beschreibt. Da sie aber in der Dokumentation nicht weiter beschrieben wird, wurde der Quellcode untersucht.

Innerhalb der Klasse **Ped::Ttree** wird die Variable **depth** in einer getter-Funktion und in der Methode **addChildren()** verwendet [4] . In dieser Methode werden vier neue Kindesbäume generiert und an den entsprechenden Knoten gesetzt [3]. Dabei wird **depth** für jeden Baum um 1 inkrementiert. Gleiches gilt in der Klasse **Ped::Ttree:Tree** für die namensgleiche Methode **addChildren()**.

Nun wurden per Ausgabe der Variable **depth** und gleichzeitiger Betrachtung der graphischen Darstellung des laufenden Demoprogrammes folgende Beobachtungen gemacht:

- Der Startwert beträgt 0, also wird der Standartwert von **depth** entsprechend 0 sein.
- Während den ersten Sekunden nach Start des Programmes beträgt der Wert von **depth** 8. Zu diesem Zeitpunkt sind die gerade generierten Agenten auf einer sehr kleinen Fläche eng beieinander und es sind sehr kleine Quadtrees sichtbar.
- Im weiteren Verlauf hat **depth** Werte von 1 bis 7. Die Quadtrees erweitern und verringern sich währenddessen.

Es ist also davon auszugehen, dass die Variable **depth** die Tiefe eines Quadtrees repräsentiert. Die kleinsten Quadranten, die maximal vier Agenten fassen, werden vermutlich eine Tiefe von 8 haben.

3.3 Grahische Darstellung der Quadtrees

Zur Darstellung der Dichte könnten die Quadranten bzw. Rechtecke des Quadtrees entsprechend eingefärbt werden. Dafür wurde zunächst ein grundlegendes Verständnis der Klasse **Ped::Ttree:Tree** erarbeitet, welche für die graphische Darstellung der Rechtecke verantwortlich ist.

3.3.1 Methode Ped::Ttree:Tree::Tree()

```
Tree::Tree(QGraphicsScene *pgraphicsscene, Scene *pedscene, int pdepth,
    ↪ double px, double py, double pw, double ph) : Ped::Ttree(pedscene,
    ↪ pdepth, px, py, pw, ph) {
        graphicsscene = pgraphicsscene;
        scene = pedscene;
        QPen p = QPen(QColor(88,0,0), 1, Qt::SolidLine, Qt::RoundCap, Qt::
            ↪ RoundJoin);
        p.setCosmetic(true);
        rect = graphicsscene->addRect(getx(), gety(), getw(), geth(), p);
        rect->setVisible(config.showTree);
        rect->setZValue(-100+getdepth())
};
```

Listing 2: Methode Tree() aus demoapp/src/tree.cpp

3.3.2 Erklärung

Initialiserung Bei der Intitialisierung werden folgende Parameter übergeben:

- ***pgraphicsscene**, ***pedscene**: Zeiger auf die graphische und die interne Szene.
- int **pdepth**: Baumtiefe
- **px**, **py**: Die Koordinaten der oberen linken Ecke des Rechteckes. Hier ist der Ausgangspunkt des Rechteckes.
- **pw**, **ph**: Die Breite (w: width) und die Höhe (h: height) des Rechteckes.

QPen p = QPen([...]) Mit Objekten der Klasse **QPen** wird die Linienzeichung definiert [12]. Dieser Aufruf erfolgt mit folgenden Parametern:

- **QColor(88,0,0)**: Die Farbe der Linie beträgt dunkelrot.
- 1: Die Breite der Linie beträgt 1.
- Qt::SolidLine: Die Linie ist durchgehnd.
- Qt::RoundCap: Die Endpunkte der Linie sind abgerundet.
- Qt::RoundJoin: Die Verbindung von zwei Linien wird abgerundet.

p.setCosmetic(true); Die Linienzeichnung ist immer gleich breit [12].

rect = graphicsscene->addRect(getx(), gety(), getw(), geth(), p); Das graphische Rechteck wird der graphischen Szene hinzugefügt. Der Aufruf erfolgt mit folgenden Parametern:

- **getx()**, **gety()**, **getw()**, **geth()**: Die Getter-Funktionen der entsprechenden Variablen aus der Initialisierung.
- p: Übergabe des erstellten **QPen** Objekt **p**.

rect->setVisible(config.showTree); Es wird eingestellt, ob das graphische Rechteck, alle zusammen ergeben den Quadtree, sichtbar ist. Der Wert der Variable **config.showTree** wird über die Checkbox „Show Quadtree“ in der Oberfläche eingestellt.

rect->setZValue(-100+getdepth()) Anhand des ZValue werden graphische Elemente hierarisch geschichtet. Höhere Elemente übermalen niedrigere Elemente [**?**]. Durch den Wert -100 sind die Rechtecklinien immer unter den Agenten, durch **getdepth()** werden sie von ihrer entsprechenden Baumtiefe abhängig gemacht.

3.4 Erweiterung um die graphische Darstellung der Dichte

Nachdem die Methode Ped::Ttree:Tree() verstanden wurde, stellte sich nun die Frage, ob sich die Rechtecke des Quadtrees grundsätzlich einfärben lassen. Ein Blick in die Dokumentation von Qt zeigt, dass bei dem Aufruf **addRect()** zusätzlich ein **QBrush**-Objekt übergeben werden kann [13]. Mit der **QBrush**-Klasse werden Füllungen definiert, möglich sind verschiedene Füllstile und Farben [11].

3.4.1 Farbe des Quadtrees

Wie schon in Abschnitt 2.5.1 erwähnt, ist der Kontrast zwischen den dunkelroten Linien und dem Hintergrund zu niedrig. Deshalb wurde die Farbe im **QPen p** auf ein reines grün gesetzt:

```
[QPen p = QPen(QColor(0,255,0), 1, Qt::SolidLine, Qt::RoundCap, Qt::
    ↪ RoundJoin)];
```

Listing 3: Änderung der p-Farbe in demoapp/src/tree.cpp

3.4.2 Füllung

Zuerst wurde getestet, ob eine grundsetzliche Füllung der Rechtecke möglich ist. Dafür wird das Objekt **brush** der Klasse **QBrush** mit der Farbe rot erstellt und an **addRect()** übergeben.

```
QBrush brush = QBrush(QColor(88,88,88), Qt::Dense5Pattern);
[...]
rect = graphicsscene->addRect(getx(), gety(), getw(), geth(), p, brush);
```

Listing 4: Test der Rechteckfüllung in demoapp/src/tree.cpp

Das Resultat ist die Einfärbung aller Rechtecke. Die Färbung ist zudem hinter den Agenten.

3.4.3 Füllung nach Tiefe

Der nächste Schritt ist die Farbe der Füllung von der Tiefe, also der Variable **depth**, abhängig zu machen.

1. Ansatz Hierfür werden drei Farben in Anlehnung zu einer Ampel entnommen. Die Festlegung der Werte erfolgt aus den beschriebenen Beobachtungen aus Abschnitt 3.2, wobei 6, 7 und 8 die höchsten beobachteten Werte waren. Mittels eines switch-Befehles werden die Farben nach den entsprechenden Werten gesetzt.

depth	**Farbe**	**Agentendichte**
6	grün	erhöht, aber es besteht kein Risiko
7	gelb	deutlich erhöht
8	rot	hoch.

Tabelle 2: Agentdichte einfärben 1. Ansatz

Beobachtung Die Quadranten, welche max. vier Agenten fassen, waren nicht rot, sondern gelb gefärbt. Diese Quadranten haben also einen **depth**-Wert von 7. Rot waren Quadranten zum Beginn der Simulation, hier gibt es Quadtrees mit einem **depth**-Wert von 8. Dies ist ein Artefakt, da der Bereich für die Anzahl der generierten Agenten zu klein ist und sie im Entstehungsmoment übereinander gesetzt werden.

2. Ansatz Die Werte werden den Beobachtung entsprechend angepasst.

depth	**Farbe**	**Agentendichte**
5	grün	erhöht, aber es besteht kein Risiko
6	gelb	deutlich erhöht
7	rot	zu hoch.

