TU WIEN Informatics

Agent Based Pedestrian Simulation in Visdom

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Stefan Zaufl, BSc
Matrikelnummer 00925357

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr. Jürgen Waser

Wien, 27. November 2021

Stefan Zaufl	Eduard Gröller

TU WIEN Informatics

Agent Based Pedestrian Simulation in Visdom

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Stefan Zaufl, BSc
Registration Number 00925357

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr. Jürgen Waser

Vienna, 27th November, 2021

Stefan Zaufl	Eduard Gröller

Erklärung zur Verfassung der Arbeit

Stefan Zaufl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. November 2021

Stefan Zaufl

Danksagung

Ich möchte mich bei Dr. Eduard Gröller und Dr. Jürgen Waser für die Beteuung der Arbeit bedanken. Durch ihren Einsatz konnte die Qualität nochmals gesteigert werden. Mein Dank geht auch an meinen Eltern und meinem Bruder, die mich immer unterstützt haben. Weiters möchte ich mich bei meiner Freundin Melany Rieger bedanken, die mich immer vorangetrieben hat.

Nicht zu vergessen sind außerdem die unglaublich netten Leute meiner Tanzgruppe, die immer einen guten Ausgleich zur Arbeit und dem Schreiben dieser Ausarbeitung schaffen konnten.

Zu guter Letzt möchte ich auch noch meinen Arbeiteiskollegen danken - insbesondere Tanja Weiringer und Petra Halasz ohne die ich diese Arbeit wohl nie zu Ende gebracht hätte.

Acknowledgements

I would like to thank Dr. Eduard Gröller and Dr. Jürgen Waser for their supervision and support. Because of their continued efforts the quality of this work was improved. I would also like to thank my parents and my brother who always supported me. Thanks are also due to my girlfriend Melany Rieger, who always pushed me beyond my comfort zone.

We must not forget my dancing group which consists of extraordinary people and always help in maintaining a healthy work-life balance.

Last but not least I would like to thank my colleagues - especially Tanja Weiringer and Petra Halasz. Without them this work would still be unfinished.

Kurzfassung

In dieser Arbeit wird eine neue Fußgängersimulation in Form eines Plugins für das Visdom Visualisierungssystem präsentiert. Zuerst wird über die allgemeine Struktur einer solchen Simulation gesprochen indem ein Schichtenansatz verfolgt wird. Unterschiedliche Kandidaten für die Schichten werden präsentiert und die am besten geeignetsten ausgewählt. Für die taktische Schicht wurde ein Schnellster-Pfad Algorithmus ausgewählt und für die operationale Schicht wurde eine modifizierte Version des ORCA (Optimal Reciprocal Collision Avoidance) Algorithmus verwendet. Diese werden im Detail beleuchtet.

Außerdem wird die Visualisierung des Simulationszustandes erkärt. Es wird auf die unterschiedlichen Arten, auf die die Objekte der Simulation dargestellt werden, eingegangen, sowie auf zusätzliche Visualisierungen, die dem/der BenutzerIn helfen sollen, tiefere Einblicke in die Daten zu gewinnen.

Danach wird die konkrete Implementierung unter die Lupe genommen. Vor allem wie die unterschiedlichen Schichten in der Visdom Application integriert sind.

Im letzten Kapitel werden die Resultate diskutiert. Zuerst wird das System mittels der RiMEA Testfälle validiert. Danach werden mittels einer Studie eines Echtweltszenarios die Fähigkeiten des Systems präsentiert.

Abstract

In this thesis a new pedestrian simulation plugin for the Visdom visualization system is presented. First the general layout of such a system is discussed using a layered system. Different candidates for these layers are presented and the best fitting one for the use are picked. For the tactical layer a fastest-path algorithm is used whilst for the operational layer a modified ORCA (Optimal Reciprocal Collision Avoidance) algorithm has been implemented. These will be discussed in detail.

We will also talk about the visualization of the simulation state. Both the kind of representation for different objects in the world, as well as additional visualizations that aim to help the operator to gain insight, are going to be presented.

Then the implementation itself is going to be discussed. Especially how the different layers are integrated into the Visdom application.

In the last chapter the results are presented. First the system will be validated using the RiMEA test cases and then a real world case study showcases the capabilities of the proposed system.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

The VRVis [vrv] is developing a scenario based decision support system in the environmental and geospatial domain called Visdom [vis]. Currently this system supports the simulation and visualization of river flooding and protection measures against that, surface run-off in heavy rain, a sewer simulation that is coupled with the surface, and a logistics simulation for setting up counter-measures. The main goal of this software is to support decision makers by playing through different scenarios. The user can then make an informed decision on actions that have to be taken. Because Visdom computes most of their simulations on the Graphics Processing Unit (GPU), it is able to do these faster than real-time, which enables it to be used in time-critical situations that require fast actions.

To further enrich the utility, decision makers can draw from Visdom, an agent-based pedestrian simulation should be added to the system. As the main focus of the system is the disaster simulation, the pedestrian model should be an evacuation simulation. The reason for this is that planning evacuations can also be important in a flooding event. If for example the local hospital is endangered to be flooded, the software should provide the tools for planning the evacuation of said building. It is also possible to plan for disasters in outdoor events with this kind of simulation. A typical example for this would be a festival with many people like the Donauinselfest in Vienna.

The overall goal of the work is to implement an agent based simulation that should resemble reality as close as possible, yet run in real-time to enable decision makers to respond to changes as fast as possible. Because the user has to make decisions based on the outcome of the simulation, both the visualization of the result and the interactions with the simulated world are important. As a result the interface should feature tools for the user that help identifying dangerous regions in the simulated area, track the path of

individual agents, investigate the flow of agents in a bottleneck and change the simulation to discover other threats or solutions in dangerous situations.

1.2 Aim of the Work

The goal of this work is to create an agent based simulation. We have three main areas to address: the simulation itself, the visualization, and the interactivity. The simulation of pedestrians is not easy and still an ongoing research area. Because of this the first task has to be the identification of a simulation model that adheres to the following constrains: It has to be agent based, otherwise the user cannot follow the path of individual pedestrians. It has to run faster than real-time otherwise the interactivity is not optimal and decisions will be delayed. It should be stable, so agents may not pass through obstacles and small changes in the world should not strongly affect the outcome of the simulation. It should be reproduceable, multiple runs of the same simulation should yield the same result. And finally it should mimic real-world crowd behavior as close as possible.

The methods that fulfill these requirements will be integrated in the Visdom software as a plugin. Visdom features a branching time-line model [RWF+13] that the plugin has to be compatible with. This model empowers the user to create alternative time-lines by changing parameters in the simulation while keeping the prior configuration for comparison. Parameters can range from numbers for the used models to adding walls, or new pedestrians. This requires the plugin to support state serialization and loading. The integration of the model will benefit the interactivity of the simulation as the user can quickly change the simulated world and evaluate this change against the prior state.

The technical side of the result visualization will be managed by Visdom and the plugin will use the system's build-in tools to present the simulation data. Agents will be modeled as glyphs to provide all necessary information while reducing visual noise that could potentially distract the user. The glyphs should at least encode position, orientation, and speed of the agent. The floor will also be used to display information such as density or agent paths using a heat map. Walls and obstacles will be displayed and modeled as lines to avoid occlusion problems. The world will be rendered in 3D, because agents may also move vertically (e.g., walking up a small hill).

The central research question of this work is how does the combination of the Quickest Path Model by Kretz et al. [KGH+11] and a variation of the ORCA Model by Curtis and Manocha [CM14] perform? How is the computational performance of the combination of these models and how well does this combination of models describe the reality?

1.3 Methodological Approach

The methodological approach consists of these steps:

1. **Literature Study**

Gather information about existing models and visualizations and decide which ones to use.

2. **Implementation of the Plugin**
   The plugin has to be implemented using the gathered information.

3. **Validation**
   The implemented plugin has to be tested and validated using:

   a) **RiMEA Test Cases** [rim]
   These test cases are used in Germany to validate pedestrian models.

   b) **Real World Study**
   Test the implementation based on a real-world scenario.

1.4 Contributions

The main contributions of this work are:

- The simulation of pedestrians as a combination of the Quickest Path Model by Kretz et al. [KGH+11] and a variation of the ORCA Model by Curtis and Manocha [CM14]
- Implementation for a scenario-based decision support system. This implies a very strict state handling that is optimized for branching and switching between different time-lines.
- Coupling with the flood and storm weather simulation via a danger domain
- Integrated on-the-fly visualization of relevant evacuation information to help identify environmental hazards such as choke points and areas of high pedestrian density
- Validation of the simulation using the RiMEA test cases [rim]
- A real world large-scale case study of a concert evacuation

CHAPTER 2

Analysis of Existing Approaches

2.1 Pedestrian Simulation

We will use Hoogendoorn and Bovy's [HB04] categorization of pedestrian simulations. They use three levels: strategic, tactical and operational (see Fig. 2.1). The strategic level makes high level decisions that have a destination as output, for example the goal of buying a ticket at the nearest ticket station. The tactical level then takes care of planning a route to the given destination and has a desired direction or a sub goal as output. The operational level takes this and moves the agent. It makes sure that the agent resolves collisions with other agents and does not pass through obstacles.

Though it is useful to have this categorization, not all systems implement all three levels

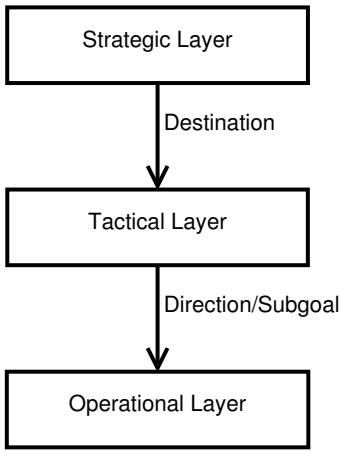


Figure 2.1: The three layers of a pedestrian simulation according to Hoogendoorn and Bovy's [HB04] definition

and they might spend more energy and afford on one level than on the others. For example, a simulation of pedestrians crossing a street [FDCZ13] does not need to have a strategic level, because the goal of the agent is predefined: it has to walk to the other side of the street. The tactical level is very primitive as the desired velocity only points in a fixed direction, as there are no static obstacles on a crosswalk. All of the complexity is in the operational level.

2.1.1 Operational Level

The operational level executes the actual movement of the agent. It can host a variety of different models, but we will structure them into three main categories: Cellular Automata, Social Forces Models, and optimization techniques. There is also a fourth category that we will not cover as it seems to have become less popular. It is called the behavioral model where agents act on a predefined set of rules like in a state machine.

Cellular Automata

Cellular Automata divide the space into regular convex regions called "cells". A cell can hold multiple pedestrians or just one, depending on the model. In case of the work of Feng et al. [FDCZ13] a cell can hold only one agent, because they chose the grid size in a way that only a single agent would fit into it space wise. In their simulation agents cross a street (see Figure 2.2). The model for the locomotion is discrete so every agent can move from one cell to another one, but cannot stand between cells. Pedestrians choose, based on local information stored in the grid of cells, where to move next and they are updated sequentially, one after another. In their work they encouraged agents to form lanes by adding an attraction parameter that boosts the probability of an agent following another one with the same target, but repels it from the other ones with a different target - the side it started on.

Social Forces Models

The original version of the Social Forces Model ws created by Helbing and Molnár [HM95]. They suggest that the motion of pedestrians can be described as a physics based particle simulation where the particles are subject to social forces. Multiple forces are applied to the agent in each simulation step in order to compute the next position. These forces can be simple like a force pointing in the direction of the desired target position or be more complex like a force acting between agents - pushing or pulling it towards/away from each other. They are also computed for each individual agent and do not have to be symmetrical. For example when looking at agent A there might be a force pulling agent A towards agent B and at the same time when looking at agent B there might be a force that pushes agent B away from agent A. This is a very popular model as it is highly customizeable and offers many parameters to customize it. Many variants of this model have been published over the years.

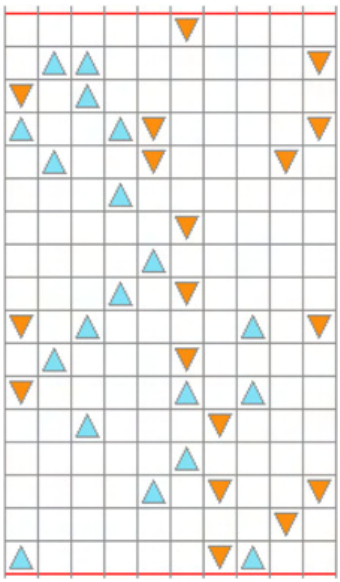


Figure 2.2: Simulated pedestrians (agents) on a grid. Pedestrians start from the top and the bottom, trying to reach the other end. Figure by Feng et al. [FDCZ13]

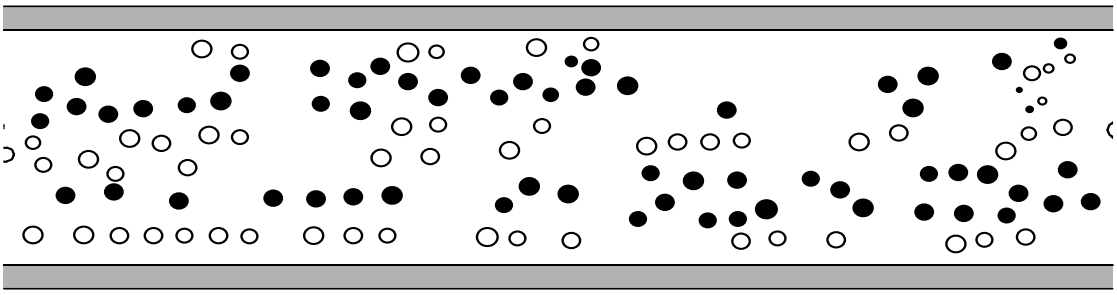


Figure 2.3: Lane formation of agents using a social forces model. The radius of the circles represent the velocity of the agents. Figure by Helbing and Molnár [HM95]

Optimization Techniques

Guy et al. presented PLEdstrian [GCC+10]. This method uses the Principle of Least Effort (PLE) to simulate pedestrians more realistically than for example a shortest path algorithm. A comparison of the operative model of PLEdestrian to the social forces model can be seen in Figure 2.4. They defined an energy function that each agent wants to minimize in order to maximize their comfort. This function drives the operative and the tactical layer's decisions. In the tactical layer they have a network with all possible intermediate points between the agent and their goal and each edge of the network has an energy value assigned to it. By using an A* algorithm they can determine the best next sub-goal for the agent. In the operative layer they evaluate the continuous energy function in the immediate neighborhood and look for the minimum there in order to extract a few candidates for the velocity. In order to enable the algorithm to run in real

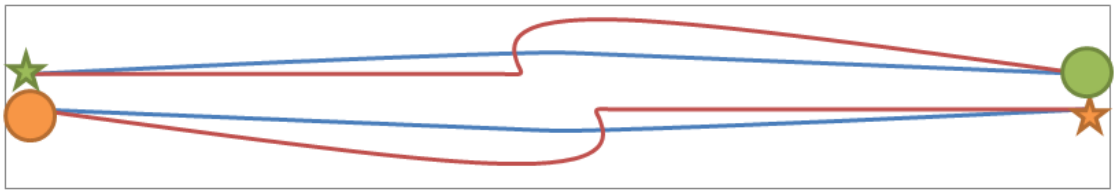


Figure 2.4: Paths of two agents avoiding each other. Comparison between Helbing and Molnár's Social Forces Model (red path) and PLEdstrian (blue path). Figure by Guy et al. [GCC+10]

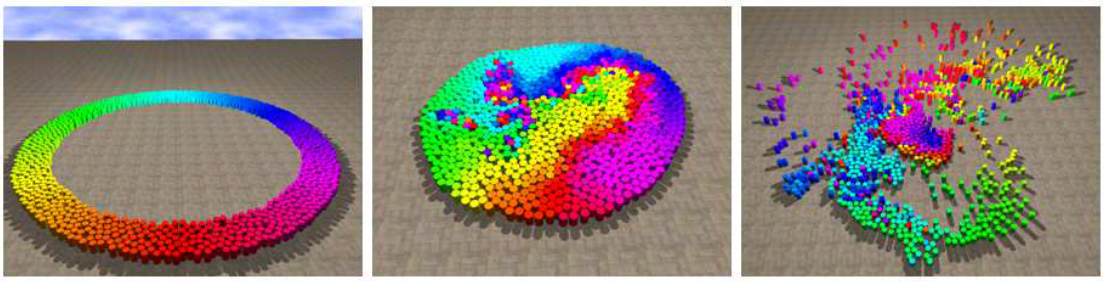


Figure 2.5: 1000 agents trying to pass through the middle of the circle. Figure by Berg et al. [VDBGLM11]

time they use a heuristic instead of the true minimum calculation.

Berg et al. developed a method for robots to move without colliding with each other or their environment: Optimal Reciprocal n-Body Collision Avoidance (ORCA) [VDBGLM11]. They use Velocity Obstacles (VO) [FS98] to judge if the agent is about to collide with other ones. Using them they can extract a half-plane in velocity-space for all other agents that represents a save zone for the current agent to pick its velocity. These half-planes can be seen as a linear program where the goal is to stay as close as possible to the desired velocity. This method has been adapted for pedestrian simulation by applying a speed model that limits the maximum velocity of the agent by the density [CM14]. See Figure 2.5 for an example.

2.1.2 Tactical Level

The tactical layer plans a path around static obstacles and creates subgoals for the operational layer that can be reached by walking there in a straight line. These models can be categorized into shortest path and quickest path methods.

Shortest Path

One way to find a valid path for an agent is to compute the shortest possible path. People want to minimize their energy spent walking to maximize their comfort, so they naturally seek for the shortest path. Most of these algorithms generate a weighted graph of the

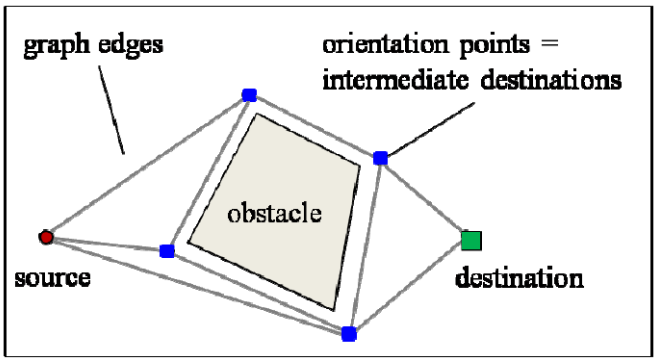


Figure 2.6: A visibility graph connecting the source and the destination. Figure by Höcker et al. [HBK+10]

environment and use some sort of Dijkstra's algorithm to extract the shortest path to the destination.

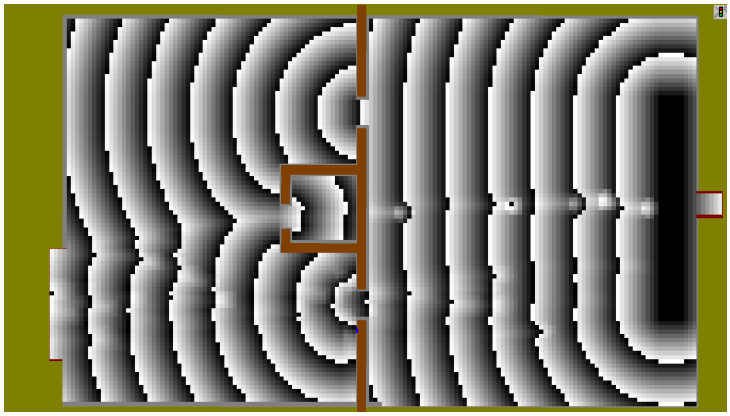
The easiest method of generating such a graph discretizes the space into regular convex polygons [BT00] - the most popular one is the square. A cell can be blocked (by an obstacle or a wall) or empty so agents may pass through it. The navigation graph is implicitly given by the structure of the discretization: each cell is a node and neighboring cells are connected if none of the two is blocked. These graphs work well if all obstacles can be optimally described in these grids and agents always walk perfectly aligned to the grid. Otherwise the space discretization creates visible artifacts. For example, when using squares agents tend to walk in zig-zag patterns when they should move diagonally. The algorithm is very easy to implement and can be extended to overcome most shortcomings (like casting rays on diagonals to find out if the direct path is free).

A visibility graph (see Figure 2.6) takes a set of obstacles and creates nodes at the corners of them [HBK+10]. Then all nodes are connected, except for those that are not mutually visible by each other. The resulting navigation graph leads the agents around the corners of the obstacles. This method works very well at low agent densities, but at higher densities agents tend to pile up before a corner, because all of them want to pass as close by the corner as possible. This is due to the fact that shortest path algorithms are designed for a single agent.

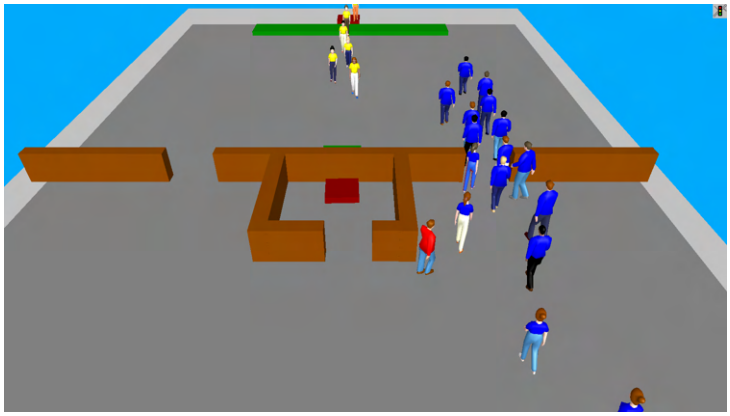
Quickest Path

Another way to think about this problem is to change the optimized value: instead of optimizing for the shortest path, we optimize for one that takes the least amount of time to walk - the quickest path. Calculating the actually quickest path does not reflect human behavior as we make mistakes due to incomplete information. The goal must be to create an accurate model of the estimated quickest path that humans use.

A possible way to achieve this is by using a dynamic distance field [KGH+11] (see



(a) A dynamic distance field based on a grid. Figure by Kretz et al. [KGH+11]



(b) A render of the scene. Figure by Kretz et al. [KGH+11]

Figure 2.7: Quickest path model. Figures by Kretz et al. [KGH+11]

Figure 2.7). This method divides the walk-able space into a regular quadratic grid where each cell stores the current agent density and the estimated distance to the destination. The negative gradient of the distance field is used to extract the direction in which a pedestrian should move when standing on this cell. There are at least two distance fields: a static distance field that never changes throughout the simulation as it does not include any agents and a dynamic distance field that is updated in each step. The static field just includes the real distance from each cell to the destination and is computed by solving the Eikonal Equation for the field. The dynamic field takes the static field as a basis, but also factors in the agent densities of the cells. This allows it to estimate a lower speed for these cells effectively pushing the goal further away on this path. The algorithm is not perfect as it can only react to a congestion rather than avoiding it in advance.

Kretz et al. [KLH14] tried to address this problem by changing several things. First they were using a sparse graph that provides meaningful alternative routes for avoiding obstacles and congestion. This graph is generated using distance maps and obstacle

analysis. A graph generated that way differs from a simple visibility graph as it also places additional nodes in empty spaces to provide extra choice. Second they try to create a so-called "User-Equilibrium": the pedestrians should choose their paths in such a manner that the pedestrian density is as low as possible and simultaneously have the shortest possible traveling time. This is only achievable with an iterative approach as a non-iterative never always finds the best solution - it only finds good solutions. Iterative means that they have to compute the whole simulation multiple times to find paths with these low-density, quick paths.

2.2 Visualizations of Agent Based Data

Faninil and Calori [FC14] presented a system that can take either simulation or tracking data (measurements) and visualize it. The system aims at presenting the data in a realistic way, so for visualization 3D-models of the buildings and pedestrians can be used. Besides the realistic rendering, a statistical module is also provided in the system that is capable of rendering lines for each pedestrian in order to visualize their path. Additionally these paths can be colored so the color represents the density of pedestrians at a particular point on the path - like a heat map does. Another interesting visualization the system is providing is a 3D density graph that gets drawn directly on top of the street. So the horizontal axes are the 2D-location in the world and the up-axis represents the density of the pedestrians at this particular point in space. An example can be seen in figure 2.8.

Handel1 et al. [HGPA15] also created a system that offers similar features to the prior presented system: it also takes already existing data of pedestrian positions and visualizes them in a realistic way. However, only one statistical analysis method is provided: a heat-map that shows the density of the pedestrians (see Figure 2.9). It is computed by counting the number of pedestrians in a cell and normalizing it by the total number of pedestrians in a scene. The authors point out that a coarse grid works best with this kind of measurement.

Guo et al. [GWY+11] presented a visualization system that lets the user explore the behavior of cars, cyclists, pedestrians, and public transportation vehicles at a single crossing. It features multiple views: a stylized 2D representation of the crossing, a theme river showing the amount of traffic for each type and time, scatterplots and a parallel coordinate plot showing multi-dimensional data about the crossing. These views are linked to each other during brushing or selecting. The system enables the user to identify dangerous situations and analyze the cause.

2.3 Comparison and Summary of Existing Approaches

In this chapter we discussed existing methods for simulating pedestrians and how to visualize the generated data. Each of the discussed approaches has its flaws and benefits, there is no best solution that suits all cases perfectly. Cellular Automata are very well

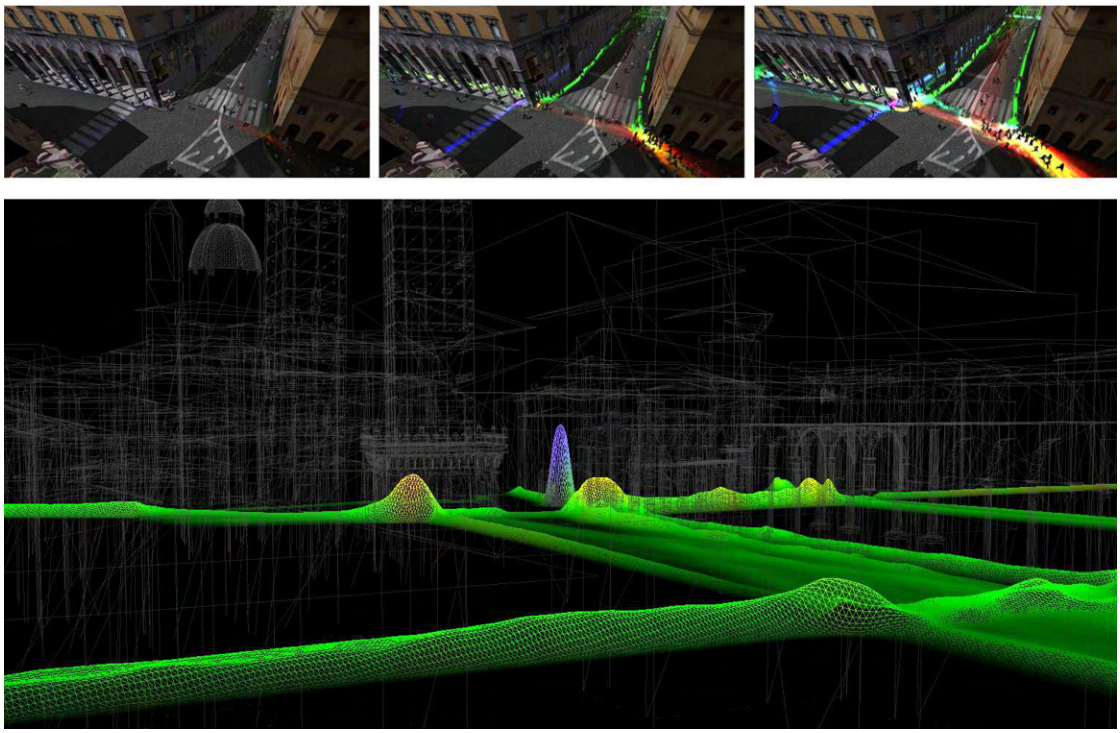


Figure 2.8: Fanini1 and Calori's [FC14] simulation system rendering. On the top: real-time overlay of the pedestrian's paths colored by density like a heat map. On the bottom: the 3D density graph of the pedestrians.

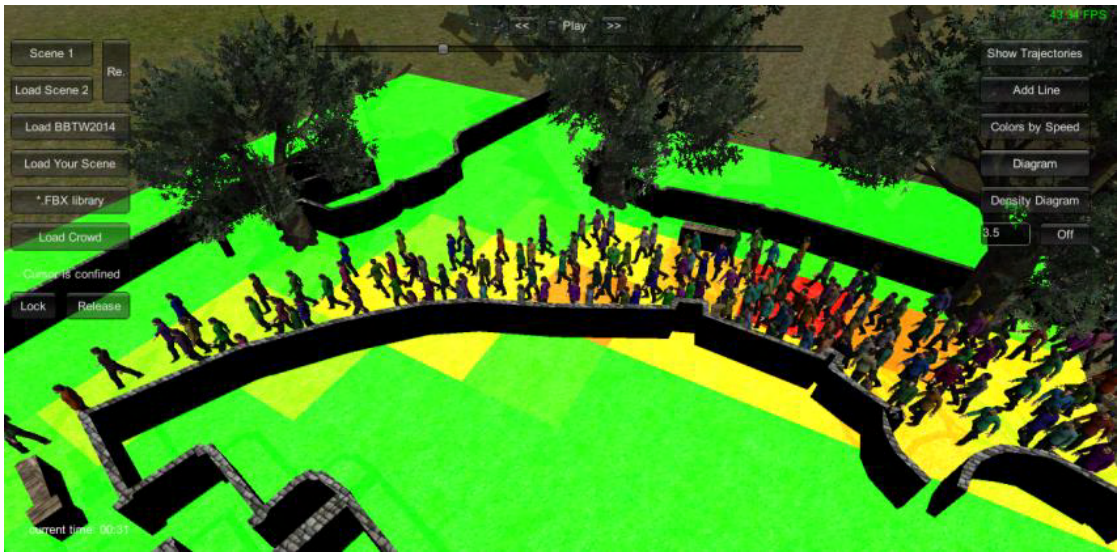


Figure 2.9: The density of agents is visualized by the color of the floor. Figure by Handel1 et al. [HGPA15]

suited for large-scale simulations as they are very fast, but the separation of the world into discrete positions does not reflect the reality accurately. Social Forces models do not have this drawback, but have very many parameters that need to be adjusted carefully. This is an advantage and a disadvantage at the same time: one the one hand the model is very flexible, on the other hand it is very labor-intensive to set up and each simulation might require a different configuration of parameters. Problems might also arise if the density of agents is too high. The forces acting on them in these situations are very high and artifacts like the shaking of agents or agents passing through walls might occur. Optimization methods do not have that kind of problem as they plan their path in a way that prevents these artifacts from occurring. The problem with these methods is often that they are too efficient - they have more information available than a normal human in certain situations and might move faster through dense crowds than an actual pedestrian.

At the tactical level, quickest path algorithms try to minimize travel time instead of travel distance, which improves the performance of the simulated agents as we will see later. These algorithms are more complex and computationally involved than pure shortest path solvers.

The majority of visualization algorithms discussed before focus on rendering a realistic view of the virtual scene. They then measure the density of agents and display it. Only Guo et al. [GWY+11] took an approach that uses techniques from the field of information visualization. By using glyphs instead of realistic representations of pedestrians it is possible to encode more information in the representation itself and simultaneously reduce visual noise that carries no information. In their work they encoded different types of agents in the glyph. A disadvantage of glyphs is that they are not intuitive and have to be learned in order to be effective.

CHAPTER 3

Agent Based Simulation for Evacuation

When creating an interactive pedestrian simulation we have to do actual simulation of the pedestrians first. Without the simulated data we would have nothing to display and the user would have nothing to interact with. This chapter will explain how we approach the challenge of simulating pedestrians using an agent-based method.

3.1 Overview

As discussed in the previous chapter, we divide the simulation of pedestrians into three layers [HB04]: the strategic, tactical and operational layer. The strategic layer is trivial in our approach - the user sets the strategic goal of the agents by placing target zones in

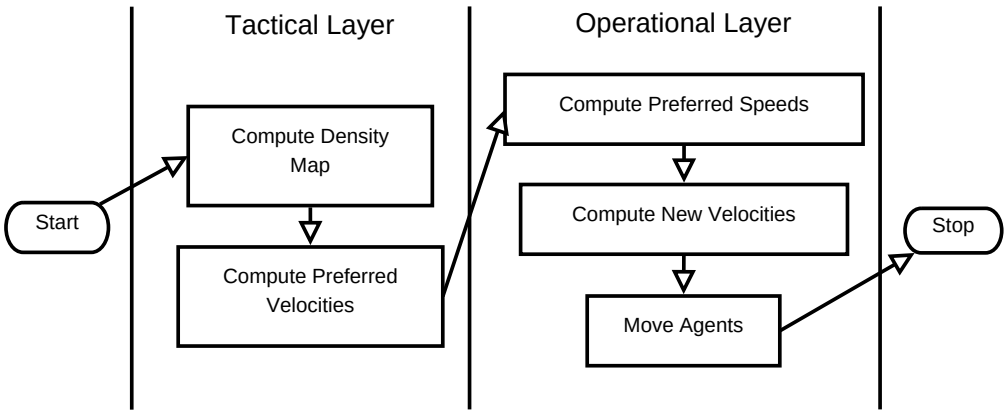


Figure 3.1: Overview of the simulation of a single timestep

the simulated world and assigning them to agents. Our simulation integrates over time by advancing the simulation time by a fixed amount and then recomputes the state of the system at that moment. This shift is called a timestep. The computation of one timestep can be seen in Figure 3.1 and starts with the tactical layer.

The tactical layer first of all computes a density map of the agents. This map is then used for the next step: the execution of the fastest path algorithm by Kretz et al. [KGH+11]. The results of this algorithm are the preferred velocities of the agents. These velocities would bring the agents closer to their goal, but might move them through each other.

The operational layer then takes these velocities as input. Following the method presented by Curtis and Manocha [CM14] the preferred speed is computed first. This speed takes the available space into account as humans have a stride length that limits their speed when they have limited space. Then geometric reasoning in velocity space is used to compute a collision-free trajectory for each agent to take. This involves selecting each agent individually and computing zones where the selected agent cannot go, because other agents will move there. This whole computation will take place in velocity space - this means that all positions are relative to the selected agent and the two dimensions describe a possible velocity of the selected agent. The details of this method will be explained later. This ends the computation in one timestep.

3.2 The Operational Layer

For the operational layer we chose the modified version of the Optimal Reciprocal n-Body Collision Avoidance (ORCA) model as presented by Curtis and Manocha [CM14]. This decision was made because in a prior attempt to implement an agent-based simulation one of the big problems has been agents passing through walls. By using the ORCA model we ensure that this can not happen again. The ORCA model has the problem of computing too high agent speeds for pedestrian simulation, so the speed model of Curtis and Manocha was used to eliminate this shortcoming.

3.2.1 Optimal Reciprocal n-Body Collision Avoidance (ORCA)

The Optimal Reciprocal n-Body Collision Avoidance (ORCA) algorithm tries to move agents in a virtual space without them colliding with each other. It was first presented by Berg et al. [VDBGLM11]. In their method we assume a 2D-space and each agent in it is represented by a circle and defined by a few properties: position, radius, velocity, maximum velocity and preferred velocity. See Table 3.1 for the variable names.

p_A	position of agent A
r_A	radius of agent A
v_A	velocity of agent A
v_A^{max}	maximum velocity of agent A
v_A^p	preferred velocity of agent A

Table 3.1: The definition of agent A

Roughly spoken the simulation of the agents is divided into steps. In each step a new velocity has to be computed in such a way that the agents are collision-free for a set amount of time τ. In order to compute the new velocity of a specific agent A we take every obstacle into account that might collide with agent A: walls, objects and other agents. For each obstacle a set of collision-free velocities is computed and then the intersection of all these sets is created. From the resulting set the closest velocity to the preferred velocity is taken as the new velocity for agent A.

The passing of time is simulated with the Euler method [IIC+96] which approximates the passing of time by dividing time equally. Each time-step has the same size τ which denotes how much time passed since the last step. All movements within a step are assumed to be linear which is of course not completely correct, but the error of this assumption is small for small step-sizes τ. Because the Euler method is a first-order method, the error per step is proportional to the square of the step-size τ.

At a more detailed look all computations are made in velocity space as suggested by Berg et al. [VDBGLM11]. The velocity space is the space of all possible velocities an agent may take. Not all velocities in the space are valid though: Because we want the agents to be collision-free they are not allowed to pick velocities that would result in a collision. Regions in the velocity space that would lead to a collision before τ time has passed are called velocity obstacles. These velocity obstacles don't take the velocity of the obstacle itself into account, for example, if the obstacle is another agent. In this case a collision will only occur, if the difference of velocities $v_A - v_B$ lies within the velocity obstacle.

For each velocity obstacle a half-plane is computed. This half plane contains the maximal amount of collision-free velocities for one agent and one velocity obstacle that are as close as possible to the agent's preferred velocity v_A^p, so it is optimal. If the velocity obstacle originates from another agent then the same half plane can be used for the other agent as well when mirrored by the velocity space's origin - so it is reciprocal. Because of these qualities this half plane is called 'Optimal Reciprocal Collision Avoidance' or ORCA for short [VDBGLM11].

The intersection of all ORCAs from all velocity obstacles is the region of allowed velocities. From this region we will the velocity that is closest to the agent's preferred velocity v_A^p by using linear programming [VDBGLM11].

We will now discuss the detailed approach to creating the ORCA for an agent, then we will discuss how to adapt these computations for static objects.

Given two agents A and B: we want to compute the ORCA for agent B when using agent A as our reference. So agent A is our current agent we want to compute the velocity for and agent B is the obstacle A has to avoid. The velocity obstacle's shape can be scribes as follows: take a circle and put a line through the velocity space's origin and the origin of the circle. Now move the circle along the line away from the velocity space's origin and scale its radius at the same rate as you move it. The area the circle swept over is the velocity obstacle. This circle-sweep starts for agent B's velocity obstacle (VO) for agent

A at the following position and radius in velocity space (read: position of the velocity obstacle to A induced by B):

$$P_{VO_{A|B}} = p_B - p_A \tag{3.1}$$

$$r_{VO_{A|B}} = r_A + r_B \tag{3.2}$$

This is sufficient if the time τ during which we want to guarantee collision-free movement is 0. For a non-zero amount of time τ we have to start with the following values:

$$P_{VO^{\tau}_{A|B}} = \frac{p_B - p_A}{\tau} \tag{3.3}$$

$$r_{VO^{\tau}_{A|B}} = \frac{r_A + r_B}{\tau} \tag{3.4}$$

Note that for larger values of τ the velocity obstacle comes closer to the velocity space's origin thus limiting the possible velocities to pick from even further. As this happens for each agent in the simulation a large value of τ will cause agents to move slower when they come near obstacles or move in crowds.

Another way of describing a velocity obstacle is via sets. Let $D(p, r)$ denote a disk:

$$D(p, r) = \{q \,|\, \|q - p\| < r\} \tag{3.5}$$

then the velocity obstacle $VO^{\tau}_{A|B}$ can be defined as:

$$VO^{\tau}_{A|B} = \{v \,|\, \exists t \in [0, \tau] :: tv \in D(p_B - p_A, r_A + r_B)\} \tag{3.6}$$

A graphical representation of velocity obstacles can be seen in Figure 3.2.

In order to guarantee that agent A and B are collision free for at least τ time, agent A's velocity has to be picked in such a way that $v_A - v_B$ does not lie within the velocity obstacle $VO^{\tau}_{A|B}$. Any set V_A that does not include a velocity that will lead to a collision is a permitted set. So V_A is an arbitrary set that is a subset of all possible velocities for agent A that are collision free. Equally a permitted set V_B does also exist for agent B. Because we want to pick the best velocity for agent A as well as for agent B we have to consider both sets at the same time.

Now let $X \bigoplus Y = \{x + y \,|\, x \in X, y \in Y\}$ denote the Minkowski sum of two sets. If $v_B \in V_B$ and $v_A \notin VO^{\tau}_{A|B} \bigoplus V_B$ then A and B are guaranteed to be collision free for at least τ time. $CA^{\tau}_{A|B}(V_B)$ is the set of *collision-avoiding velocities*.

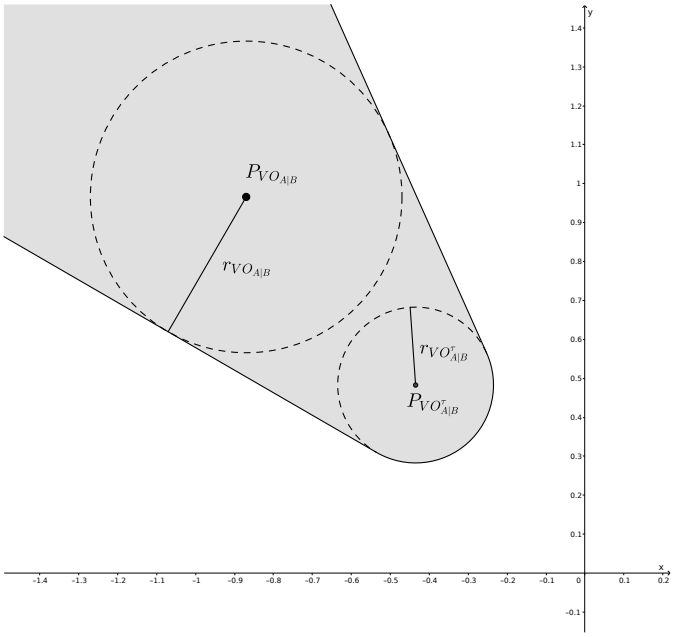


Figure 3.2: Velocity obstacle $VO^{\tau}_{A|B}$ in A's velocity space, figure redrawn from original paper [VDBGLM11]. The gray area contains all velocities $v_A - v_B$ that will result in a collision between agent A and B.

$$CA^{\tau}_{A|B}(V_B) = \{v \mid v \notin VO^{\tau}_{A|B} \bigoplus V_B\} \tag{3.7}$$

Because of the nature of the problem choosing the best solution for one agent might lead to a poor solution for another agent. For example if agent A and B are on a collision course, the best solution for A would be to stick with its velocity, but agent B would then have to make way for agent A and completely abandon its goal. What we want is a fair solution where both agents share the responsibility of avoiding collisions.

A possible way to do that is to define a half-plane in agent A's velocity space that includes only velocities that are guaranteed to be collision-free with respect to agent B: $ORCA^{\tau}_{A|B}$. To make the definition more general the *optimization velocities* v^{opt}_A for agent A and v^{opt}_B for agent B are introduced. Normally $v_A = v^{opt}_A$, but other choices are possible as well. In this work we chose v_A as our optimization velocity. Berg et al. formally defined this half-plane as follows [VDBGLM11, p. 6]:

"**Definition 1 (Optimal Reciprocal Collision Avoidance).** $ORCA^{\tau}_{A|B}$ and $ORCA^{\tau}_{B|A}$ are defined such that they are reciprocally collision-avoiding and maximal, i.e. $CA^{\tau}_{A|B}(ORCA^{\tau}_{B|A}) = ORCA^{\tau}_{A|B}$ and $CA^{\tau}_{B|A}(ORCA^{\tau}_{A|B}) = ORCA^{\tau}_{B|A}$, and such that for *all* other pairs of sets of reciprocally collision-

avoiding velocities V_A and V_B (i.e. $V_A \subseteq CA^{\tau}_{A|B}(V_B)$ and $V_B \subseteq CA^{\tau}_{B|A}(V_A)$), and for *all* radii $r > 0$,

$$|ORCA^{\tau}_{A|B} \cap D(v_A^{opt}, r)| = |ORCA^{\tau}_{B|A} \cap D(v_B^{opt}, r)| \geq min(|V_A \cap D(v_A^{opt}, r|, |V_B \cap D(v_B^{opt}, r)|).$$

This means that $ORCA^{\tau}_{A|B}$ and $ORCA^{\tau}_{B|A}$ contain more velocities close to v_A^{opt} and v_B^{opt}, respectively, than any other pair of sets of reciprocally collision-avoiding velocities. Also the distribution of permitted velocities is "fair", as the amount of velocities close to the optimization velocity is equal for A and B."

In order to construct such an $ORCA^{\tau}_{A|B}$ we need to define the vector u. u is a vector in agent A's velocity space that points from $v_A - v_B$ to the nearest point on the edge of the velocity obstacle $VO^{\tau}_{A|B}$. More formally:

$$u = \left(\underset{v \in \partial VO^{\tau}_{A|B}}{\arg\min} ||v - \left(v_A^{opt} - v_B^{opt}\right)|| \right) - \left(v_A^{opt} - v_B^{opt}\right) \tag{3.8}$$

If $v_A - v_B$ lies within the velocity obstacle, u is the minimal correction that agent A needs to apply to its current velocity in order to stay collision-free. If $v_A - v_B$ lies outside of the velocity obstacle, u sets a maximum delta that can be applied to A's velocity if we want to stay collision-free. Because the other agent B will run the same calculations we can assume that B will also try to avoid agent A. So when using u to correct the current velocity we will assign half of the responsibilities to each agent. So instead of using u to correct the velocity, we will only use $\frac{u}{2}$. This way both agents will share the responsibility of avoiding a collision equally.

Now let n be the unit normal vector at the point on the velocity obstacle where u points at. In our example above n defines the direction in velocity space that will always be collision free. With both u and n defined we can now define $ORCA^{\tau}_{A|B}$:

$$ORCA^{\tau}_{A|B} = \left\{ v \mid \left(v - \left(v_A + \frac{u}{2} \right) \right) \cdot n \geq 0 \right\} \tag{3.9}$$

Figure 3.3 shows the graphical representation of the above equation.

Now we will take a look at how Berg et al. solved the problem for multiple agents. First an optimal reciprocal collision avoiding set (see Equation 3.9) is computed for each agent that might collide with our current agent A. So if for example agent B, C and D are in the vicinity of agent A we will calculate $ORCA^{\tau}_{A|B}$, $ORCA^{\tau}_{A|C}$ and $ORCA^{\tau}_{A|D}$. A collision between two agents A and B might occur if $p_B \in D\left(p_A, v_A^{max} + v_B^{max}\right)$. Then

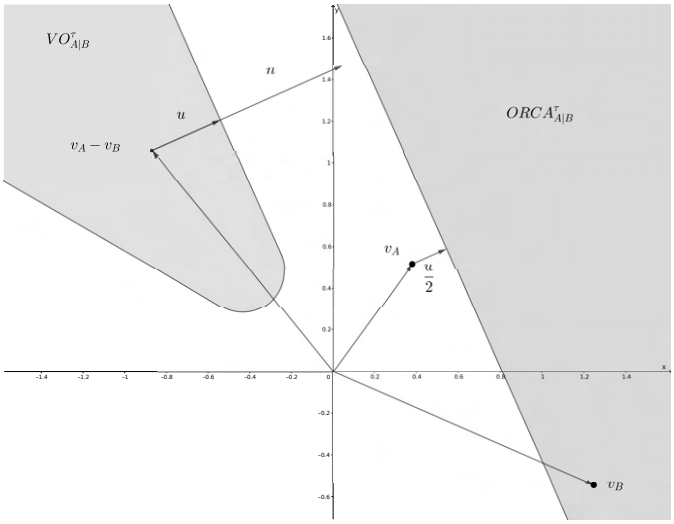


Figure 3.3: Graphical representation of $ORCA^{\tau}_{A|B}$: the set of permitted velocities for agent A with respect to agent B in A's velocity space, figure redrawn from the original paper [VDBGLM11]

the closest velocity to the preferred velocity v^p_A that is still contained in the intersection of all prior computed half-planes is chosen for the next simulation step (see Figure 3.4). Using a modified version of linear programming that takes the maximum velocity of the agent into account can solve this problem efficiently is recommended.

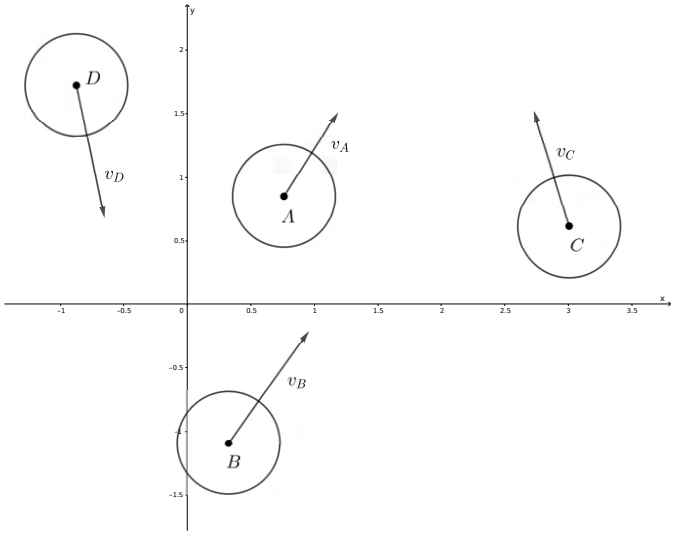
Under very dense conditions this linear system may not be solvable which means that we cannot guarantee a collision-free trajectory for the next τ amount of time. Berg et al. proposed using a 3-dimensional linear system in this case that tries to move the agent in such a way that it is overlapping as little as possible with the other agents. In Order to achieve this the half-planes are placed on a 45° angle around their intersection with the xy-plane in such a way that the normal vector of the plane points in the negative z direction. This system is guaranteed to have a solution.