Tabelle 3: Agentdichte einfärben 2. Ansatz

Der finale switch-Befehl lautet:

```
switch(depth) {
               case 5:
                      b = QBrush(Qt::darkGreen);
                      break;
               case 6:
                      b = QBrush(Qt::darkYellow);
                      break;
               case 7:
                      b = QBrush(Qt::darkRed);
       }
```

Listing 5: Ergänzender switch-Befehl in der Methode Tree() aus demoapp/src/tree.cpp

3.4.4 Trennungsversuch von Quadtree und Füllung

Bisher ist die Darstellung der Dichte komplett über die Steuerung „Showtree“ abhängig. Es wurde ein Versuch unternommen, diese zu trennen. Die Bearbeitung der Oberflächensteuerung war möglich, jedoch war die Trennung in der Klasse **Tree** auch nach mehreren Versuchen nicht zu erreichen. Schließlich wurde der Versuch aufgrund der knappen Zeit nicht weiter verfolgt.

3.4.5 Ergebnis

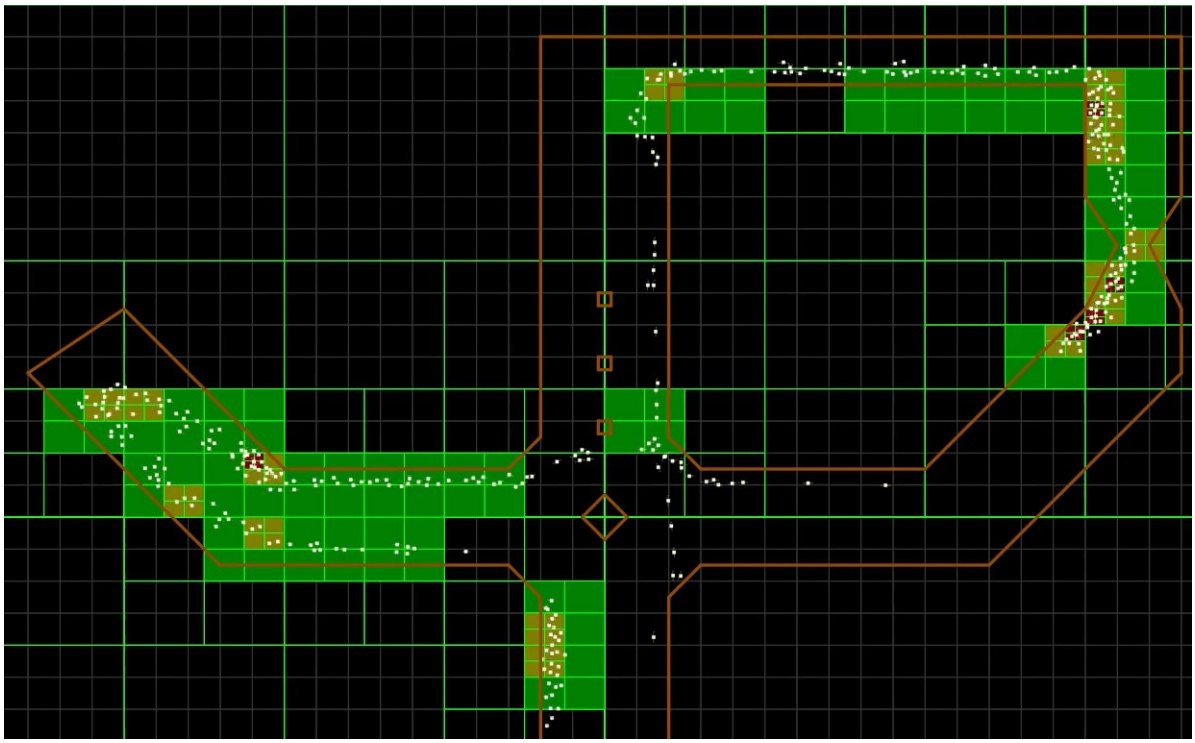

Abbildung 4: Screenshot aus 02-DichteEinfärbung.mp4

- Sobald ein einzelner Quadrant Agenten enthält, und eine Baumtiefe von mindesten 5 erreicht, werden alle vier Quadranten eingefärbt.
- An Engstellen wie Flaschenhälse wird die hohe Agentdichte, gut sichtbar, durch die rote Färbung dargestellt.
- Die Steigerung der Agentendichte lässt sich über die Farben, von grün, nach gelb und schließlich zu rot, verfolgen.
- Auch an weiteren Stellen wird überraschenderweise eine hohe Agentendichte sichtbar. Diese entstehen durch einen zu geringen Passierradius, die Wegpunkte die abgegangen werden müssen und die bloße unmittelbare Kollisionsvermeidung. Agenten können deshalb nicht vorrausschauend einer entstehenden Ansammlung von anderen Agenten ausweichen.