All static obstacles are represented as lines. This has the advantage of simpler computations and more complex obstacles can be represented as a series of lines. So if for example we want to model a desk, we would use 4 lines that make up the boundary of the desk.

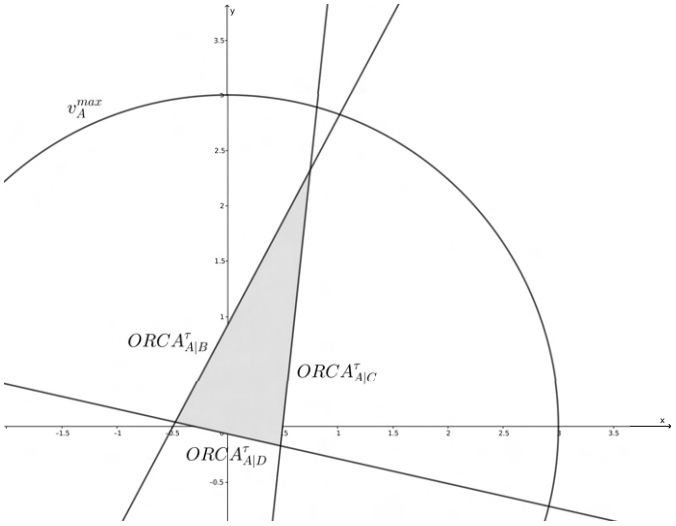
The velocity obstacle of lines is a 2D capsule. A 2D capsule can be imagined as a circle that is cut in half, the two semi-circles are then moved away from each other along the normal direction of the cut and finally the opposing ends of the semi-circle are then connected with lines. More formally:

$$VO^{\tau}_{A|O} = \{v \mid \exists t \in [0, \tau] :: tv \in \oplus - D(p_A, r_A)\} \tag{3.10}$$

As O is not moving the inverted set of the velocity obstacle would be fully sufficient in



(a) The agents and their current velocities in world space



(b) The $ORCA$s with respect to agent A and their intersection in A's velocity space

Figure 3.4: Avoiding multiple agents from the perspective of agent A, figure redrawn from the original paper [VDBGLM11]

choosing a collision-free velocity, but we want to also use it in our linear programming. So we define a half-plane for static obstacles as follows:

$$ORCA^{\tau}_{A|O} = \{v|(v - (v_A + u) \cdot n \geq 0\} \tag{3.11}$$

Note that for static obstacles we use u instead of $\frac{u}{2}$ which is used in the agent-agent case. As discussed before u represents the vector that has to be added to the agent's current velocity in order to stay collision free. When two agents avoid each other both take half of the responsibility, so each agent only has to account for $\frac{u}{2}$. With static obstacles the agent has to take full responsibility, because static obstacles cannot move, so the agent has to account for the full u.

3.2.2 ORCA for Pedestrian Simulation

The ORCA algorithm was originally designed for robots, but Curtis and Manocha pointed out that it could also be applied to pedestrians [CM14]. There they found that "The ORCA algorithm has several desirable properties" [CM14, p. 5] and pointed to other papers that stated the following points:

- Efficiency ($35,000$ agents in better than realtime on an Intel i7 running at 2.67 GHz) [CGZM11]
- Stability & consistency for large timesteps (0.2 s) [CSM12]
- Self-organizing behavior (jamming, lane formation, etc.) [GCLM12]
- Realistic microscopic interactions [GLM10]

There is one property that the model does not exhibit: pedestrians become slower if the density of the crowd increases. The interaction between speed and density can be visualized using a fundamental diagram [Wei93]. The fundamental diagram has two axis: one axis is the speed of the pedestrian and the second axis is the density of the crowd near the pedestrian where we measured the speed. They used the experimental data from Seyfried et al. [SSKB05] to show that the ORCA algorithm lacks this behavior as shown in Figure 3.5.

In order to address this issue Curtis and Manocha proposed a new model [CM14]. This model limits the maximum speed of the agent based on the density in such a way that it fits the experimental data [SSKB05]. The model consists of two factors: a physiological and a psychological one. The physiological factor captures how much space is available to take a stride, because the speed of a pedestrian depends on the stride length. The psychological factor describes how much people would like to stay away from others in order to keep comfortable.

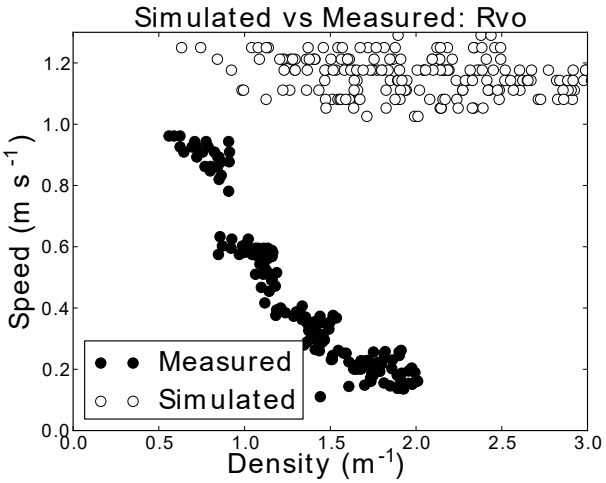


Figure 3.5: Fundamental diagram of measurements of pedestrians and simulated agents using ORCA. The speed of the simulated agents stays the same with increasing densitiy whereas the measurements show a decreas in the speed with increasing density. Figure from original paper[CM14]

The physiological constraint describes the relation between stride length (L) and the natural walking speed (v^{nat}). This was described by Dean [DCSRO65]. Formula 3.12 shows his model where $\alpha = 1.57$ is the stride factor and $H = \frac{height}{1.75}$ is the normalized height of the pedestrian in meters.

$$L(v^{nat}) = \frac{H}{\alpha}\sqrt{v^{nat}} \tag{3.12}$$

The psychological constraint models the mental repulsion between people: Two persons will keep a certain distance from each other if they walk past each other even if it means to take a slightly longer path. If this distance cannot be maintained the pedestrians will slow down even thought, physically speaking, they would have enough space to keep their current speed. This psychological constraint is not always equal: it's lower when two persons know each other and it also differs from culture to culture.

To model these complex interactions we will simplify it and use a factor that is multiplied with the natural stride length $L(v^{nat})$ in order to scale the physically needed space up, so agents in our simulation will slow down earlier than physically necessary. We will call this factor the *stride buffer* β. Curtis and Manocha used data by Seyfried et al. [SSKB05] to compute a stride buffer of $\beta = 0.9$. With β we can define the *extra buffer space* B:

$$B(v^{nat}) = \beta L(v) \tag{3.13}$$

With this information we can create a relationship between the available space S and the natural speed v^{nat} by simply adding the physiological constraint, given by Equation 3.12, to the psychological constraint, given by Equation 3.13.

$$S(v^{nat}) = L(v^{nat}) + B(v^{nat}) = \frac{H(1+\beta)}{\alpha}\sqrt{v^{nat}} \tag{3.14}$$

Equation 3.14 describes how much space must be available for a given speed. We need it the other way round: We need a model that describes what the natural speed for a given space is. Of course agents cannot become faster than their maximum speed v^{max}, so we have to limit it using a min function.

$$v^{nat}(S) = \min\left(v^{max}, \left(\frac{\alpha}{H(1+\beta)}S\right)^2\right) \tag{3.15}$$

In order to apply formula 3.15 to a simulation the available space has to be measured. Because an exact measurement is not trivial we will estimate it. This estimation is designed to model the impact of other agents on the current one. In order to compute it for one agent A we will measure the distance to every other agent in the neighborhood of A. This distance will be modified by the heading and position of each agent: agents that move away or are behind from our agent A will be perceived farther away while agents on a collision course or in front of agent A will be perceived closer than they really are (see Figure 3.6). We will call this the *effective distance metric* between two agents A and B e_{BA}. The final space estimation is then done by taking the minimum of all effective distances (see Equation 3.22).

First we will need to compute the maximum Euclidean distance where agent A might influenced by other agents and we will call it δ_A.

$$\delta_A = \frac{(1+\beta)H}{2\alpha}\sqrt{v_A^{max}} \tag{3.16}$$

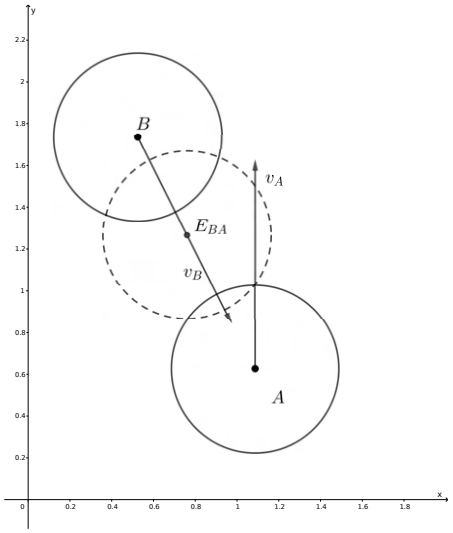
Then we will compute a penalty for the direction Δ_{BA} and a penalty for the orientation o_{BA}. Let in the following $\overrightarrow{v}^p_A$ the preferred direction of walking for agent A as a unit-vector.

$$d_{BA} = ||p_B - p_A|| \tag{3.17}$$

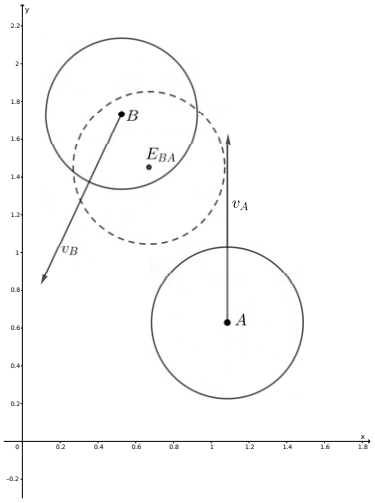
$$\overrightarrow{d}_{BA} = \frac{p_B - p_A}{d_{BA}} \tag{3.18}$$

$$\Delta_{BA} = \delta_A\left(1 - \left(\overrightarrow{v}^p_A \cdot \overrightarrow{d}_{BA}\right)\right) \tag{3.19}$$

$$o_{BA} = \max\left(r_B, \frac{H\sqrt{|v_B|}(1+\beta)|\overrightarrow{v}^p_B \cdot \overrightarrow{d}_{BA}|}{2\alpha}\right) \tag{3.20}$$



(a) Agent B is moving towards agent A and is perceived closer than it actually is.



(b) Agent B is moving away from agent A and is still perceived closer, because B is in front of A.

Figure 3.6: The effective position (E_{BA}) of agent B as perceived by agent A, figure redrawn from the original paper [VDBGLM11]

The effective distance e_{BA} between agent A and B is then just the Euclidean distance with the two penalties applied.

$$e_{BA} = d_{BA} + \Delta_{BA} - o_{BA} \tag{3.21}$$

Let N_A be the set of agents in the neighborhood of agent A, excluding agent A itself. Then the available space for agent A is the minimum effective distance between agent A and any agent in it's neighborhood.

$$S_A = \min_{n \in N_A} (E_{nA}) \tag{3.22}$$

3.2.3 Pedestrian Slope Speed Model

Our simulation allows the user to either choose a flat domain or a 2.5D domain. The 2.5D domain is based on a height-map, so it is capable of modeling hills, but not caves for example. In other words for each point in the domain there can only be one height assigned to it. Pedestrians moving up/down a slope are slower in contrast to moving on a flat surface, because they have to spend more energy for going up or downwards as they have to lift their body against gravity or stop themselves from tumbling down the hill.

The slope s is computed as shown in Equation 3.24 where v_{2D} is the 3-dimensional vector of the agent's velocity with its z-value set to 0 and v_{3D} is the 3-dimensional velocity of the

agent. Because our model is in 2D there is no 3-dimensional velocity vector available. In order to obtain it, we estimate the last position of the agent in 2D by adding the inverse velocity $-\overrightarrow{v}$ to the current position p and computing the height-difference between those two points. Let $HM(x)$ be the height of a point x on our height-map.

$$v_{3D} = \begin{pmatrix} \overrightarrow{v}_x \\ \overrightarrow{v}_y \\ HM(p) - HM(p - \overrightarrow{v}) \end{pmatrix} \tag{3.23}$$

Where $\overrightarrow{v}_x$ is the x-component of $\overrightarrow{v}$ and $\overrightarrow{v}_y$ the y-component of $\overrightarrow{v}$. The angle of the slope can then be computed in the following way:

$$s = \cos^{-1}\left(\frac{v_{2D} \cdot v_{3D}}{|v_{2D}||v_{3D}|}\right) \tag{3.24}$$

The slope is then used to compute the new preferred speed of the agent A based on the slope v_A^{slope} (see Equation 3.25).

$$v_A^{slope} = mMs + v_A^{max} \tag{3.25}$$

$$M = \frac{v_A^{max}}{1.84} \tag{3.26}$$

$$m \approx -1.4954 \tag{3.27}$$

Where M (Equation 3.26) normalizes the maximum speed by the average maximum speed so it reaches 0 at the same inclination for different maximum speeds. m (Equation 3.27) is a value obtained from approximating experimental data in meter times degree per seconds. The model was based on experimental data gathered by Fujiyama and Tylor [FT04].

This model is active at all times in addition to the speed model explained in section 3.2.2. To obtain the overall preferred speed of the agent the minimum of the two speed models is taken.

$$v_A^p = \min\left(v_A^{nat},\ v_A^{slope}\right) \tag{3.28}$$

3.2.4 Improving the Speed of the Algorithm

The operation layer so far consists of the ORCA-algorithm and two speed models that limit the maximum speed of the agent. In order to increase the computational speed of the algorithm we first have to understand where the bottlenecks are.

In the ORCA algorithm we have to iterate through every agent and check it against every other agent to make sure we do not collide with any of them. This would lead to a

quadratic runtime dependent on the number of agents. But depending on the situation we don't have to look at every other agent. Let's assume we have two agents A and B: A and B can only collide with each other if and only if $||p_A - p_B|| \leq r_A + r_B + \Delta t(v_A^{max} + v_B^{max})$, given Δt is the amount of time till the next simulation step. We can generalize this statement even more when using the maximum of all maximum velocities v^{max} and the maximum of all radii r^{max}:

$$||p_A - p_B|| \leq 2r^{max} + \Delta t 2v^{max} \tag{3.29}$$

All agents that fulfill this inequality are defined to be in the neighborhood of each other. This definition of the neighborhood might include agents that cannot collide with each other, but the simplifications allow for a less complicated search algorithm. We will call $2r^{max} + \Delta t 2v^{max}$ the *neighborhood radius*.

The neighborhood is also important when using the speed model of Curtis and Manocha [CM14], because we need to calculate the minimum effective distance to all agents in the neighborhood.

So we have to find a data structure that allows for fast retrieval of agents in the neighborhood of another agent as well as a short creation-time, because the structure has to be re-created in each simulation step as all agents are moving. Possible solutions to this are grid-based approaches like a regular search grid or a kd-tree [Ben75]. Because it is faster to build and easier to implement we chose a regular search-grid.

This data structure divides the domain into axis-aligned cells of equal sizes. Each cell contains a reference to all agents that have their center p inside the cell. To retrieve an agent's neighborhood a circle with the neighborhood radius is tested against the grid to retrieve a list of cells intersecting with said circle in $\Theta(1)$ runtime. Each agent in those cells is then tested in turn if they really lie within the requested boundary.

This means that the runtime of the algorithm depends on the number of agents that are retrieved, but this number is limited by the physical constraints of the agents. Agents have a radius and should not occupy the same space as other agents. In reality there might be some overlap as discussed in section 3.2.1, because the linear system might not be solvable and a collision might be unavoidable. But this should only happen under very dense conditions, so for the sake of this estimation we simplify by stating that agents do not overlap. So in this case the number of agents in a single cell is limited by the size of the cells in relation to the radius of the agents. This means that the runtime is ultimately dependent on the size of the cells in relation to the radius of the agents.

So smaller cells yield a better runtime, but the drawback of this is the memory consumption. Each cell has to be allocated in memory, so smaller cell sizes have a higher memory footprint. Also there is no benefit in cell sizes smaller than the agent radii, because then iterating through the cells will become the limiting factor as there are now more cells than agents in an agent's neighborhood.

So by using a regular grid for the neighborhood search we can cut computational time and speed up the computation of the models.

3.3 The Tactical Level

3.3.1 The Quickest Path Model

For the tactical level we decided to use a quickest path approach. Kretz et al. [KGH+11] presented an algorithm that is able to compute an approximation to the quickest path for pedestrians in realtime. It generates a discrete distance field to the targets, enriches it with additional information and then computes the first derivative of it. The direction in which the first derivative vector points is the direction an agent should walk. Each group of pedestrians with the same target can share such a vector field.

A distance field divides the domain into axis-aligned cells of equal sizes. Each cell contains the shortest distance from the goal. In a simple case, like standing on an open field, this distance is equal to the Euclidean distance from a cell's center point to the goal, but we also need to be able to model more complex scenarios with obstacles. This is not a simple problem and the true solution to this is generated by solving the Eikonal Equation [Fra27]. In order to improve performance we will approximate the solution to the Eikonal Equation by using a flood-filling algorithm.

The flood-filling algorithm computes the distance from some set of starting cells, also called seed cells, and assigning a distance from these starting cells to their neighbors. This is repeated with all neighbors until the whole domain has been computed. In order to assign a distance to neighboring cells we have to define two more fields: a *cost field* F_c and a *neighbor cost matrix* N_c.

The cost field F_c is needed for defining obstacles. This field has the same properties as our distance field, but instead of storing distances it stores the cost we need to add when entering the cell. If the cell is free, the cost is defined as 0 and if it is blocked, -1 is assigned. Later we can add other costs to this field in order to estimate the travel time instead of the distance.

The neighbor cost matrix N_c defines the cost that has to be added to the distance based on the direction of the movement. It can be described as a $3x3$ matrix:

$$N_c = \begin{pmatrix} n_{1,1} & n_{1,2} & n_{1,3} \\ n_{2,1} & n_{2,2} & n_{2,3} \\ n_{3,1} & n_{3,2} & n_{3,3} \end{pmatrix} \tag{3.30}$$

In this matrix the middle $n_{2,2}$ represents the current cell and the other 8 are its neighbors. The values in the matrix are the costs of going to that particular neighbor.

The values of the two costs F_c and N_c are added up in order to obtain the cost of entering a particular cell from a particular cell. For example if we want to get the cost of moving

from the cell at position p_1 to it's right neighbor at position $p_2 = p_1 + (1, 0)$ the cost would be calculated like this: $F_c(p_2) + n_{2,3}$

$$\text{cost} = \begin{cases} F_c(p_2) + N_c(p_2 - p_1) & \text{if } F_c(p_2) \geq 0 \\ \infty & \text{otherwise} \end{cases} \tag{3.31}$$

In our example $N_c(p_2 - p_1)$ accesses the neighbor cost matrix at $n_{2,3}$. This cost matrix is an approximation of the distance between the center points of the two cells. Choosing a cost matrix influences the resulting distance field fundamentally. Because we are using these cost matrices to measure distances we will call them metrics.

$$N_{chess} = \begin{pmatrix} \infty & 1 & \infty \\ 1 & 0 & 1 \\ \infty & 1 & \infty \end{pmatrix}, N_{man} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}, N_{\sqrt{2}} = \begin{pmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & 0 & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{pmatrix} \tag{3.32}$$

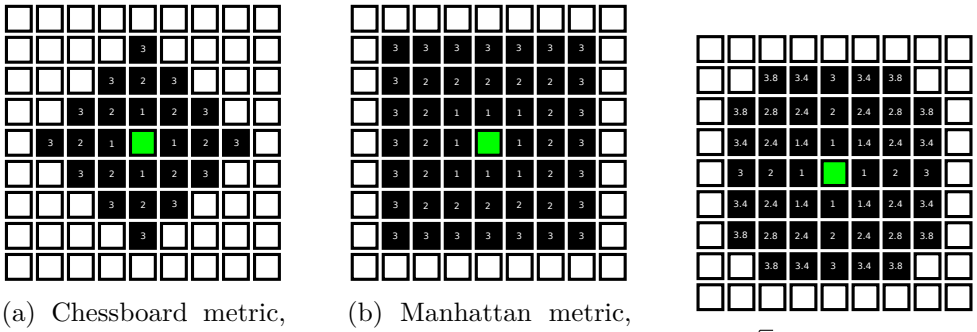
Equation 3.32 shows three popular cost matrices: N_{chess} results in the chessboard metric, N_{man} results in the Manhattan metric, and $N_{\sqrt{2}}$ results in the $\sqrt{2}$-metric. When talking about these metrics we will imagine a walker that walks on grid cells always applying the cost matrix for each step it takes. Of all possible moves to reach a cell, we will only look at the move with the minimal cost.

As discussed before, the ground truth for the metric is the solution of the Eikonal Equation. When coloring in all cells of the distance field containing the solution of the Eikonal Equation smaller than a set distance, the colored cells will form a circle on an open field with no obstacles. This image will serve us as the ground truth to all of our approximations done by flood filling using a certain metric.

As seen in Figure 3.7a The chessboard metric creates s diamond-shape after a fixed distance, because the walker cannot walk diagonally. In contrast to that with the Manhattan metric the walker is allowed to walk diagonally - with the same cost as going straight. This results in a square and can be seen in Figure 3.7b. Both the Chessboard as well as the Manhattan metric have large errors compared to the ground truth, e.g., for the 8 neighbors of a cell the mean error (mean deviation from the ground truth) is 20.7% for the Chessboard metric and 14.6% for the Manhattan metric.

The $\sqrt{2}$-metric is the middle ground between the former two. Here the walker is also allowed to walk diagonally, but it costs as much as the actual distance between the two cells is, i.e., $\sqrt{2}$. On a small scale this seems like a perfect approximation, e.g., for the 8 neighbors of a cell the mean error compared to the ground truth is 0%. This error grows with larger distances from the start, e.g., for the 24 cells (2 rows) surrounding a certain cell the mean error amounts to 2.7%. The shape produced after a certain distance is an approximation of a circle. The result of this can be seen in Figure 3.7c.

Now we can put everything together in the flood filling algorithm describes in Algorithm 3.1. For this algorithm we will use a queue to keep track of cells where we still



(a) Chessboard metric, see N_{chess} in Equation 3.32

(b) Manhattan metric, see N_{man} in Equation 3.32

(c) $\sqrt{2}$-metric, see $N_{\sqrt{2}}$ in Equation 3.32

Figure 3.7: The different flood-filling metrics. The green cell is the start, all black cells have a distance < 4.

have to compute the distances to all neighbors. This queue is initialized with our seed cells. As long as there is something in the queue, we will get and remove the first element and iterate though all neighbors of the retrieved cell. For each neighbor n a new cost is computed, but only assigned if n is not blocked or if the distance field already has a smaller distance at this position. If the new cost is assigned, n will be added to the queue.

Algorithm 3.1: The flood filling algorithm

input : A cost field F_c , a list of seed cells S, the neighbor cost matrix N_c
output : The distance field F_d

```
1  F_d ← set all cells to ∞, except cells ∈ S;
2  Q ← a new queue;
3  Q.addAll(S);
4  while not Q.isEmpty() do
5  |  cell ← Q.pop();
6  |  foreach neighbor n of cell do
   |  |  // Negative costs are impassable cells
7  |  |  if F_c[n] ≥ 0 and F_d[cell] + N_c[n] + F_c[n] < F_d[n] then
8  |  |  |  F_d[n] ← F_d[cell] + N_c[n] + F_c[n];
9  |  |  |  Q.push(n);
10 |  |  end
11 |  end
12 end
```

To obtain a distance field that produces Euclidean distances instead of approximations a more sofisticated solver has to be applied. The Fast Marching Method [Set99] and the Fast Iterative Method [JW07] are examples for such a solver.

After the distance map has been computed the derivative of it has to be taken. This derivative is a 2D vector that points in the direction of the seeded cells. For the computation we have to distinguish between 2 cases: the unblocked case where all neighbors of the cell are not blocked and the blocked case where at least one neighbor of the cell is blocked. In the unblocked case the matrices in Equation 3.33 are convoluted with the field. For a discrete derivation a derivation matrix is placed over the cell to derive, all overlapping values are multiplied and the resulting numbers are then added up. In order to obtain the x and the y-component of the derivative we have to this twice. In the following let ∇x be the derivation matrix of the distance map in x-direction and ∇y be the derivation matrix of the distance map in y-direction.

$$\nabla x = \frac{1}{9}\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, \nabla y = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \tag{3.33}$$

In the blocked case we also have to cope with the special case of non-walkable cells. These should not contribute at all to the derivative. A simple solution could be to assign a very high cost to these cells, but this would result in a repelling behavior next to blocked cells as the high value would outweigh all the other distances easily.