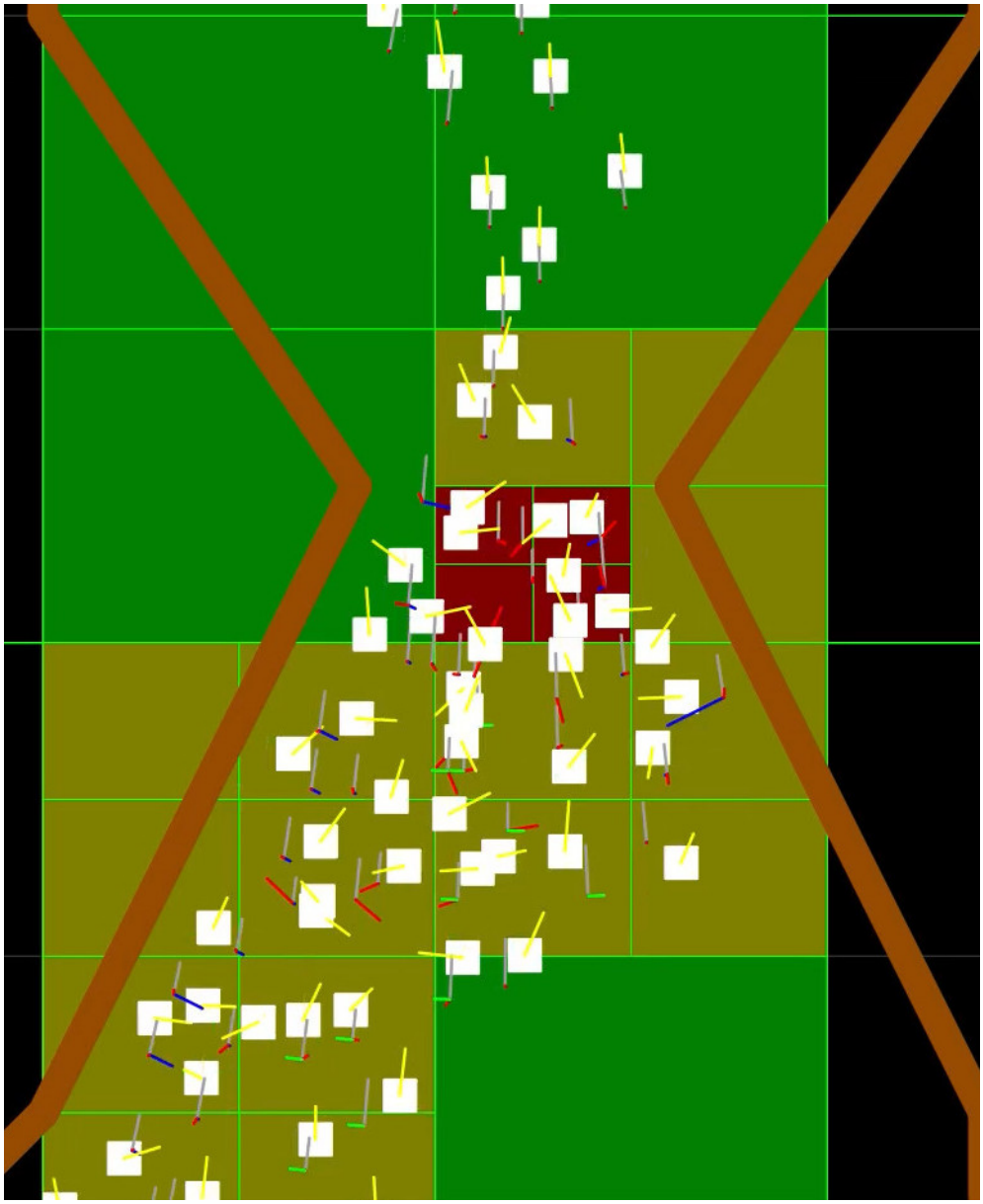

Abbildung 5: Screenshot aus 03-DichteEinfärbungNah.mp4

4 Diskussion

Gegenstand dieser Arbeit war die Beschreibung von PEDSIM und die Erweiterung um eine graphische Darstellungen der Agentendichte. Die Darstellung der Agentdichte kann über eine Einfärbung der Quadtrees in Relation zur Tiefe erreicht werden. Folgende nächste Schritte sind denkbar:

- Statt alle Quadranten einzufärben, könnten nur die Quadranten eingefärbt werden, die eine entsprechende Anzahl von Agenten enthalten.
- Statt die Dichte in Relation zu der Tiefe der Quadtrees zu setzten, könnte die tatsächliche Dichte gemessen werden.
- Auch andere Ansätze zur Darstellung der Agentendichte könnten verfolgt werden. So könnten die Agenten, der Dichte entsprechend, eingefärbt werden.
- Es könnten Szenarien mit unterschiedlichen Parametern wie Agentenanzahl und Agentengeschwindigkeit ausgeführt werden. Dabei könnte beobachtet werden, wie sich die Dichte verhält, wenn die Parameter verändert werden.

.

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 21.11.2020.

[3] Christian Gloor. libPEDSIM Documentation: Class Ped::Ttree. `http://pedsim.silmaril.org/documentation/libpedsim/latest/classPed_1_1Ttree.html`. Zugriff am 21.11.2020.

[4] Christian Gloor. libPEDSIM Documentation: ped_tree.cpp. `http://pedsim.silmaril.org/documentation/libpedsim/latest/ped__tree_8cpp_source.html`. Zugriff am 21.11.2020.