We decided to use a greedy approach to approximate the derivative in these special cases. Out of the eight neighbors only walkable cells are taken into consideration and the derivative is approximated by the vector pointing to the lowest value that is greater or equal to zero. See Algorithm 3.2 for a pseudo-code of the algorithm.

So far we managed to create a distance map that is able to find the shortest path to the target areas. In order to find the quickest path, Kretz et al. [KGH+11] took two factors into consideration: the density of the agents at a particular cell and the mean direction agents in a cell are heading. The model for the estimated walking speed $\tilde{w}$ looks like this:

$$\tilde{w}(X) = \max\left(0, g\left(1 + h\frac{v(X) \cdot \nabla F_d(X)}{\overline{v_0}\,|\nabla F_d(X)|}\right)\right) \tag{3.34}$$

$F_d(X)$ is the distance field sampled at the point X, g is a parameter setting the overall strength of the model, h is a free parameter that sets the weight of the penalty based on the facing of an agent, $v(X)$ is the mean direction of all agents occupying the cell at X and $\overline{v_0}$ is the mean desired speed of all agents.

With the help of this model a time field F_t can to be computed using the distance field F_d. F_t contains the estimated travel time for any agent at any point in the domain. We take a slightly different approach from the original paper at this point, because we are using the flood-filling algorithm for the field computations and we also have to add the slope of the terrain into the equation as well.

In Order to compute the cost field F_c for the domain we have to combine all models with each other. The density D of the agents is multiplied by the prior discussed estimated

Algorithm 3.2: Compute the derivative of a field

input : A distance field F_d
output : The derivative of the distance field F'_d

```
1  foreach cell in F_d do
2      valid ← true;
3      foreach neighbor N of cell do
           // Check if the cell is walkable
4          if F_d[N] < 0 then
5              valid ← false;
6          end
7      end
8      if valid then
9          F'_d[cell] ← computeDerivative(F_d,cell);
10     else
11         F'_d[cell] ← vector from cell to min(F_d) at each neighbor of cell;
12     end
13 end
```

walking speed $\tilde{w}$ (see Equation 3.34). As discussed in section 3.2.3 the slope parameter w_s is multiplied with the slope $s(X)$ (see Equation 3.24) at position X and gets added to the cost. Then the flood filling algorithm computes F_t and uses F_c as its cost field as shown in algorithm 3.1. The derivative of F_t is then finally used to compute the direction of the fastest path for every position in the domain.

$$F_c(X) = 1 + D(X) \max\left(0, g\left(1 + h\frac{v(X) \cdot \nabla F_d(X)}{\overline{v_0}\,|\nabla F_d(X)|}\right)\right) + w_s s(X) \tag{3.35}$$

Figure 3.8 shows the green agents moving towards the green target area. The ground visualizes the time field: The values are divided into multiple equally big intervals and each interval is represented by a color. It can be seen that although the top corridor is clearly the shortest path to go from the left to the right room, the time map shows that using the bottom corridor yields a comparable estimated arrival time, because of the congestion at the entrance to the top corridor. In this example several agents are already taking the lower corridor, because they estimated it to be the faster route for them.

3.3.2 Density Field Computation

The density of pedestrians can be computed in different ways. A very simple approach would add one to a cell's density if a pedestrian's center point is inside that cell (Figure 3.9a). One could imagine this method as just counting the number of agents inside a cell. This works well if the cell size is much bigger than an agent, because then the agent's

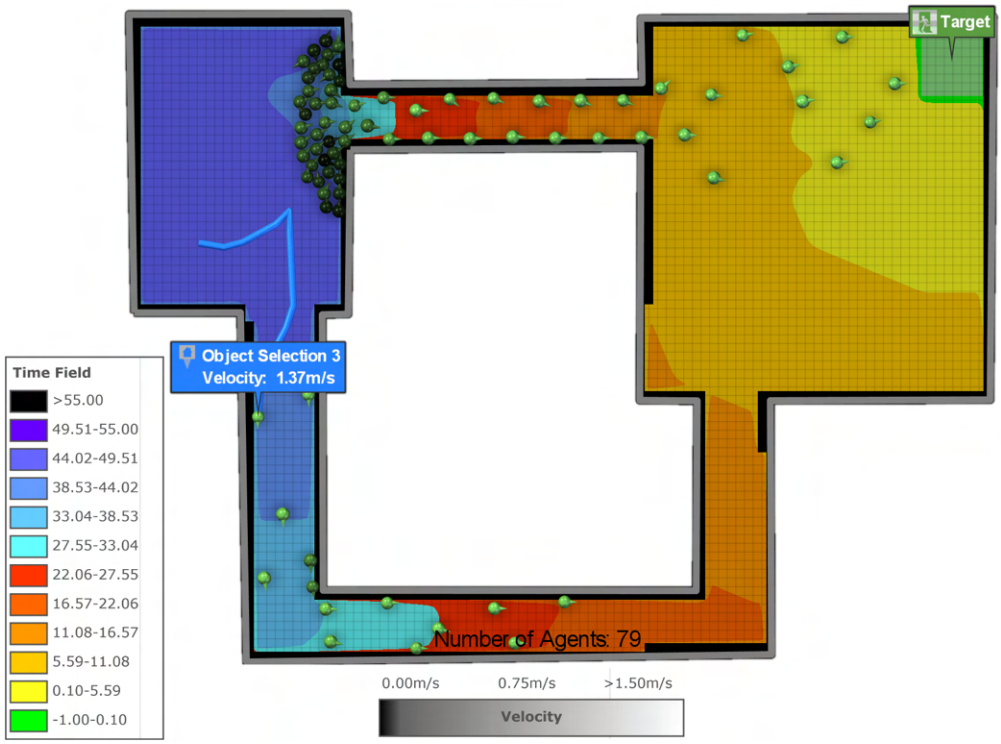
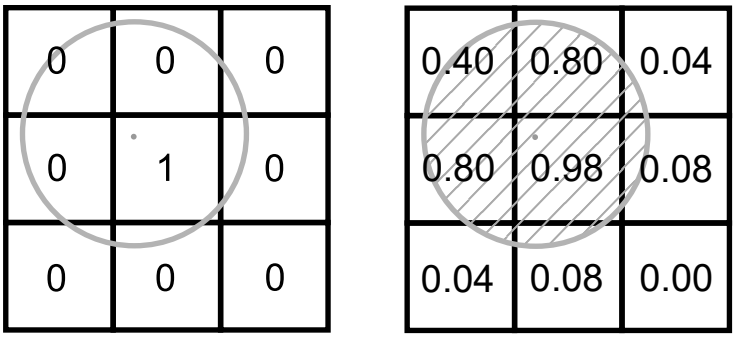


Figure 3.8: The time field of the estimated arrival of the agents. Number of agents is the current number of active agents in the simulation. The path of one agent is traced. It first tried to go through the upper corridor, but then switched to the lower corridor when the congestion formed.

radius is not that important. If it is the other way around, an agent might cover multiple other cells that will not get added one to their density, because the agent's center is not inside them. It seems like an easy fix to just add one to each cell that intersects the agent's radius, but this would create not very representative densities in cases where the agent barely overlaps a cell. In this instance the cell would get added one to its density despite the agent not really overlapping it.

Another approach is computing the overlapping area of the cell and the agent normalized by the area of the cell (Figure 3.9b). This means that each cell contains an overlapping-percentage where a value of < 1 means that some part of the cell might be overlapped by an agent, a value of 1 means that the cell is complete overlapped by an agent and a value > 1 means that the cell is over-populated: it contains more agents than it has room for. In this last case the agents themselves are overlapping each other, which should not, but might happen under very dense conditions. Tests with this kind of density calculation yielded unnatural agent behavior, because the agents were avoiding each other much more than was necessary.

To solve this issue we used a sampling method where the agent's center point is the



(a) The density of a cell is the number of agents with their origin inside a cell

(b) The density of a cell is the areal overlap of the agents with a cell

Figure 3.9: Two density computation methods

center of a Gauss bell. Unfortunately the Gauss bell is not constraint, so the influence of an agent would reach infinitely far which would be bad for performance. So we decided to use a modified Gauss bell that is constraint, so we do not have to update the whole density field when adding the influence of a single agent to it. This modified bell is constraint to a circle at the agent's center point in such a way so the bell reaches zero at the edge of the circle. The circle's radius r is the radius of the agent plus a free influence-variable ρ. Because this circle is related to an agent we will call it the agent's *influence circle*. The standard-deviation σ of the modified Gaussian curve $g(x)$ is chosen so that $r = 3.5\sigma$. Equation 3.36 shows that we also subtract the value at the edge of the circle from the whole equation - in fact moving it down so $g(x)$ becomes 0 at the edge, see Figure 3.10.

$$r = r_A + \rho, \quad \sigma = \frac{r}{3.5}$$

$$g'(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} - \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{3.5^2}{2}}$$

$$g(x) = \begin{cases} g'(x) & \text{if } -r < x < r \\ 0 & \text{otherwise} \end{cases} \tag{3.36}$$

In order to sample $g(x)$ on a cell that is intersecting with the agent's influence-circle a set number of equally distributed sample points (we used nine) are defined and $g(x)$ is evaluated at these points as seen in Figure 3.11. Then the average sampled value is added to the cell's density.

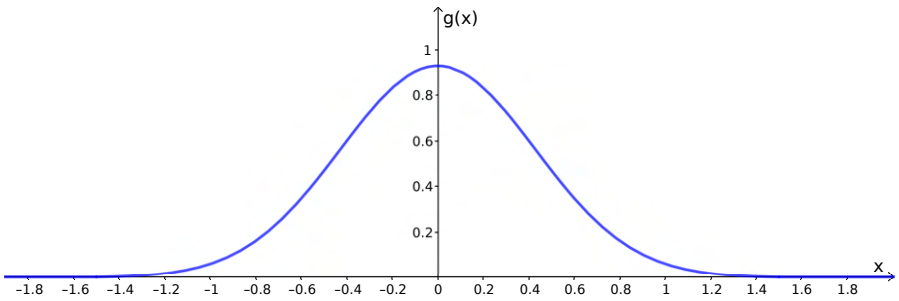


Figure 3.10: The modified Gauss $g(x)$ from Equation 3.36 for $r = 1.5$. This modified Gauss produces only values greater than zero within the interval $]-r; r[$ outside of this interval the function is zero.

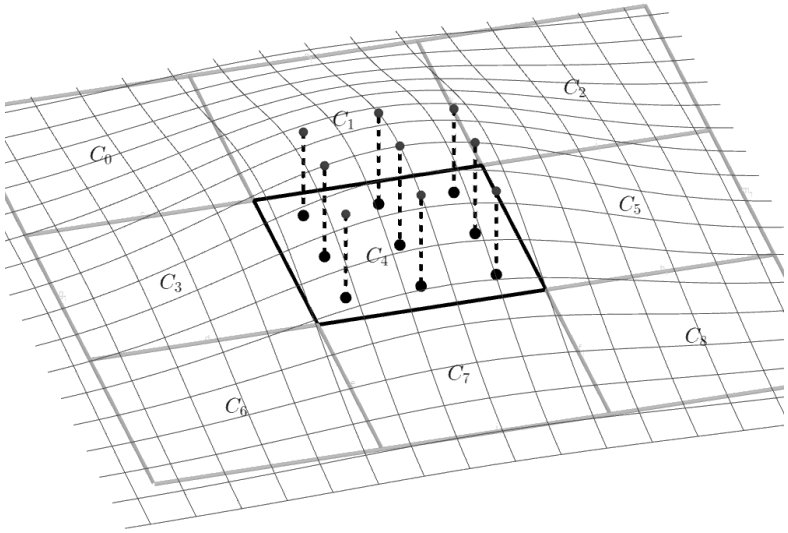


Figure 3.11: C_0 through C_8 are cells of a grid. Over the grid the modified Gauss bell curve $g(x)$ is positioned and the 9 sampling points of cell C_4 are shown.

3.4 Initial States and Boundary Conditions

When creating a simulation the initial state has to be defined. This happens via boundary conditions. The simulation has the following boundary conditions:

- Size of the simulation domain
- Target area for each agent
- Starting position of each agent
- Position and form of walls and other static obstacles in the domain
- All of the free parameters mentioned in the previous sections

In the following we will go through all boundary conditions mentioned above and explain how they can be specified.

First we will take a look at the simulation domain. The simulation domain is an axis-aligned rectangle that defines where agents are able to exist. There should not be an agent outside of the domain, because the domain defines the dimensions and cell-size of all above discussed fields like the density field, distance field and time field. Because of this the dimensions should be chosen as small as possible to minimize computational effort. To prevent agents from being pushed out of the domain, the edges of the domain are treated like walls. Additionally a height-map can be defined for the domain which assigns each point a certain height. The height map supports three modes: flat terrain, procedural terrain and geo-data. Flat terrain produces a height map with height zero at every point. Procedural terrain uses a noise function to generate a bumpy landscape and geo-data mode imports geographical height data from an input-file.

Agents in our simulation need one or more targets. Target are specified as areas. An area can be defined by defining a polygon. A target is given a name and agents will identify their targets by those names. An agent may have multiple targets, in that case the target with the lowest estimated arrival time (calculated via the time map, see Section 3.3.1) is chosen. If an agent arrives at a target, the agent is removed from the simulation. This is the only way agents can be deleted.

When placing agents the target area can be specified. An agent will always try to reach the assigned target area as fast as possible. Each parameter of an agent can be set between a lower and an upper limit. The actual parameter value is then chosen randomly within these

Parameter	**Minimum**	**Maximum**
Maximum Speed v_a^{max}	1.25 m/s	1.5 m/s
Radius r_A	0.18 m	0.2 m

Table 3.2: The default values of the boundary conditions for an agent. The position p_A is defined via one of the methods described in the text. All other values of an agent like v_a or v_A^p are set to zero.

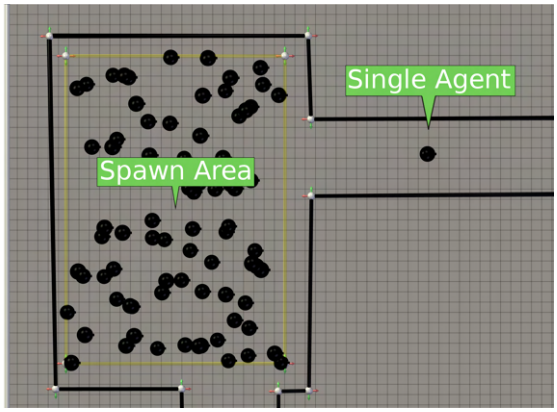


Figure 3.12: Agents spawned in a polygon on the left and a single agent on the right.

limits. The default setting for
an agent can be seen in Table 3.2.

The starting position of an agent is a special case. In order to improve usability we used three methods of defining an agent's position:

- Placing agents individually
- Specify an area
- Create an emitter

When placing agents individually, the position is chosen by the user, but only one agent at a time can be placed. When placing agents using an area, a whole group of agents can be created at once. We will call such an area a *spawn area*. The number of agents created by a spawn area can be defined in two ways: defining the number directly or defining a desired agent density. Either way the agents will then be placed inside the spawn area at random.

Whether placing agents individually or by using a spawn area, the agents will be created in a burst: at a certain point in time these agents will appear. Currently this is only possible at the start of the simulation. Figure 3.12 shows a rectangular area that has spawned a burst of agents as well as a single hand-placed agent.

There is also another method available: using an emitter. An emitter creates agents over time rather than creating them all at once. It will start at a certain point in time and create agents until the simulation ends. The rate at which it creates them can be defined as well as the location. The location can be designated by a point, a line, or an area.

All static obstacles are represented as lines. The user is able to draw them as they please. Obstacles may range from potted plants to walls or fences. It is not possible for agents

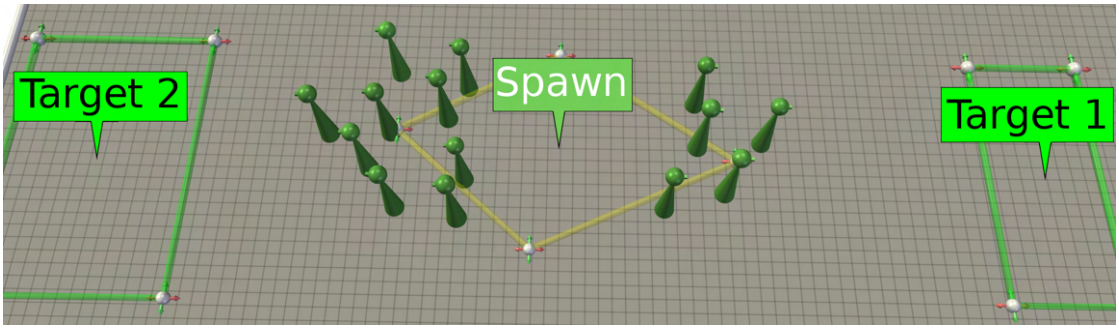


Figure 3.13: A spawn area with two targets assigned to it. The 14 created agents will move to the target with the lowest estimated arrival time.

to surmount these - the only possible way is to move around them. This may not be very realistic as real people can climb over low obstacles or even fences. Especially in an emergency situation, people in panic may resort to such methods.

CHAPTER 4

Visualization and Interactive Simulation Control

In the previous chapter we talked about how we generate simulation data. In this chapter we will talk about how we visualize and interact with that data. We will talk about the used methods, the chosen representations of agents, obstacles etc., and how Visdom [vis] supports the user with its already build-in features.

4.1 Data Visualization

The data generated by the simulation for each simulation step is the following:

- The position and velocity of each agent
- The current density field F_d generated by the tactical layer
- The estimation of the quickest path to the target area, generated by the tactical layer

We will discuss how to visualize all of the above in the following sub-sections.

4.1.1 Agent Visualization

We propose to represent agents by simple glyphs instead of life-like models of humans, because we want to reduce visual clutter and focus on the data we want to visualize: the position, the speed, and the heading of an agent. Such a glyph looks like a board-game piece. A sphere sits on top of a cone to form a head and an additional cone as a nose so the user can identify the walking direction of an agent even if viewing a still image.

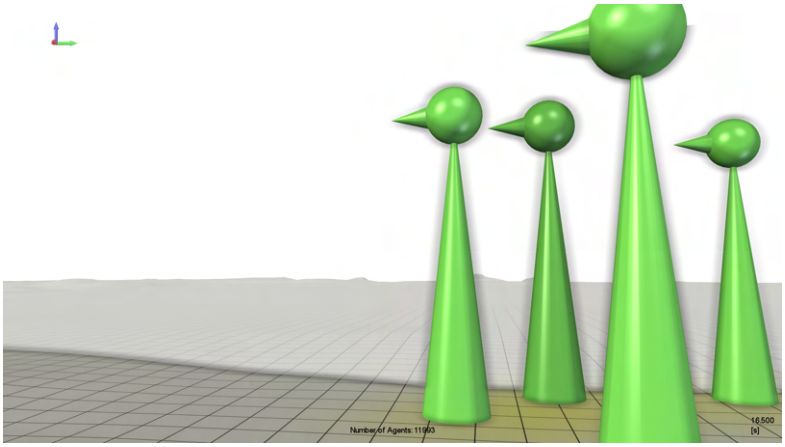


Figure 4.1: A closeup of agents. An agent consists of a cone, a sphere, and another cone as a 'nose'

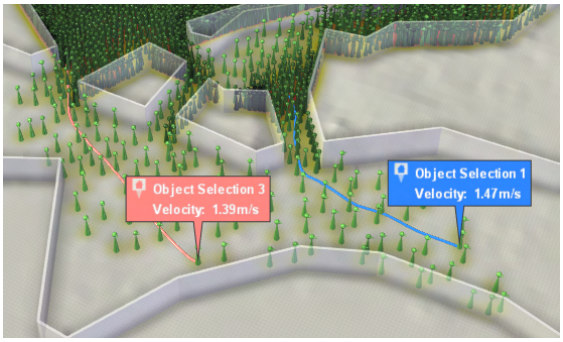
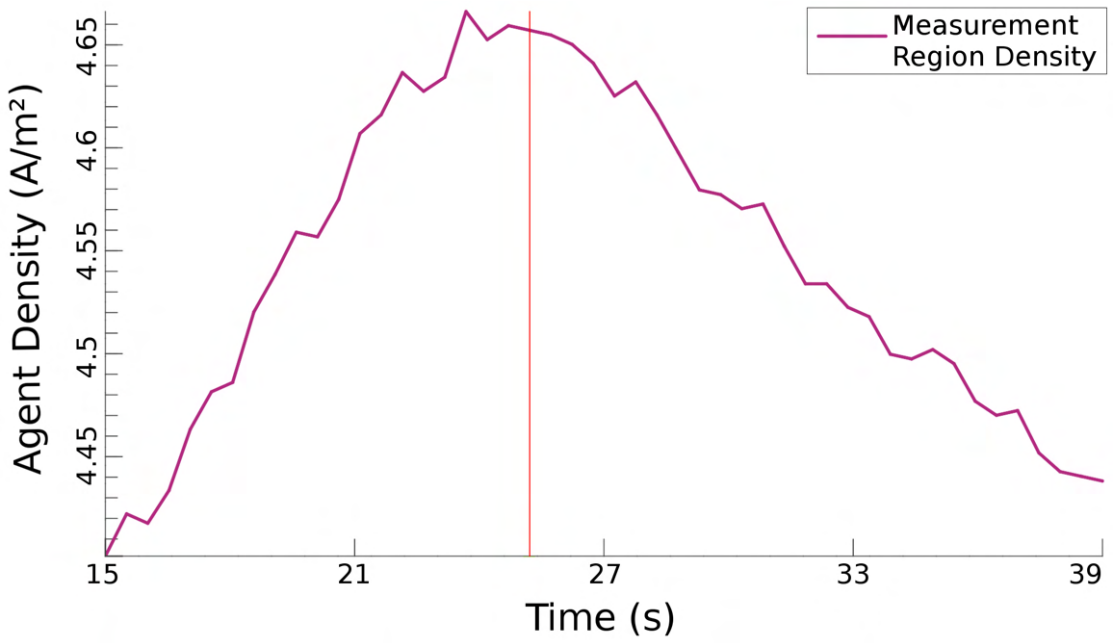


Figure 4.2: Two selected agents traced by two lines with the velocity displayed in an information box above them.

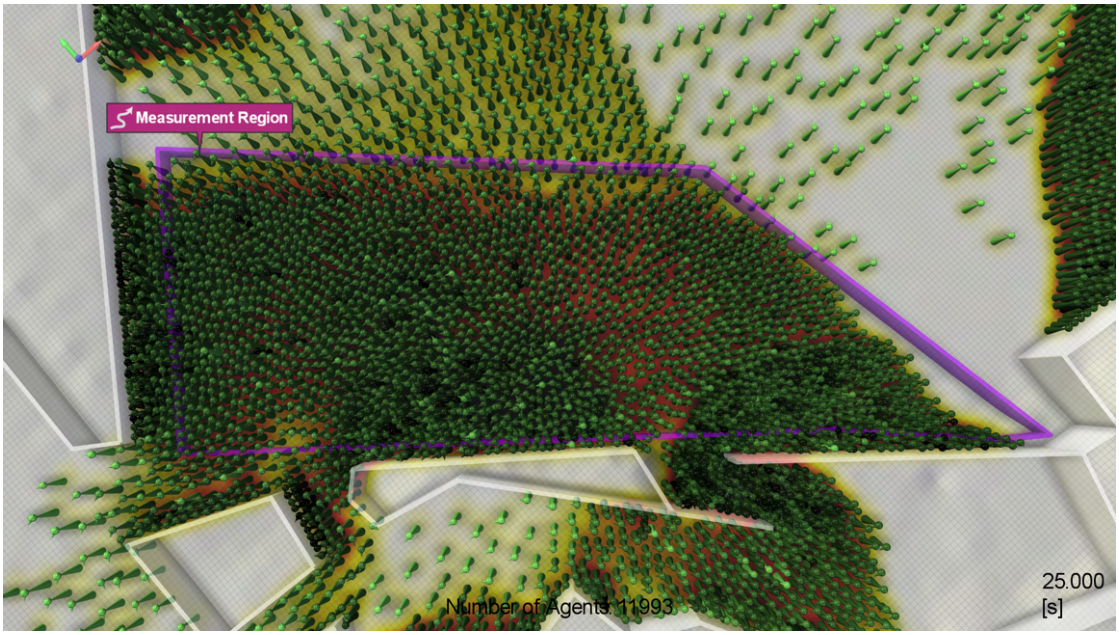
The radius of the cone and sphere is the radius of the simulated agent and the height is the radius multiplied by a constant, so agents with smaller radius are also smaller in height. A close up of an agent can be seen in Figure 4.1. Another variable encoded in this glyph is the current speed. It is represented by the brightness of the agent's color where a speed of zero is black and the maximum speed is the full color brightness.

Individual agents may be selected (see Figure 4.2). If this happens the path of the agent so far is drawn as a line on the screen and additional information, like its current velocity, is displayed. The speed of the agent is also plotted on a time vs velocity graph.

There is also a plot available for agents in a specific area. In this case the plot can either show the current number of agents in an area or the agent density inside that area. Figure 4.3 shows the agent density of a region in front of a series of narrow corridors.



(a) A plot of the agent density over time



(b) The scene used to take the measurements

Figure 4.3: Measuring the agent density in a certain region

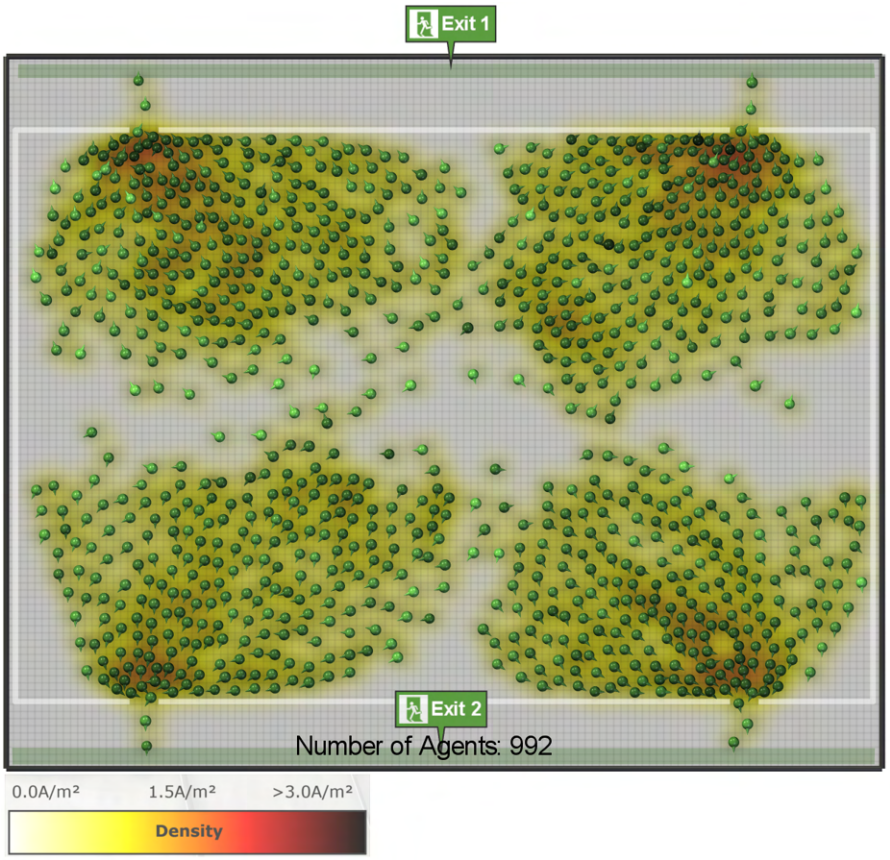
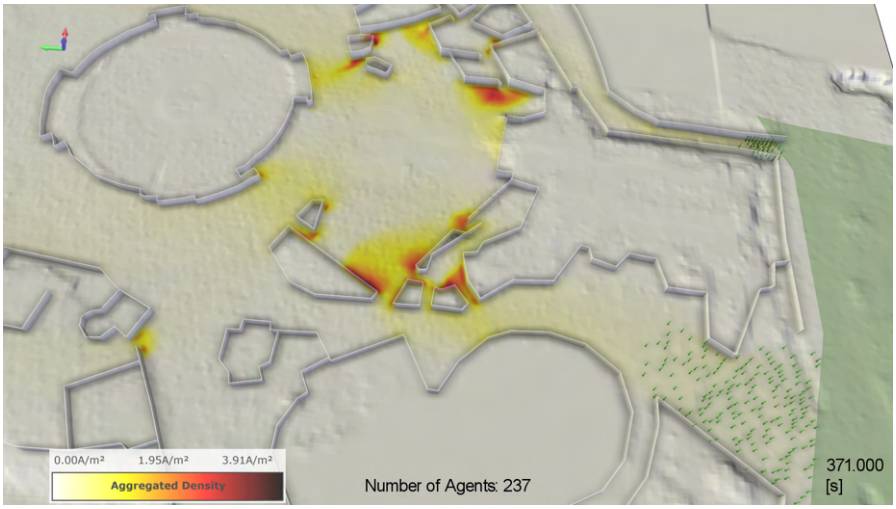


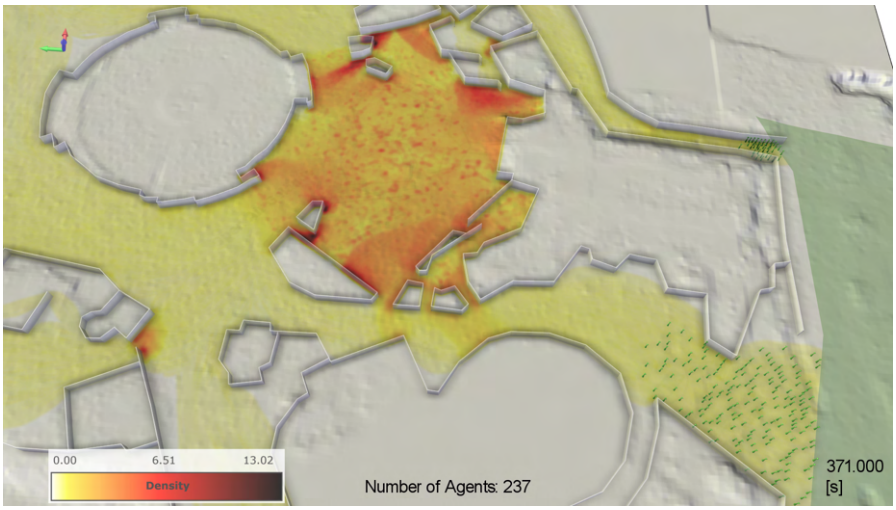
Figure 4.4: The density field of the agents can be seen on the floor. White represents a density of zero agents per square meter (A/m^2), yellow a density of one A/m^2, red a density of two A/m^2, and black a density greater or equal to three.

4.1.2 Agent Density Visualization

The density of agents can also be visualized on the domain itself. In Section 3.3.2 we described how the density of a group of agents can be measured, now we want to present this data to the user. As shown by Fanini1 and Calori [FC14], and Handel et al. [HGPA15] this can be done by the use of the floor the agents are walking on. Fanini1 and Calori rendered a 3D-graph of the density by creating a wire mesh that encodes the density of the agents in the elevation of the mesh from the floor and displaying it on top of the streets. Handel et al. took a simpler approach: They created a grid and colored the cells like in a heat map using the number of agents in that cell as data (see Figure 4.4). We did something similar to that with a few exceptions: First their heat map was normalized to the current maximum of agents in a single cell and second their cells were considerably bigger than ours. Our heat map is not normalized and turns from white, at zero agents per square meter ($0A/m^2$), over yellow ($1\,A/m^2$) and red ($2\,A/m^2$) to black ($\geq 3\,A/m^2$). This has the advantage that the user is able to spot dangerous densities of agents more



(a) The average of all density fields up to now.



(b) The maximum of all density fields up to now.

Figure 4.5: Density Aggregations

easily as the scale never changes. A disadvantage of our approach is that it is no longer possible to spot the cell with the currently highest density.

In order to give the user an overview of what has happened during the simulation, we introduced aggregations of the density field. These aggregations take the density fields from each time step up to now and aggregate the density values for each cell through time. We propose three types of aggregation:

- Average of the densities

- Square root of two of the average of the densities
- Maximum of the densities

The first aggregation is the average of the density for every cell up to the current point in time (see Figure 4.5a). With this type of aggregation areas of high densities over a long period of time are clearly visible. Areas that saw a high density over a short period of time are less visible, because the high densities get averaged out over time. This allows for spotting especially dangerous regions where high agent densities persisted over a long period of time.

A variant on the previous measurement is the square root of the average. Taking the square root of the average boosts lower densities and makes them more visible. This is useful if the user wants to keep track of both big and small congestion.

The third option is displaying the maximum of the densities up to the current point in time (see Figure 4.5b). In our system this is called "Agent Tracing", because only areas that where never visited by agents show up as withe, all other areas yellow to black. In contrast to that the average method might have areas that are white, although agents walked over them. This happens because low density values can be averaged out if there is a long enough period with density zero. The advantage of the maximum aggregation is that the user can still read the original density values - the aggregation just shows the maximum of all time. When using the average aggregation the resulting numbers are always lower than the maximum of the same region and this might make high density areas appear less dense.

4.1.3 Agent Flow Visualization

In order to analyze the movement of all agents, selecting all of them at once is not very useful, because the scene becomes hard to read due to visual clutter. All the windows and path-lines make it really hard to interpret the overall flow. Cornel et al. [CKB+16] solved this. They propose an algorithm to automatically generate adaptive flow maps out of a given data set. The first step is to create a zonation of the domain. This splits the domain into semantically relevant zones. In 2D an automated zonation is difficult and it might have to be done manually. The next step is to count the flow of agents from one zone to another. With this data a directional graph can be computed where zones are nodes and adjacent zones have weighted edges connecting them. This graph is then visualized as points and arrows. The thickness of the arrows represents the flow between the points which are also scaled to encode the number of agents passing through them. An implementation of this algorithm has been integrated with our simulation to provide additional information about agent movement.

4.2 Visualizations and Interactions in Visdom

The plugin we developed can be used within the simulation platform Visdom [vis]. In the following we will talk about some general functions of Visdom and how we composed them in order to create a work flow that helps the user achieving their desired goal.

4.2.1 Graphical User Interface

The user has to interact with the system via the graphical user interface (GUI). The application has two modes: an edit mode and an analysis mode which is the default one. In order to switch to the edit mode the user has to check the "edit mode" check box - only in the edit mode it is possible to edit the walls or add new agents. If in edit mode the floor visualization changes from the density field to a custom image. This image, e.g., a floorplan or a map, can be used as a guide for the walls. Alternatively a shape file can be used to generate the walls.

Figure 4.6 shows the layout of the application: The user has four windows (some containing tabs) at his disposal. The biggest central window is the scene view. This view shows a 3D rendering of the simulated world at a certain point in time to the user. It also has a legend on the lower left for the density map, tools for creating new walls/agents etc. on the lower right and camera controls and edit tools on the upper left.

The window at the bottom shows the time line tool that controls the current time via the red cursor. It also offers controls for changing the current branch of the simulation or creating a new one.

At the top right the settings window contains all important settings regarding the simulation model or the terrain. Right below it the plotting window is displayed that has three tabs: an overview tab summarizing some important statistics like the velocity of the agents and two plotting tabs showing plots of the selected agents and measurement areas.

Visdom [vis] also has some features and techniques that help the user to make sense of the generated data and interact with them. It also offers a way to model uncertainty. This is explained in a more in-depth fashion in papers by Ribičić et al. [RWF+13][**?**]. We give a brief overview of the system here in the context of pedestrian simulation.

A key concept is the usage of tracks and ensembles. A track is a time line consisting of multiple time steps or frames that represent the state of the simulation at a certain point in time. A cursor shows where the simulation is in time and the cursor can also be used to change the time in the simulation. An ensembles consists of multiple tracks that all represent a certain type of uncertainty in the simulation. For example, the number of agents might be different in each track of an ensemble or a simulation parameter might change. The user can then switch between the different tracks by simply clicking on them. This makes it easy to compare different scenarios with each other.

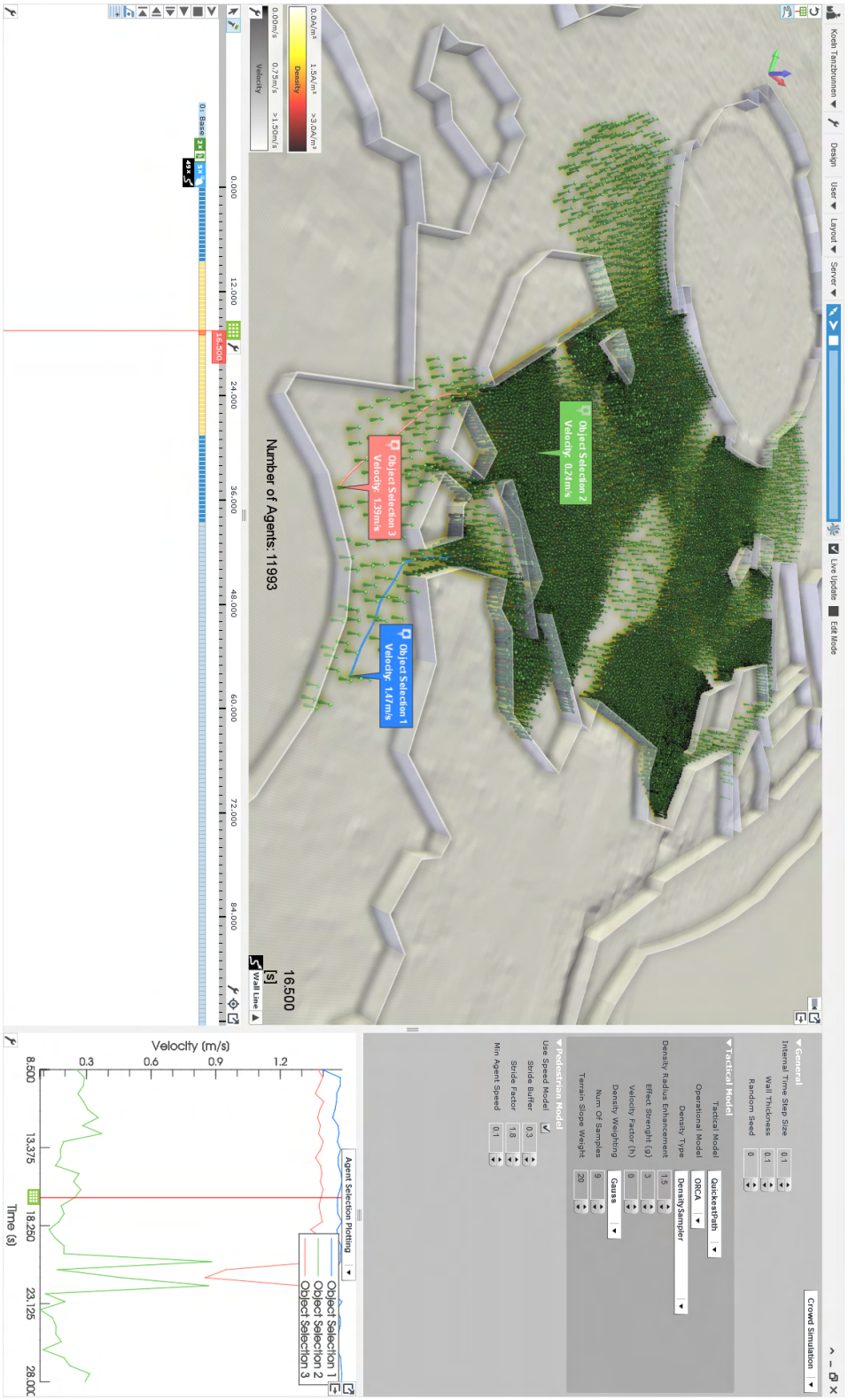


Figure 4.6: The general layout of the GUI

It is also possible to make changes to the boundary conditions of the simulation after a certain amount of simulated time has already passed, e.g., add an additional wall to block a corridor, and view the consequences of them. This is done via branches. Branches originate from a certain track at a certain point in time and create a new track starting at that same moment in time. The new track inherits all settings and objects from the parent track, so nothing differs at first. Then the user can make some changes to the new track. If a change needs to be made to an already existing object, the object first has to be copied to the current track using the "Copy to Track"-tool. The simulation can then be computed for the branch and the effects can be compared to the original scenario.

As discussed in the last section, methods from information visualization were leveraged: linking and brushing. Selecting an agent is linked to the diagram window and tracks may be brushed for temporal selections. It is also possible to brush multiple tracks at once - the diagram in the diagram window will plot lines for each track.

4.2.2 Setting up a scenario

When setting up a scenario we have to distinguish between a synthetic scenario or a real-world scenario. In a synthetic scenario the user can either use a flat terrain or a synthetic heightmap for the ground. In a real-world scenario the user might want to import geographical data to generate a heightmap out of that. In this case a map of the region is shown on top of the terrain and the user gets the option to import building boundaries to automatically generate obstacles so people might walk the streets. All settings can be made in the settings window that is docked to the side of the scene view. In a second tab of this window, parameters of the simulation may be adjusted.

It is then possible to adjust the dimensions of the simulation domain. Now agents, targets, and obstacles may be added to the simulation. In this example the walls are drawn first. Walls can either be drawn as free-form curves or as polygon lines. If using either of these tools, the curve can be selected and exact coordinates may be entered for the control points. Additionally the edit-tool can be used to move control points and to edit them on the fly. The free-form curve also offers handles to adjust the tangent at a control point. As discussed before, in order to help the user with creating real world-scenarios overlay images can be imported and are displayed instead of the floor color/map in edit mode. The user can then trace lines of obstacles (e.g., walls) to set up the scenario. Doors can either be modeled as gaps in the walls or with the door tool that cuts out a piece of the wall to make room for the agents.

All controls for drawing lines are also applicable to polygons. Also as discussed in the previous chapter agents may be created by placing them individually, using a spawn area where they will instantly spawn in one burst or by placing an emitter that will add agents to the simulation over time. Targets are also created by drawing polygons and assigning names to inform the agents where to go. Each group of agents may also be colored, so they can be distinguished.

CHAPTER 5

Implementation

5.1 Visdom

The software we were using as a framework is called Visdom [vis] and can be understood as a simulation and visualization tool. This framework had not been implemented during the course, it was re-used. In this section we will explain important concepts of Visdom and in Section 5.2 we will talk about the actual implementation that had been done during this thesis.

In Visdom each scene that is created can have very different capabilities. A scene contains not only the setup of the boundary conditions, like for example where houses or pedestrians are placed, it also contains a script that controls what is simulated in the first place and how it is displayed. So although Visdom is written in C++ and our plugin is written in C++ as well, the script contained in the scene is written in a graphical programming language called *Data Flow Diagram*.

5.1.1 Data Flow Diagram - A Graphical Programming Language

The Data Flow Diagram is a graphical programming language. A key concept of this programming language is the *data flow*. The data flow describes where data is transferred from and to. For example if we have a command that creates a mesh and a command to render a mesh we can set up the data flow between these two commands, so the mesh gets rendered.

This data flow is visualized by a directed graph where every node is a command and edges between the nodes control the data flow. Each node in this graph has incoming edges and outgoing edges. Incoming edges (input) are located on the left side of the node and outgoing ones (output) are located on the right. Each node additionally has settings that may alter the behavior of a node. Nodes offer connection points on the left and right side of the node where edges can start or end. See Figure 5.1 for an example

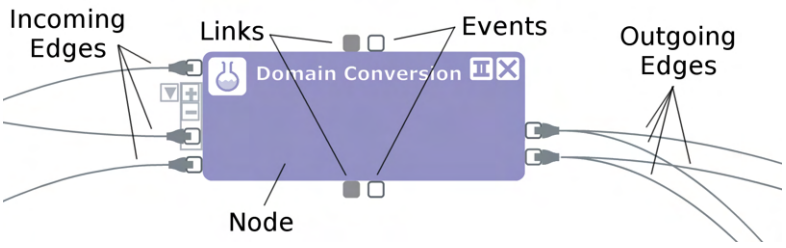


Figure 5.1: A single node

of such a node. The basic idea is that each node creates and/or consumes data that is passed along the edges between two nodes. Not all inputs of a node need to be connected to an output of another node - some inputs are optional and can be left unconnected.

The graph does not model a control flow - this means that the programmer cannot define the order of execution of commands (nodes) explicilty. The system starts with nodes that have no inputs and continues with executing commands that have all input data until all nodes have been executed. A more in-depth explantion of the system can be found in the publication of Benjamin Schindler et al. [SWR+12]. In large graphs rendering all edges might lead to situations where the programmer has a hard time identifying which nodes are connected to each other. For a better visibility in large graphs edges can also be hidden. In this case an icon at the connection point indicates this fact and the edge reappears as long as the mouse cursor is hovering over the connection point.

Figure 5.2 shows an example of a simple data flow. This simple data flow will just produce an immobile, hovering agent in an otherwise empty scene. On the far left three nodes without inputs can be seen - they are called producer nodes. Each of these nodes creates a mesh of a certain color. Because an agent consists of three basic shapes (2 cones and one sphere, see section 4.1.1) we need three producer nodes for the primitive meshes. The mesh of each producer is then passed on to the "Append Meshes" node. This node then merges the three meshes into a single one and passes the result to the "OpenGL" node that finally renders the scene. The control flow is implicit, because a node always needs the data of all outputs of other nodes connected to this node's inputs in order to process. So the "OpenGL" node requests the "Append Meshes" node to deliver the data, which in turn has to request the data of all three producers. As the "Append Meshes" node does not care, which primitive it will get first, the three producers may work in parallel. When all of them have delivered, the data flows forward until it reaches the "OpenGL" node, which consumes it and renders the scene.

The language can be extended by new nodes by writing a node definition and a new class in C++. The node definition is a configuration file that governs the look, in puts, outputs and options of a node. The C++ class defines the behavior of the node. That means the class reads the node input and configuration data, then does some custom computations and finally sets the node outputs.

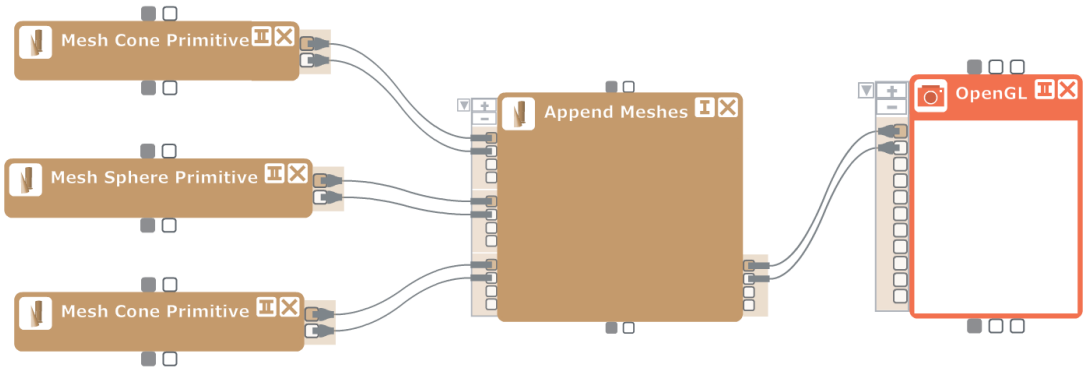


Figure 5.2: A simple data flow

5.1.2 Data Structures

Visdom provides multiple data structures that can be passed in a Data Flow Diagram. The most important ones can be seen in the Unified Modeling Language (UML) class diagram [BJR99] in Figure 5.3 and will be described in the following. Some methods and parent classes are not included for the sake of clarity.

Eigen

An important external library that provides data structures is Eigen [eig]. It contains classes and functions for working with vectors and matrices. All referenced vector classes such as "Vector2d" are part of the Eigen library.The class "std::vector" is a standard library class for lists and is not part of Eigen.

AbstractData

AbstractData is the base class for all data structures that can be passed from one node to another in a Data Flow Diagram. it offers methods for caching the data, getting/setting meta data and cloning.

GPUVector<T>

The GPUVector can be used instead of a std::vector. It offers methods for transferring the data to the GPU for easy General Purpose Computation on the Graphics Processing Unit (GPGPU), but it can also be used on the CPU. Because it extends AbstractData, it can be used to pass a collection of values between nodes in a Data Flow Diagram. The template parameter T is the type of the contained data.

GridDomain<Dimension>

A GridDomain defines a regular grid in the world of a certain size with a certain cell size. The GridDomain object itself does not contain any data besides the domain definition. The actual data is then stored in a separate GPUVector (data vector) of the size of the cell count of the domain. Each cell is then linked to an index in the vector called domain index. The data vector is always one dimensional regardless of the dimension of the domain. This separation of the data vector from the GridDomain containing the domain definition has the advantage of being memory-efficient. E.g., if there are two fields, the density field and the time field, only one GridDomain object is needed to store the domain definition and two data vectors for the actual data.

The domain offers methods for converting positions into domain indices (and the inverse), intersection tests and checks if the position is still in bounds. The template parameter Dimension is an integer that defines the number of dimensions of the domain. For better readability the type GridDomain2D has been defined as GridDomain<2>.

Labels

Labels is a container class for a list of the type of Label. Labels extends AbstractData and can be used to pass data between nodes in a Data Flow Diagram while Label does not. The label object contains the actual data: position, icon, and text that should be displayed in the scene to annotate objects.