[5] Christian Gloor. PEDSIM. A Pedestrian Crowd Simulation System .Motivation, Usage, Installation and Library Documentation. `http://pedsim.silmaril.org/documentation/libpedsim/libpedsim.pdf`. Zugriff am 21.11.2020.

[6] Christian Gloor. Pedsim demoapp documentation: Class tree. `http://pedsim.silmaril.org/documentation/demoapp/classTree.html`. Zugriff am 21.11.2020.

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

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

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

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

[11] The Qt Company. Qt Documentation: QBrush. `https://doc.qt.io/qt-5/qbrush.html`. Zugriff am 21.11.2020.

[12] The Qt Company. Qt Documentation: QPen. `https://doc.qt.io/qt-5/qpen.html`. Zugriff am 21.11.2020.

[13] The Qt Company. Qt Documentation: QQGraphicsScene Class addRect. `https://doc.qt.io/qt-5/qgraphicsscene.html#addRect`. Zugriff am 21.11.2020.

[14] Thomas Richards. A REVIEW OF SOFTWARE FOR CROWD SIMULATION. `https://urban-analytics.github.io/dust/docs/ped_sim_review.pdf`, March 2020. Zugriff am 21.11.2020.

Anhang

Videos

- Webordner mit allen Videos: `https://th-koeln.sciebo.de/s/rxeOlDfda8b8lHD`
- 01-Beispielszenario.mp4: `https://th-koeln.sciebo.de/s/wDEw1DdSFvmtPqo`
- 02-DichteEinfärbung.mp4: `https://th-koeln.sciebo.de/s/ZrFdLnGRK9AbSWt`
- 03-DichteEinfärbungNah.mp4: `https://th-koeln.sciebo.de/s/Q8w9wRM66ypfsnN`