Lines

Lines is a list of positions paired with IDs. Each pair of position and ID is called a vertex. The Lines data structure may hold multiple lines where each individual line has its own ID. Closed lines are represented by appending a copy of the first vertex to the end of the list.

MeshInstances

The MeshInstances class stores data about different instances of the same mesh. Instanced rendering (hardware accelerated rendering of the same model in different positions, orientations, and scale) is typically used in tandem with this data structure. Each instance has a transformation that defines its position, rotation, and scale in the world as well as an id, a group, and additional optional values.

Polygons

A Polygon consists of a list of distinct positions (vertices) of the corners of the polygon, a list of vertex indices that defines the edges of the polygon and an id. The data is available both as a list of Polygon2D classes and multiple lists of values (struct of lists).

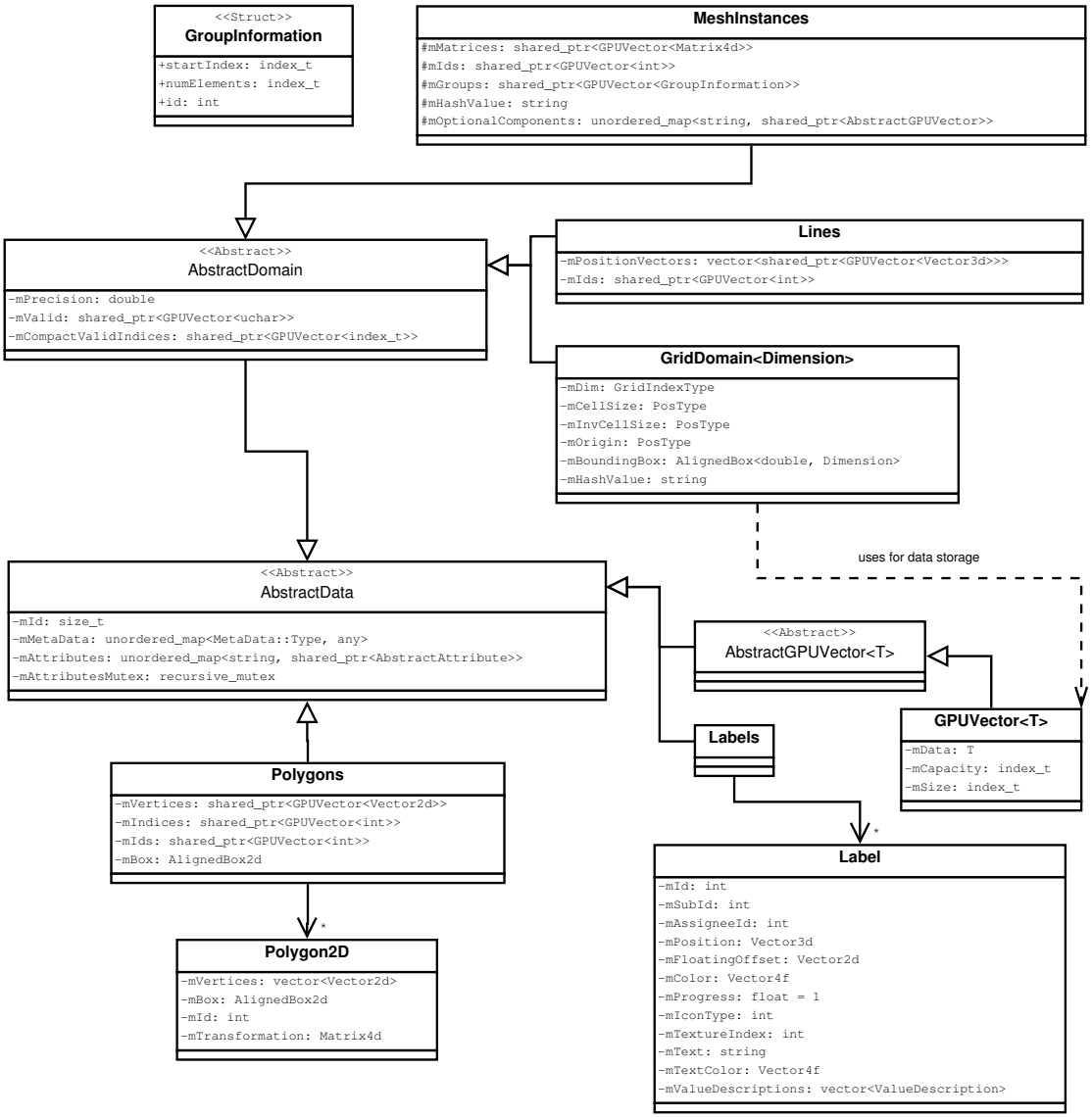


Figure 5.3: The class diagram of Visdom's most important data structures

5.2 The Pedestrian Simulation Plugin

The pedestrian simulation plugin was created for this thesis and contains the Crowd Simulation Node. This is a custom node for the Data Flow Diagram and handles the simulation of the pedestrians. The behavior of a node is determined by a C++ class. Normally nodes are not affected by the passing of simulated time, but when the node class extends *AbstractSimulationNode* then the node has the ability to change its output based on the simulated time. This is also what we have done with our plugin: we created a *CrowdSimulationNode* class which extends the *AbstractSimulationNode*. We will first give an overview of how a new time step is computed and then we will go into more detail.

The implementation of the pedestrian models discussed in chapter 3 is implemented in the *CrowdSimulationExecutor*. The *CrowdSimulationNode* creates an instance of the *CrowdSimulationExecutor* and uses it to calculate the next time step in the simulation. In order to compute the next simulation state we need the boundary conditions of the simulation. There are two types of boundary conditions: dynamic and static. Dynamic boundary conditions are changed by the simulation itself. Static boundary conditions do not change during the simulation unless the user changes them.

There is only one dynamic boundary condition and that is the list of agents. Each time step the agents of the last step will be the input to the current step. All other inputs to the simulation node are treated as static. Because static boundary conditions rarely change, additional data, computed from these inputs, can be cached and reused for future time steps.

After the inputs are handled the next simulation step can be computed. In a simulation step first the agents get enriched with additional data derived from the current state, then the tactical layer is executed. The output of the tactical layer is then taken as new input to the operational layer which computes the new positions of all agents. The implementation of each layer of the simulation (i.e., tactical and operational layer) is hidden behind an interface for better extensibility. At the end agents that have reached their target are removed from the simulation.

Now we will go into more detail regarding the inputs/outputs of the node, how the inputs are processed and how the simulation step is computed in detail.

5.2.1 Inputs and Outputs of the Crowd Simulation Node

When using the Data Flow Diagram, the pedestrian simulation is represented by a node. A description of this node can be seen in Figure 5.4. All inputs except the *emitters* are required for the node to function properly. The *gridDomain2D* and the *terrain* are both required to place the agents onto the floor they are walking on. GridDomains are explained in Section 5.1.2, *terrain* is a list of heights that create when combined with the GridDomain a height map. The *costField* is used in the tactical layer of the simulation to influence the path finding of the agents. This *costField* contains a value for each cell of

Figure 5.4: The inputs and outputs of the crowd simulation node

the simulation domain that will be added to the cost field F_c. This can be used to steer the path finding algorithm away from certain areas. Negative values are not allowed, because they are used for impassible fields.

The input *lines* expects a Lines object that contains all walls/obstacles in the scene. *goalPolygon* expects a Polygons-object with all target polygons that the agents have to reach. *agents* expects a MeshInstances object with all agents set up for the initial state of the simulation. The optional components required are listed in Table 5.1 under *inputs*. The optional input *emitters* expects one or more instances of type Emitters. Emitters are used to generate Agents throughout the simulation and also have to have their optional components set.

The output *agents* is the current state of the agents in the simulation. All optional components are guaranteed to be set - inputs and outputs. The *timeField* contains the estimated arrival times for each cell in the domain and is used in the tactical layer to compute the path of the agents. The *timeField* is mainly used as a debug output. The *densityField* is the field of agent densities in $agents/m^2$ that was used by the tactical layer. Because the tactical layer computes the density field before agents are moved, the density field outputted is computed on data of the previous frame. As the time steps are small the difference between the density fields of two consecutive frames is negligible. The *labels* output offers a label with information about each agent in the simulation. It contains the id, velocity, radius, and preferred speed of the agent.

5.2.2 Node Input Processing

The inputs are passed along to the C++ class *CrowdSimulationExecutor*. An UML class diagram of the executor can be seen in Figure 5.5. This diagram shows that the executor inherits from the *AbstractManipulatorSimulationExecutor* and implements the *IMeshInstanceEmissionAcceptor* interface. The *AbstractManipulatorSimulationExecutor* provides the basic skeleton of a simulation executor: hooks for state saving, state retrieval, setting changes, and doing the simulation. The interface is needed for emitter support.

When processing node inputs, static inputs are just taken as they are and set in the classes that need them. For example, the input *lines* contains all lines representing walls and will be set on the interface of the operational layer. The dynamic input *agents* is more sophisticated to process, because it has to be converted from *MeshInstances* to a

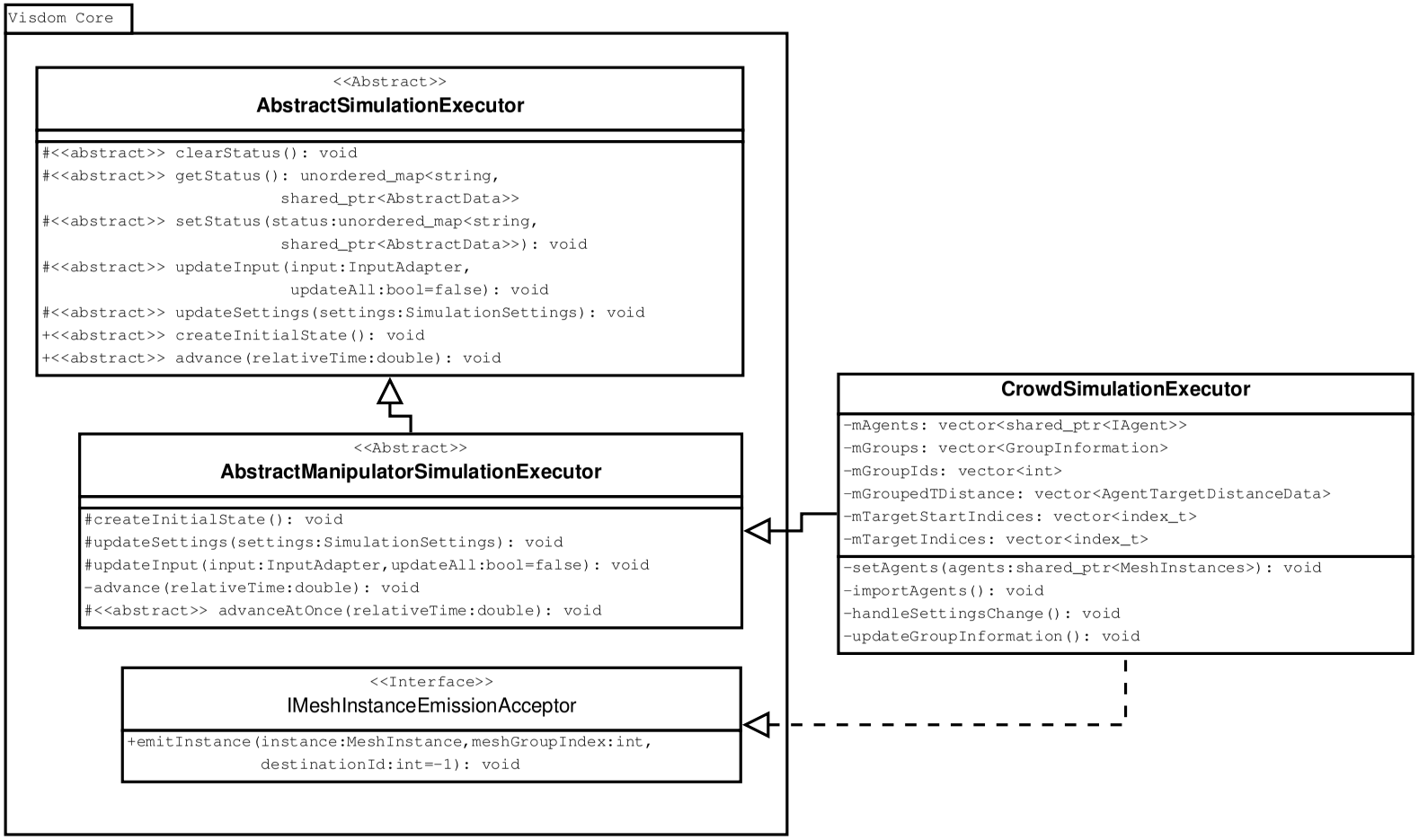


Figure 5.5: The class diagram of the simulation executor

Optional Component	Description
Inputs	
MAX_SPEEDS	The maximum speeds of all agents
RADIUS	The radii of all circles representing the agents
COLORS	The colors of all agents
TARGET_START_INDICES	The start indices of target groups (agents with the same targets) in the TARGET_INDICES list
TARGET_INDICES	A list of indices of polygongs in the goalPolygon object
Outputs	
PREF_SPEEDS	The preferred speeds of agents according to the pedestrian model as a list of scalars
PREF_VELOCITIES	The preferred velocities of agents according to the tactical layer of the simulation as a list of 2D vectors
VELOCITIES	The current velocities of agents as a list of 2D vectors

Table 5.1: The optional components of the MeshInstances object containing the agents

list of *IAgent*s. *AgentData* implements the *IAgent* interface (see Figure 5.6). This enables the usage of software patterns such as Decorators, Adapters and more. Besides methods for getting and setting data, there are also methods for writing the state of an agent to a MeshInstances object or copy an agent entirely. A constructor allows an easy conversion from the MeshInstance object to the AgentData.

Agents with the same targets are bundled up into a group. This grouping is important for the tactical layer, because most computations in the tactical layer can be shared between agents with the same targets. For each group a distance and time map has to be computed. This is a resource intense operation so it is beneficial to have as few groups as possible.

When the agents change, these groups have to be validated, because there might be distinct groups with the same targets that need to be merged. These duplicate groups can occur, because each newly generated agent is automatically placed in its own group. When this happens the grouping has to be updated. There are three cases where the agents might change and the grouping might be invalid:

- The user switches branches in the time line
- The user changes the boundary conditions
- An agent is emitted by an emitter

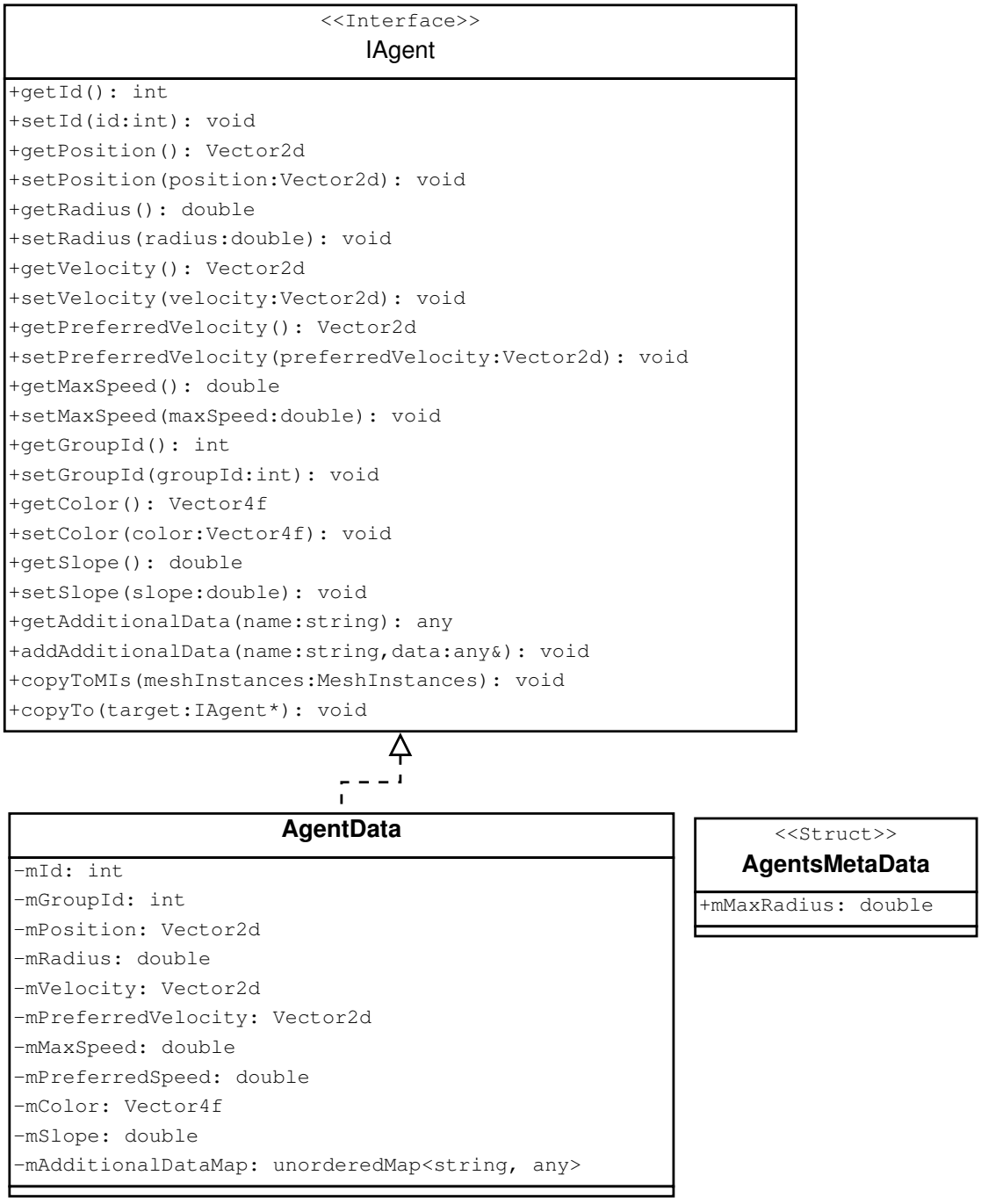
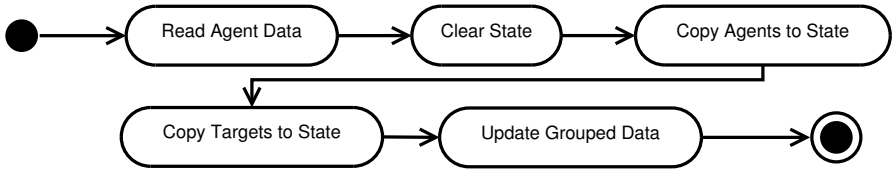
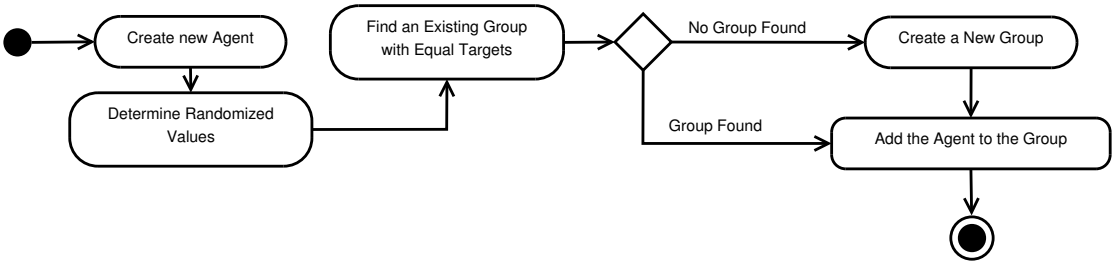


Figure 5.6: The class diagram of an agent



(a) Set agent state

(b) Update on agent boundary conditions



(c) Integrate an emitted agent

Figure 5.7: Activity diagrams describing agent creation and import

When the user switches branches, a state is set. In this case the groups do not need to be validated, because they were already validated, as the state has been a valid state before. So the only task left is copying the agents and targets to the state (Figure 5.7a).

When boundary conditions change, e.g., new agents get added to the world (see Figure 5.7b), the grouping needs to be reevaluated. Here it is possible that there are more groups than needed. The first action is to look for groups with identical target sets and merge these groups. The way in which the grouping is realized may demand a re-ordering of agents. Each group is defined by a start- and an end-index in the agent list. So if two groups that are not stored back-to-back, the second group has to be moved to the end of the first group and the end-index has to be updated to include the new members. Other groups that are behind the modified group also change, because the indices of their members change. After all possible merges are finished, the agents are copied to the state.

Agents can also be created (emitted) by emitters. Emitters are created in the EmitterExecutor and then passed along via the data flow to a class that accepts emitted instances. The base class is the AbstractEmitter, the derivatives mainly define the shape of the emitting area. The emitter shape can be a polygon, a line, or a point for example.

If an agent is emitted by an emitter, the executor expects all required fields to be set, so

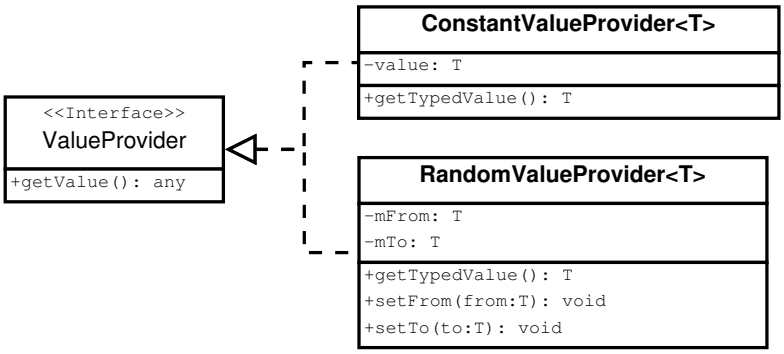


Figure 5.8: The class diagram of the value providers

it can create a *MeshInstance* representing the agent. This is a problem for our agents, because we have some values that should vary between emitted agents, like the radius for example. In Order to enable the emitter to create randomized properties for agents a new interface is introduced. The ValueProvider is that interface and it has exactly one method for returning a value (see Figure 5.8). Two implementations are available: the first one just returns a constant value. This is used for the color of an agent as it is defined per emitter. The other implementation is the RandomValueProvider, which returns a pseudo random number between the defined boundaries. This is used for agent specific values like the radius and the maximum speed.

Figure 5.7c shows the activity diagram for the creation and integration of an agent into the simulation. After the creation it is determined if there is already a group that has the same targets as the emitted instance. If this is the case, the new instance is assigned to the existing group, in the other case a new group is created.

Until now we discussed the three ways agents may be added or removed from the simulation. As discussed before, if agents are added or removed the grouping might change and the cached data for the groups has to be recomputed. This cached data contains the distance field and its derivative for each group. The class that stored the cached data is called *AgentTargetDistanceData*. It is updated by calling the static *updateGroupedData* method. This method computes the distance field and its derivative for each group in parallel. A diagram of this class can be seen in Figure 5.9. This data is used in the tactical layer to compute the preferred velocities of the agents.

Now all inputs are set and the simulation is ready to compute the next simulation state. Each layer of the simulation is defined by an interface (see Figure 5.11), so other models may be implemented besides the existing ones. In order to compute the next simulation state first the slope of the floor along the agent's velocity vector has to be computed, so this information can be used in the simulation layers. Also the biggest radius of all gents is computed and stored in the *AgentsMetaData* object. Then the tactical layer is executed and its result is passed into the operational layer which moves the agents. At the end all agents overlapping with one of their targets are removed from the simulation.

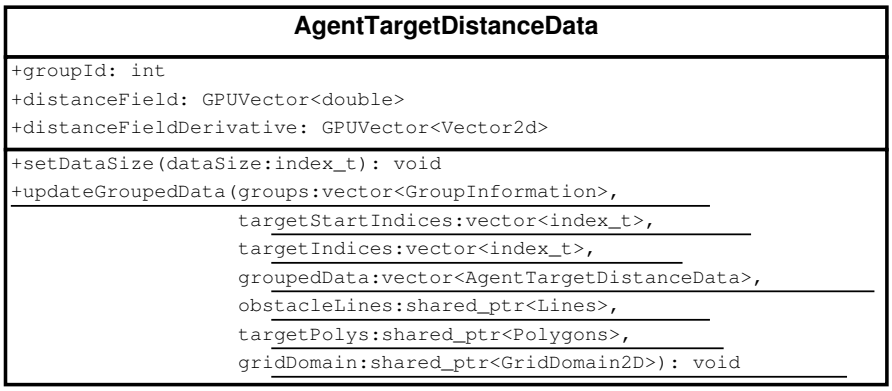


Figure 5.9: The class diagram of the grouped distance data

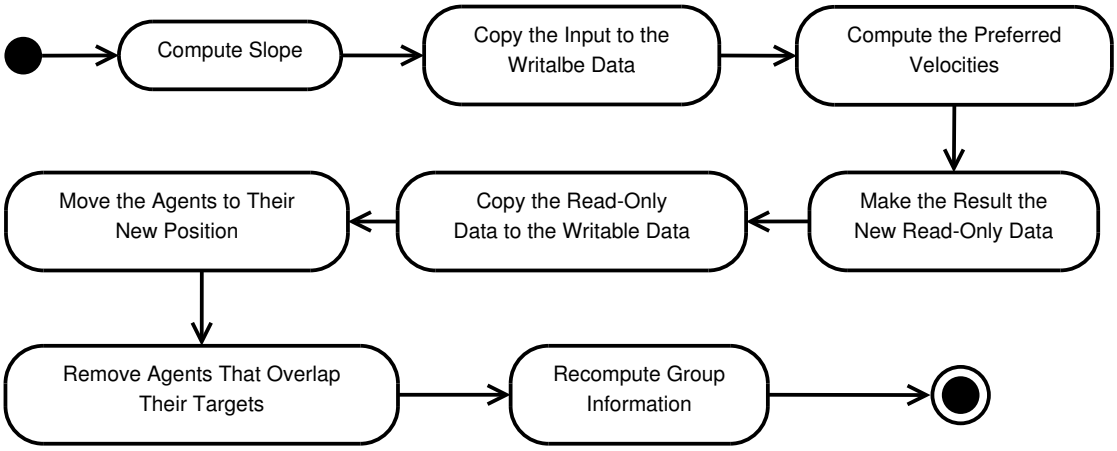
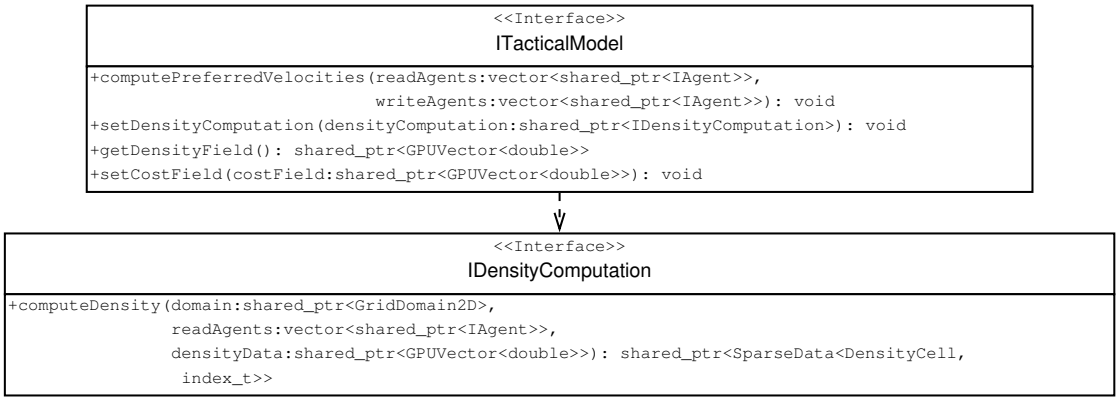


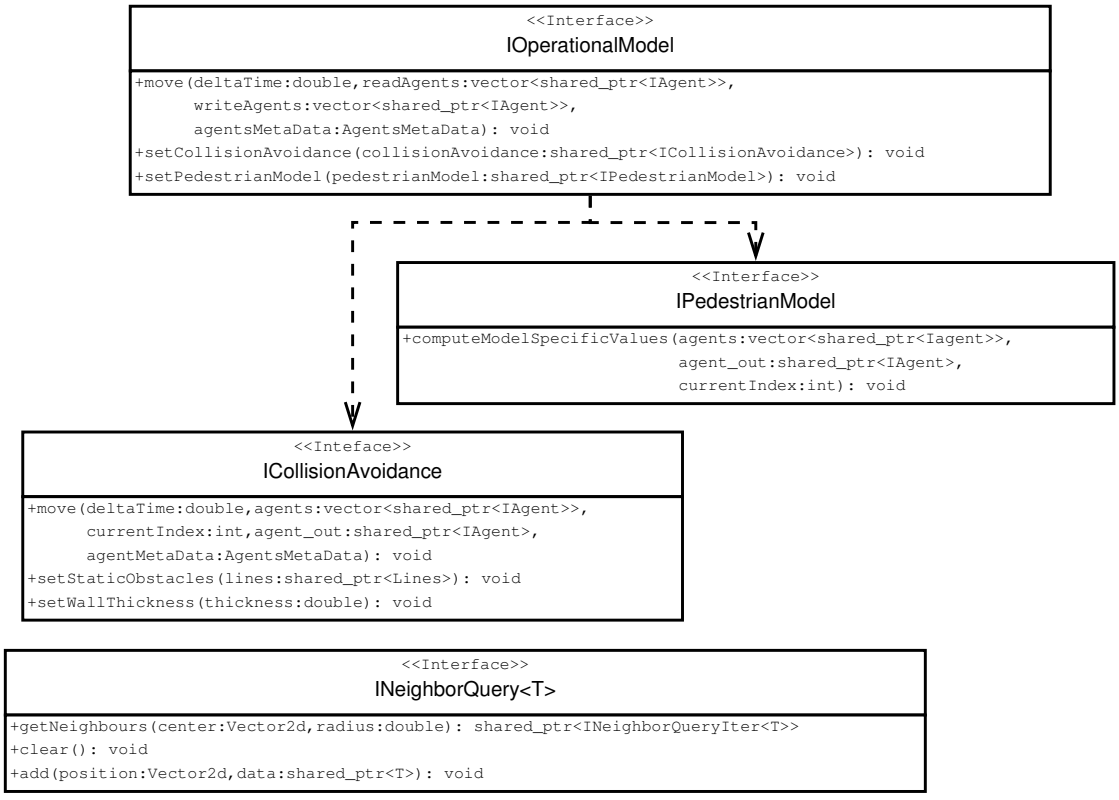
Figure 5.10: The activity diagram of the simulation executor's step computation

The agents are stored twice during the simulation, i.e., in a read-only variant and a writable variant. The read-only variant represents the state of the agent at the start of the simulation frame while the writable variant represents the end of the simulation frame, i.e., the result of the simulation step. This enables a multithreaded implementation of the algorithm where each agent's new position is calculated in parallel. The CrowdSimulationExecutor handles these two states. As seen in the UML activity diagram in Figure 5.10, the executor fist computes the slope at each agent's position and saves it. Then it sets the modified data as the new read-only data and initializes the writable list with deep copies of the agents. In the next step the preferred velocities are computed by the tactical layer. Because our new current state is now in the writable data, we have to swap and copy it like before. Then the agents are moved to their new position by the operational layer and the executor checks if they overlap with any of their targets. If so, the agents are removed. After that the groups data structure has to be rebuild to make up for the removed agents.

The tactical layer contains two interfaces. The *ITacticalModel* represents the model



(a) The interfaces of the tactical layer



(b) The interfaces of the operational layer

Figure 5.11: The class diagrams concerning the simulation interfaces of the tactical and operational layer

that should be used to compute the preferred velocities for the agents and its method *computePreferredVelocities* offers this functionality. The tactical model may also need some additional parameters like the cost field or the density computation method to use and has methods to set these parameters. The *IDensityComputation* interface is used by the *ITacticalModel* for computing a density field.

The tactical model has two implementations as seen in Figure 5.12. The QuickestPathModel is an implementation of Tobias Kretz et al.'s algorithm [KGH+11] that has been extended to handle arbitrary cost fields. The ShortestPathModel just uses the static distance map to assign preferred velocities.

The operational layer is represented by the *IOperationalModel*. Its method *move* moves the agents as close as possible in the direction of their preferred velocity while also ensuring that they do not collide with other agents or obstacles. It may utilize an *IPedestrianModel*. This model can compute additional values that may be used in the collision avoidance. In our case we implemented the pedestrian speed model described in Section 3.2.2 using this interface. The *IOperationalModel* delegates the moving of the agents to the *ICollisionAvoidance* interface which offers a *move* method for single agents. In our case the *ICollisionAvoidance* is implemented by the ORCA algorithm discussed in Section 3.2.1. While computing the new position of an agent it is very likely that its neighbors are needed. The *INeighborQuery* offers an iterator to go through all neighbors.

5.3 The Data Flow Diagram of the Pedestrian Simulation

In order to utilize the simulation data the user has to able to view and interact with it. As discussed in Section 5.1.1 Visdom offers the Data Flow Diagram as a graphical scripting language to build applications. Figure 5.13 shows an overview of the diagram used to create the presented application. The big boxes are groups that contain nodes. Because of the complexity of the diagram each group will be discussed individually.

Terrain

The most important node in the group shown in Figure 5.14 is the *Synth Grid* node, which defines the domain and the regular grid inside the domain. The user can edit it using the *Grid Range 2D* node, which provides a way to change the size and position of the domain via the GUI. The height field defined on the domain can be generated in three ways: a flat height field, a procedurally generated one, or a height field from Geo Information System (GIS) data. The *Terrain Choice* passes the height field selected by the user on to the other nodes like the simulation node or the *OpenGL* node. The node *Heightfield Derivative* node computes the derivative of the field, so the output is a vector field. The following three nodes convert the vector field into a scalar field by computing the magnitude of the vectors and scaling the results by a constant. This scalar field is then pass on to the simulation as the cost field for the tactical layer.

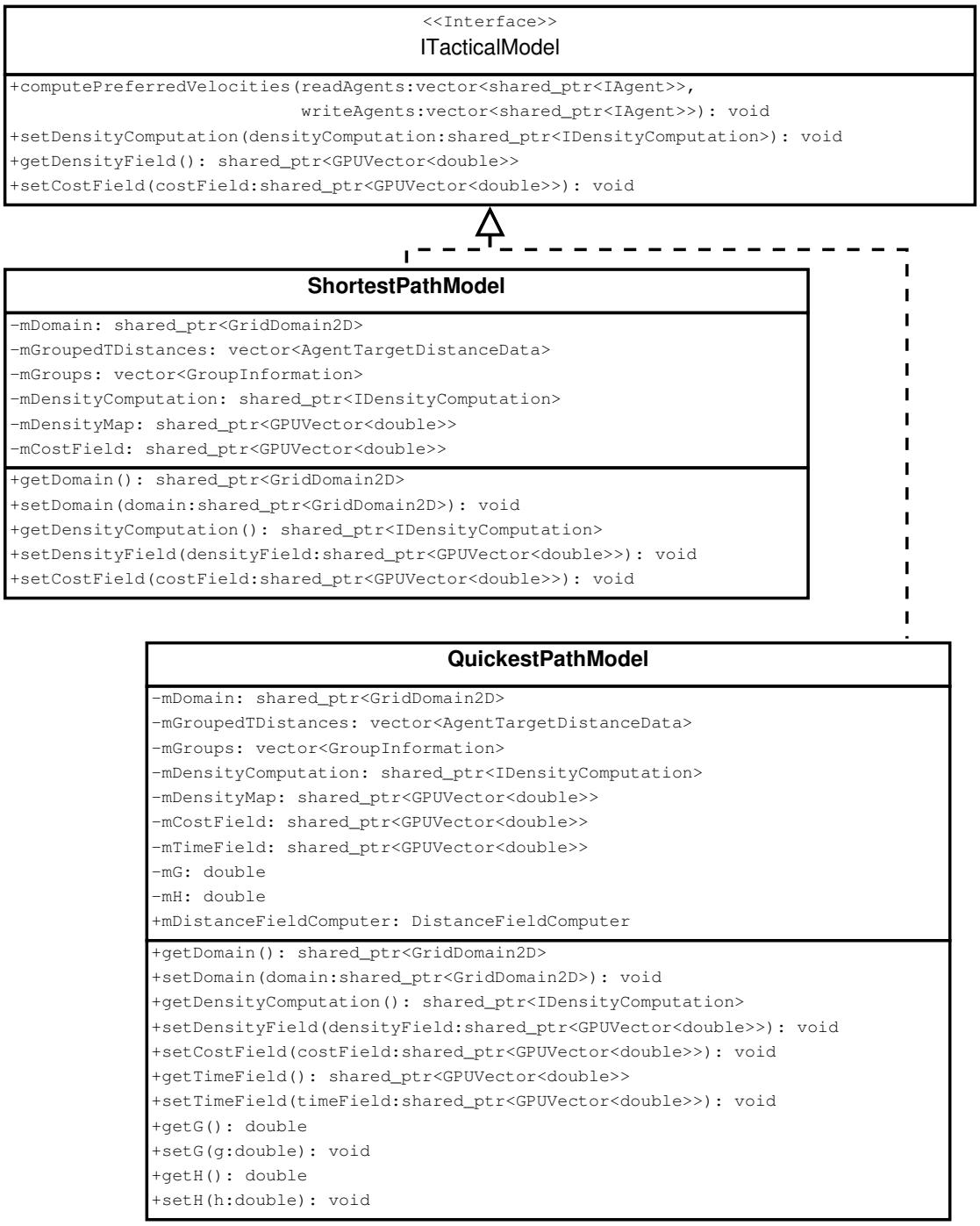


Figure 5.12: The class diagram of the tactical model

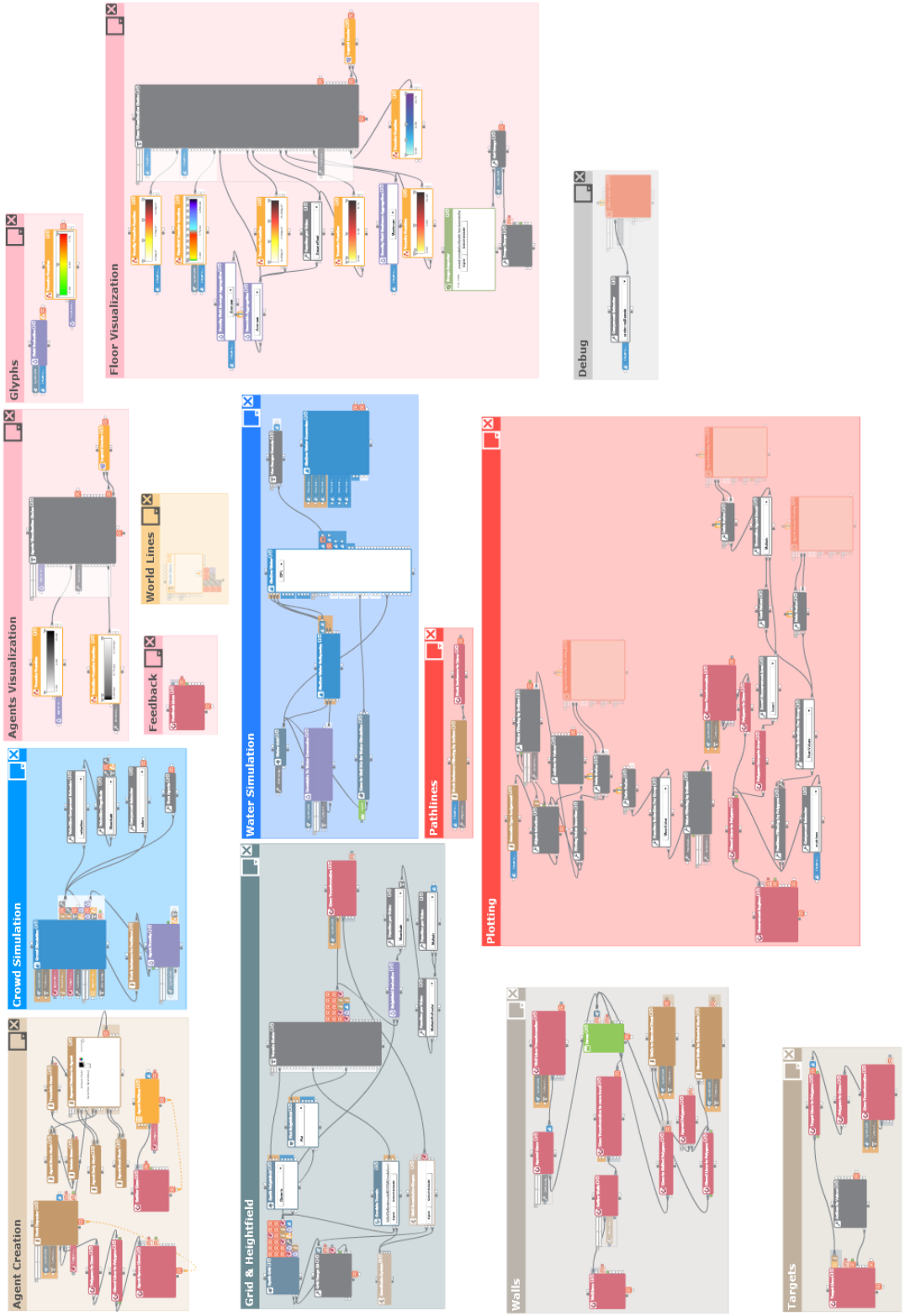


Figure 5.13: An overview of the data flow diagram of the pedestrian simulation

The GIS data needs a coordinate transformation from the coordinates in the GIS file to the simulation domain. This is provided by the *Coordinate System* node, which is used by the GIS terrain and buildings. The user may choose if the GIS buildings should be imported, which also then need to be transformed to fit to the ground.

Walls

The walls (see Figure 5.15) come from different sources. User-drawn walls are generated in the *WallLines* node. The edges of the domain are converted to walls and optional buildings can be imported too. All these lines are merged and then the doors are cut out in the *Doors* node. In the top path (annotated with *Adapt Lines for Edit Mode*) the lines are then transformed for the edit mode so that they follow the shape of the terrain.

In the lower path (annotated with *Extrude Lines to Form Walls*) the wall lines are parallelly shifted and combined to polygons to form the base of the wall. Then these polygons are extruded to create walls instead of just a simple line.

Targets

Target creation can be seen in Figure 5.16. The target lines are drawn by the user and this is handled in the *Target Lines* node. These lines have to be closed, so the following two nodes *Target Lines to Polygons* and *Polygons to Lines* take care of that by first converting the lines into a closed polygon and then back into lines. Lastly these lines need to be transformed in a way that the don't clip through the terrain.

Agent Creation

Figure 5.17 shows the group that is responsible for the agent creation. There are three nodes of interest here: the *Append Meshes to Agent*, *Mesh Repeater*, and the *Agent Emitters*. The *Append Meshes to Agent*-node takes three primitives (two cones and a sphere) and creates a single mesh that represents an agent. This mesh is then used in the *OpenGL*-node to render them.

The *Mesh Repeater* is in charge of creating agents in bursts. This means that these agents all appear in the scene at the same time. This is useful to initialize simulations. In order to place the agents on the ground, the domain of the terrain and the height field are needed. To place agents inside of regions, the lines input is required. Each line corresponds to a burst of agents in the simulation and this burst can be configured to change the density or parameters of the spawned agents. Because the agents need a target, the target lines are connected to the *Mesh Repeater*.

The *Emitter Creator*-node takes care of creating emitters that will in turn emit agents over time. This node also uses lines to define the area where agents should be emitted and it also requires the target lines to assign agents to targets.

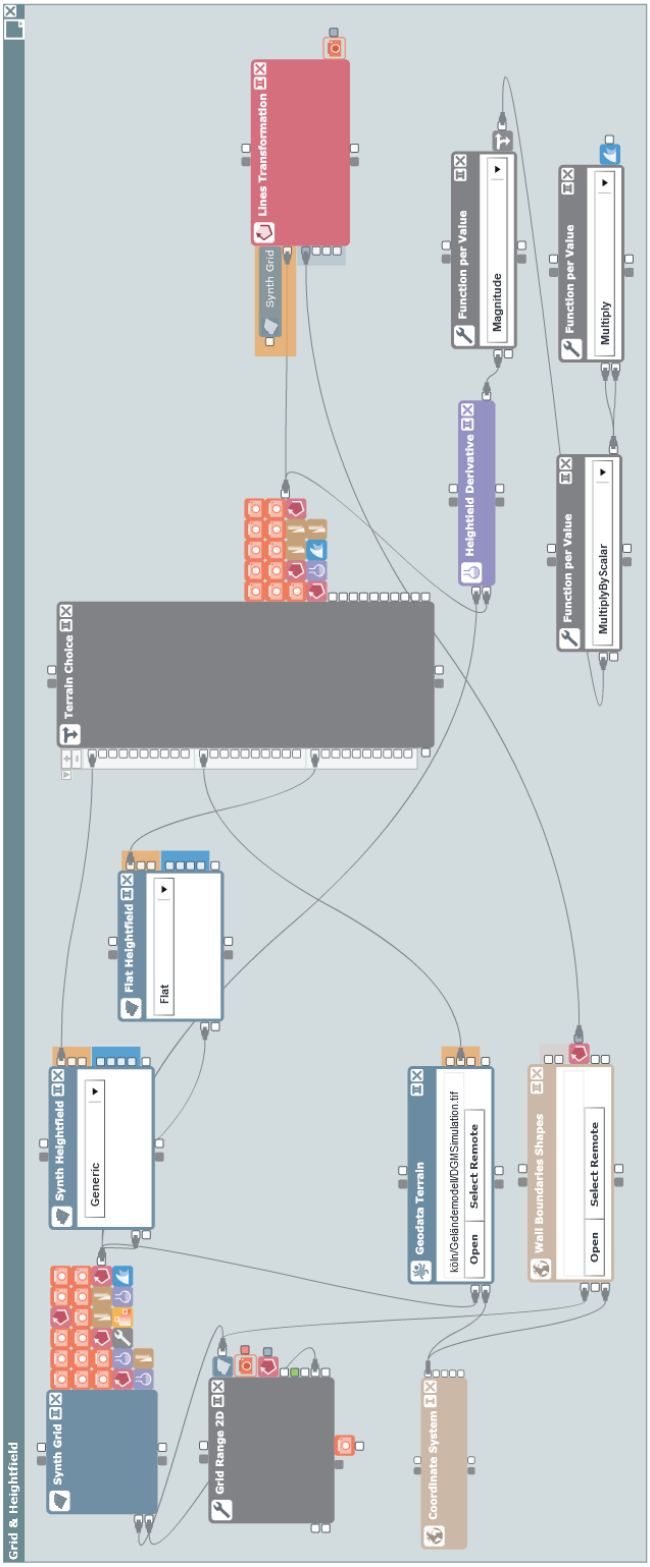


Figure 5.14: The Data Flow Diagram of the terrain

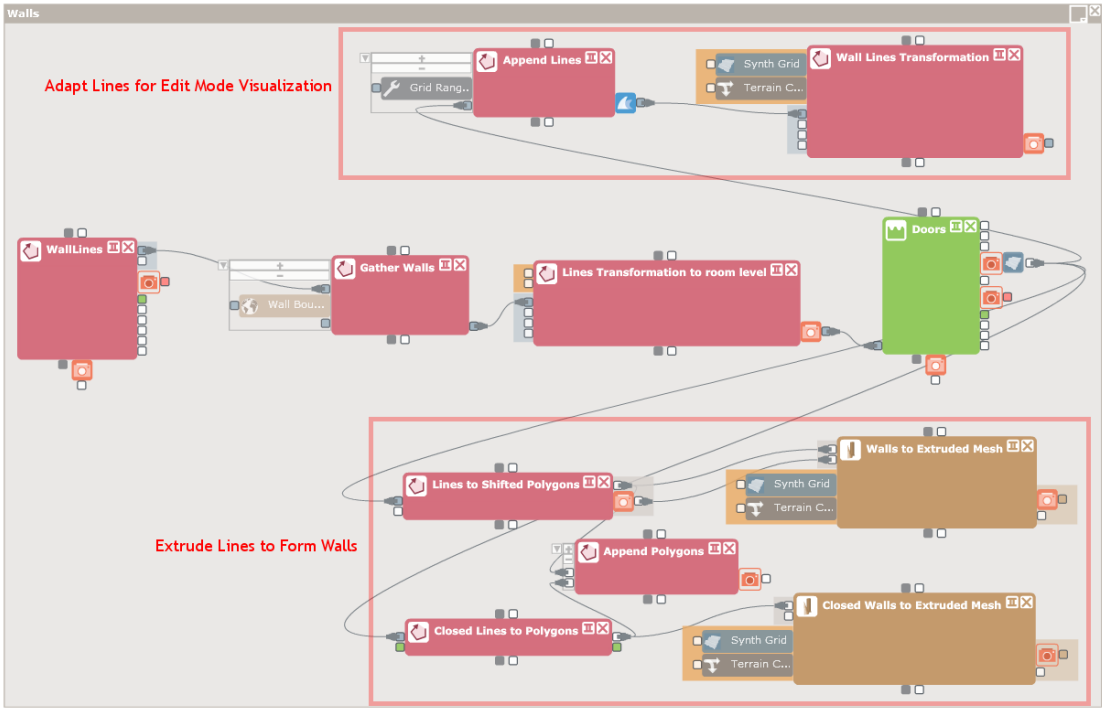


Figure 5.15: The Data Flow Diagram of the walls

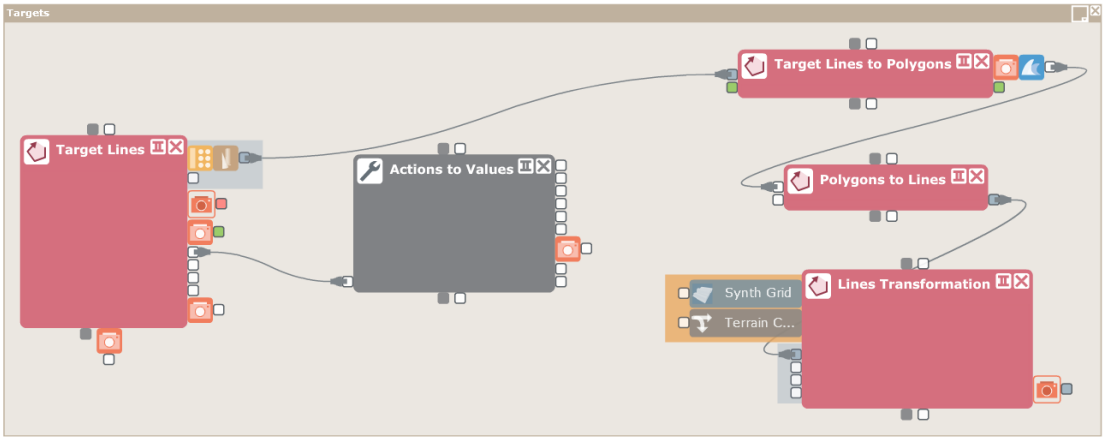


Figure 5.16: The Data Flow Diagram of the targets

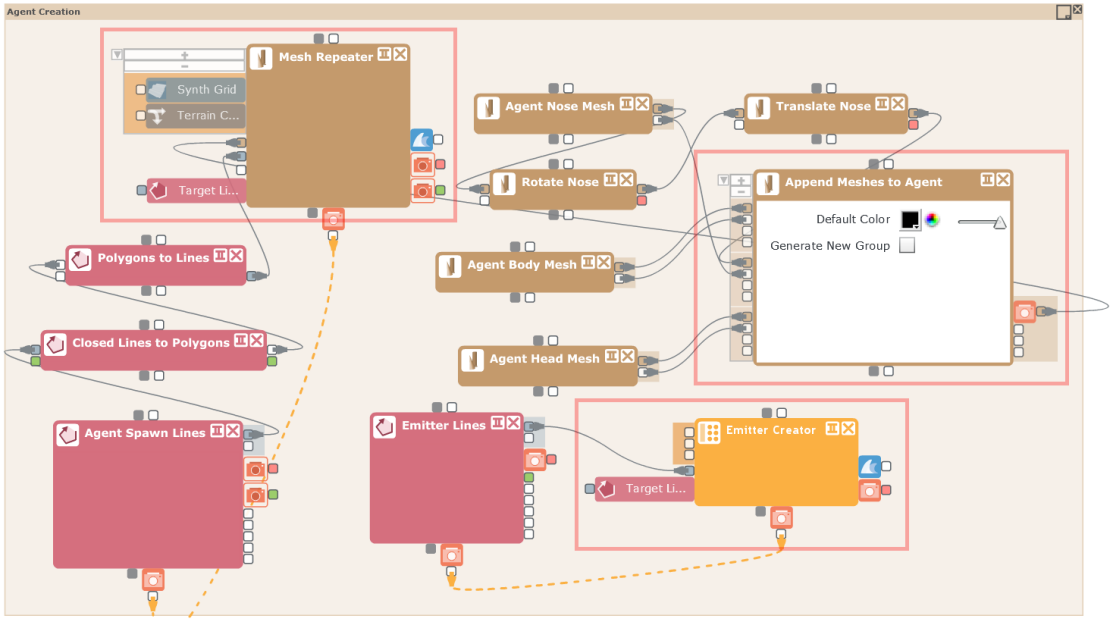


Figure 5.17: The Data Flow Diagram of the agent creation

Water Simulation

In Figure 5.18 the domain of the pedestrian simulation is converted to a domain of lower resolution as the flood simulation does not need such a high-resolution grid to work on. The simulation is done in the *Shallow Water* node which then outputs several data sets like the domain of the water, the height-field, etc. The *Shallow Water Conversion* node converts these sets into a renderable representation. The domain-output of the *Shallow Water* node is also used in the pedestrian simulation to determine flooded areas.

Pedestrian Simulation

The most important node in Figure 5.19 is the *Crowd Simulation* node. As described in previous sections, it computes the simulation. The other nodes in the figure process the agents as outputted by the simulation node. The two nodes at the bottom left first convert the agents into positions and then compute the density map. Additionally the current speed and color are extracted for each agent as well.

Terrain Visualization

The terrain visualization group handles the coloring of the terrain, the agents are moving on, and can be seen in Figure 5.20. Each data field to display has its own *Transfer Function* node which handles the mapping from a numeric value to a color. Some fields are computed here using data from the simulation group, e.g., the *Density Field Average Aggregation* node computes the average of all past density fields up to this point.

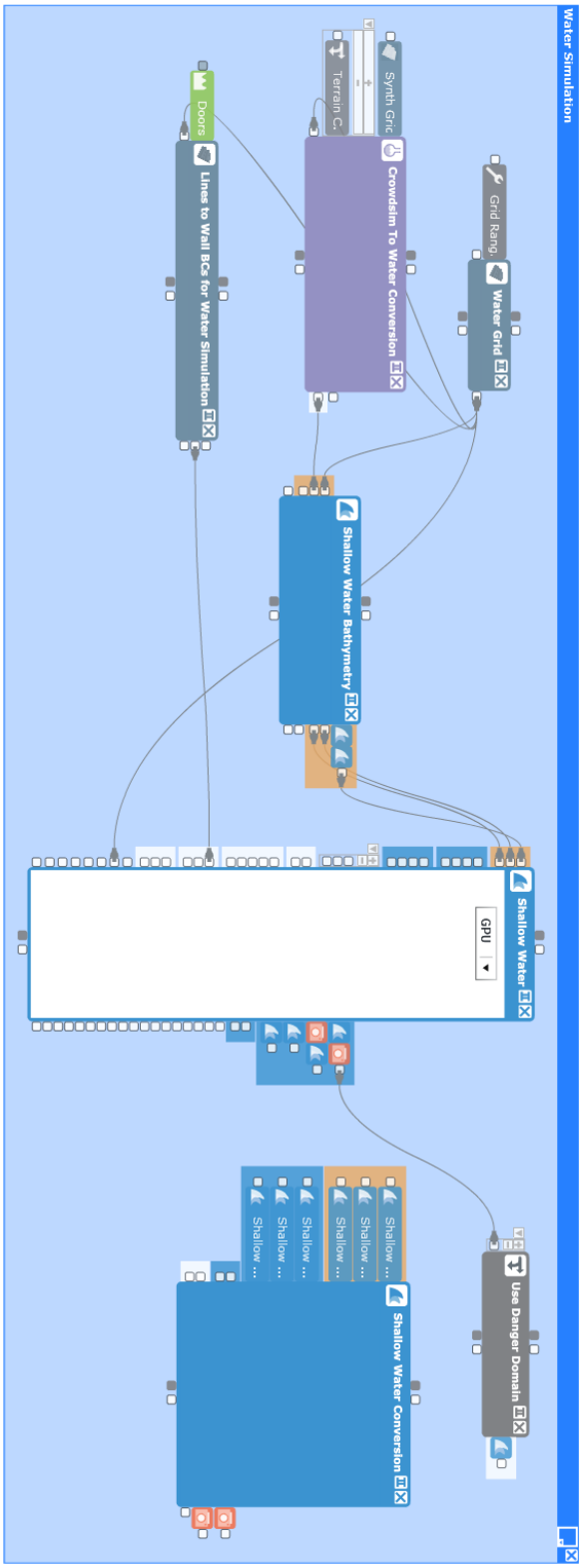


Figure 5.18: The Data Flow Diagram of the water simulation

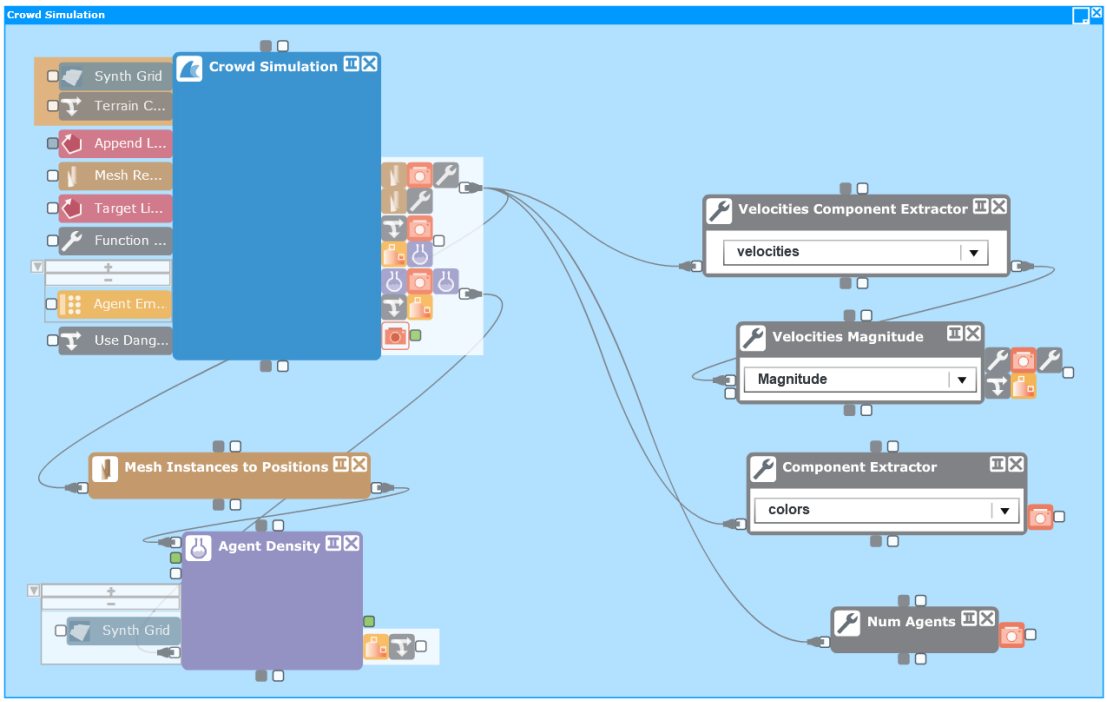


Figure 5.19: The Data Flow Diagram of the pedestrian simulation

The *Domain Visualization Choice* node lets the user choose which of the predefined visualizations to use. The bottom three nodes enable the user to display a custom image on the terrain, e.g., a floor plan. The *Cut Image* node can be used to position and scale the custom image on the terrain by altering the uv-coordinates of the image texture, effectively cutting the image, so it only displays the relevant part. The image texture is then rendered on the terrain.

Agent Visualization

Figure 5.21 shows the group that handles the visualization which is applied to the agents. The transfer functions here should only map the values to colors between white and black as this color is then multiplied with the base color of the agents to adjust their brightness. The *Agent Visualization Choice* node lets the user choose which of the two measurements should be displayed as brightness: the density or the speed.

Plotting

The plotting group can be seen in Figure 5.22 and shows how the diagrams *Selected Agents Velocities*, *Agent Count in Region*, and *Agent Density in Region* are being made. At the top of the group the *Selected Agents Velocities* plot is handled. An *Object Selection* node filters agents by user selection and passes it on. Next the velocities of these agents are extracted in the *Value Filtering by Indices* node at the top. Because the *Data Series*

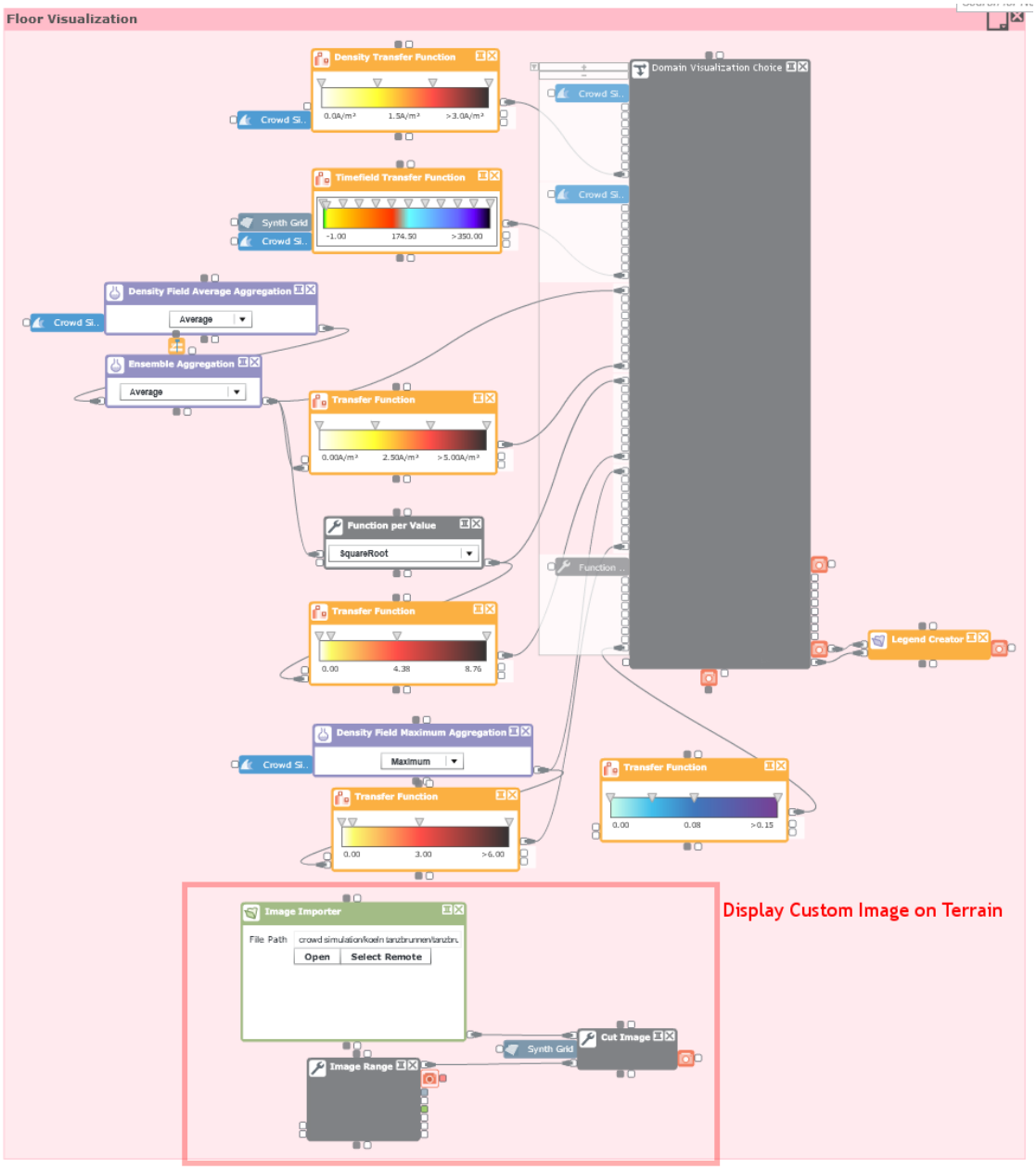


Figure 5.20: The Data Flow Diagram of the terrain visualization

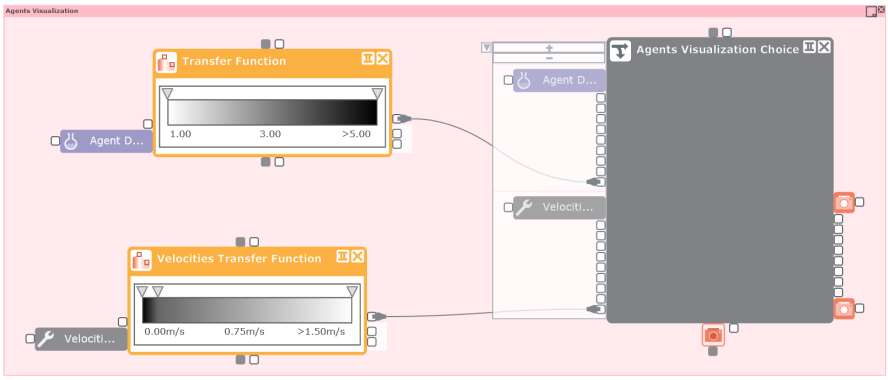


Figure 5.21: The Data Flow Diagram of the agent visualization

node expects a constant number of agents being displayed, we have to insert a zero value for each agent that, first, has been selected by the user and second, has reached its target and is no longer part of the simulation. The *Missing Value Insertion* node takes care of this and the *Data Series* node is able to build a series out of the data that can then be displayed by the *Agent Selection Plotting* node as a graph.

The *Agent Count in Region* and *Agent Density in Region* plots are similar to the above, but they use the *Positions Filtering by Polygons* node to extract the indices of the agents in regions defined by the user. Using the indices the number of agents inside the regions is computed. For the *Agent Count in Region* diagram these counts are forwarded to the *Data Series* node in order to create a data series that can be rendered. For the *Agent Density in Region* diagram the inverse of the region's area is multiplied with the agent count in the *Normalize Agent Count* node to obtain the agent density. Then this density is also passed on to a *Data Series* node and the generated series is then rendered.

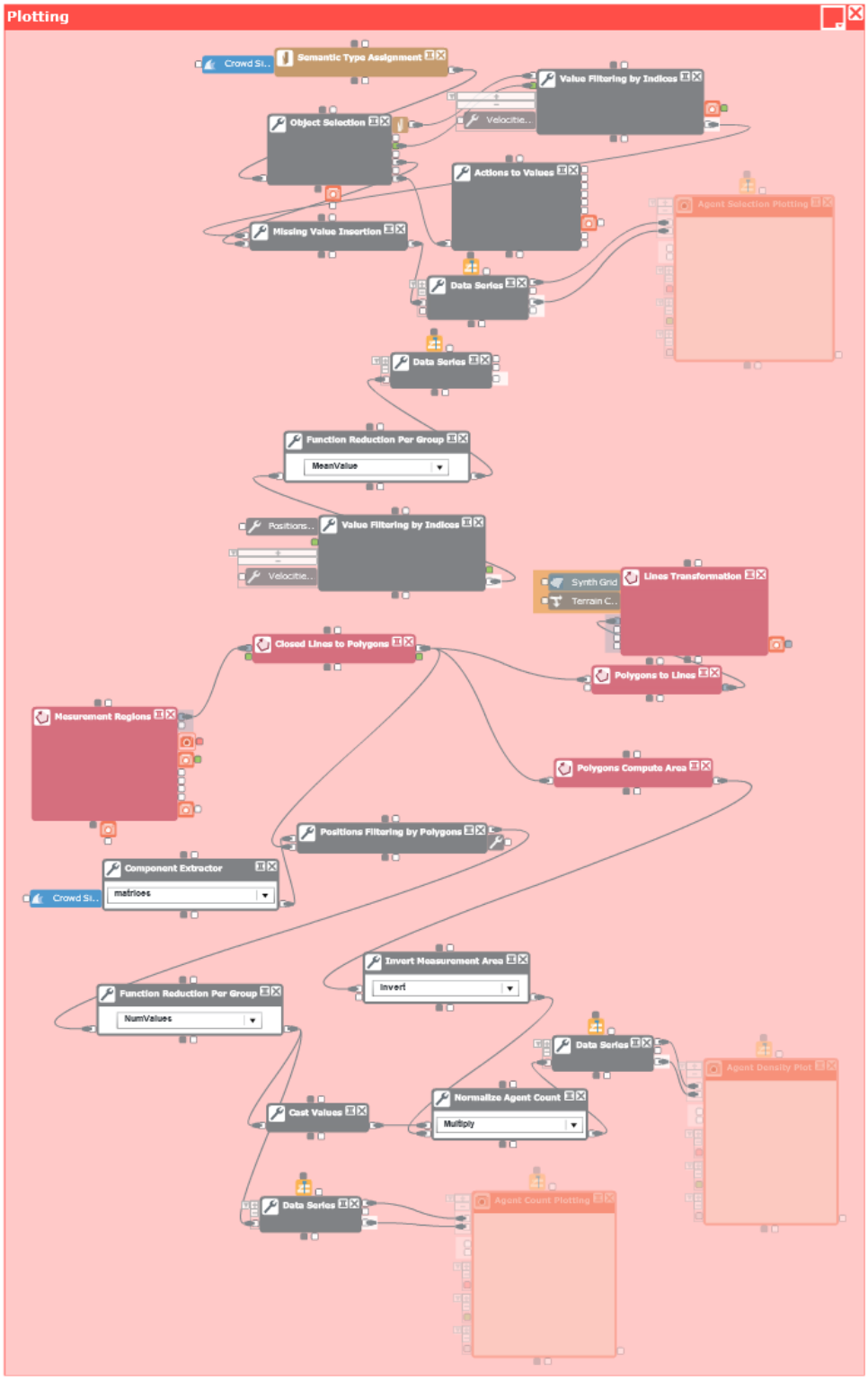


Figure 5.22: The Data Flow Diagram of the plotting

CHAPTER 6

Validation and Case Studies

The model has been validated by test cases from RiMEA [rim]. This document is used by German authorities to offer a standardized way to validate evacuation simulations:

The methodology for a simulation-based evacuation analysis outlined in this guideline is designed to permit assessment of the effectiveness of an escape and rescue concept as part of a built environment. (RiMEA version 3.0.0 [rim], page 6)

Besides the validation provided by RiMEA, we also wanted to test how a user would interact with the system, so we created a real-world scenario at the Tanzbrunnen in Cologne. Here visitors of an open air concert are evacuated plus different scenarios are being simulated and tested against each other.

We will start with the validation by the RiMEA test cases and then continue with the real-world scenario.

6.1 RiMEA Test Cases

6.1.1 Test Case 6

In this test case 20 agents have to walk around a corner without passing through a wall. The system passed this test as shown in Figure 6.1. This figure also shows that agents walk in pairs of two around the corner rather than forming a line at the corner thanks to the used tactical model.

6.1.2 Test Case 9

Here $1,000$ agents have to leave a public space through four doors (see Figure 6.2) and then the test is repeated with only two open doors. In case of the four open doors the evacuation should be nearly twice as fast as with only two open doors. In our simulation

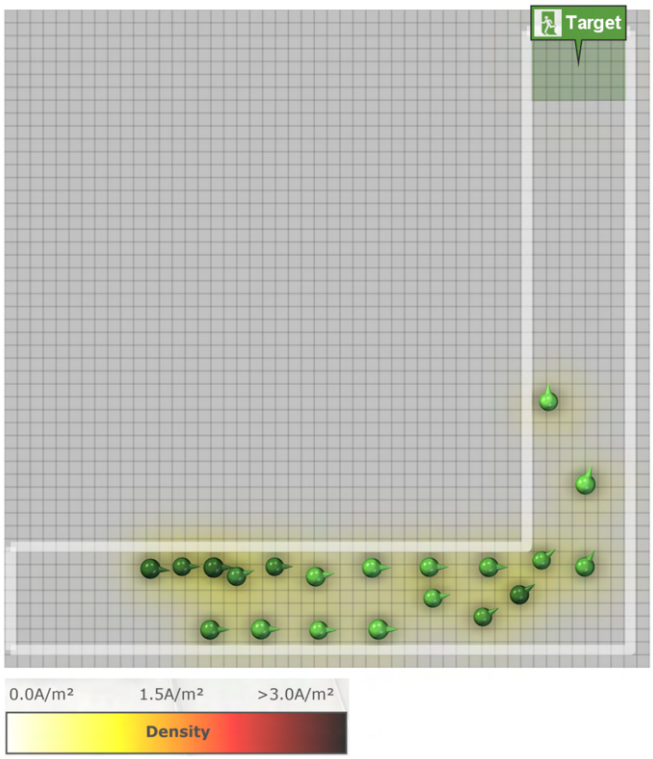


Figure 6.1: RiMEA Test Case 6 after 6 seconds

the first case had an evacuation time of 206s and the second case an evacuation time of 428s. This means that the scenario with two open doors requires 208% of the time of the case with four open doors, which satisfies the testing criterion.

6.1.3 Test Case 10

In this case 23 agents are distributed among 12 rooms that are connected by a corridor with two exits. The agents in the four rightmost rooms have to leave through the right exit while all other agents have to leave through the top exit (see Figure 6.3).

6.1.4 Test Case 11

A large room filled with 1000 people shall be evacuated through two doors. All agents approach the two doors from the left. The expected result is a congestion at the first door, but some agents are using the alternative (more distant) exit. As can be seen in Figure 6.4 with our approach agents utilize both doors.

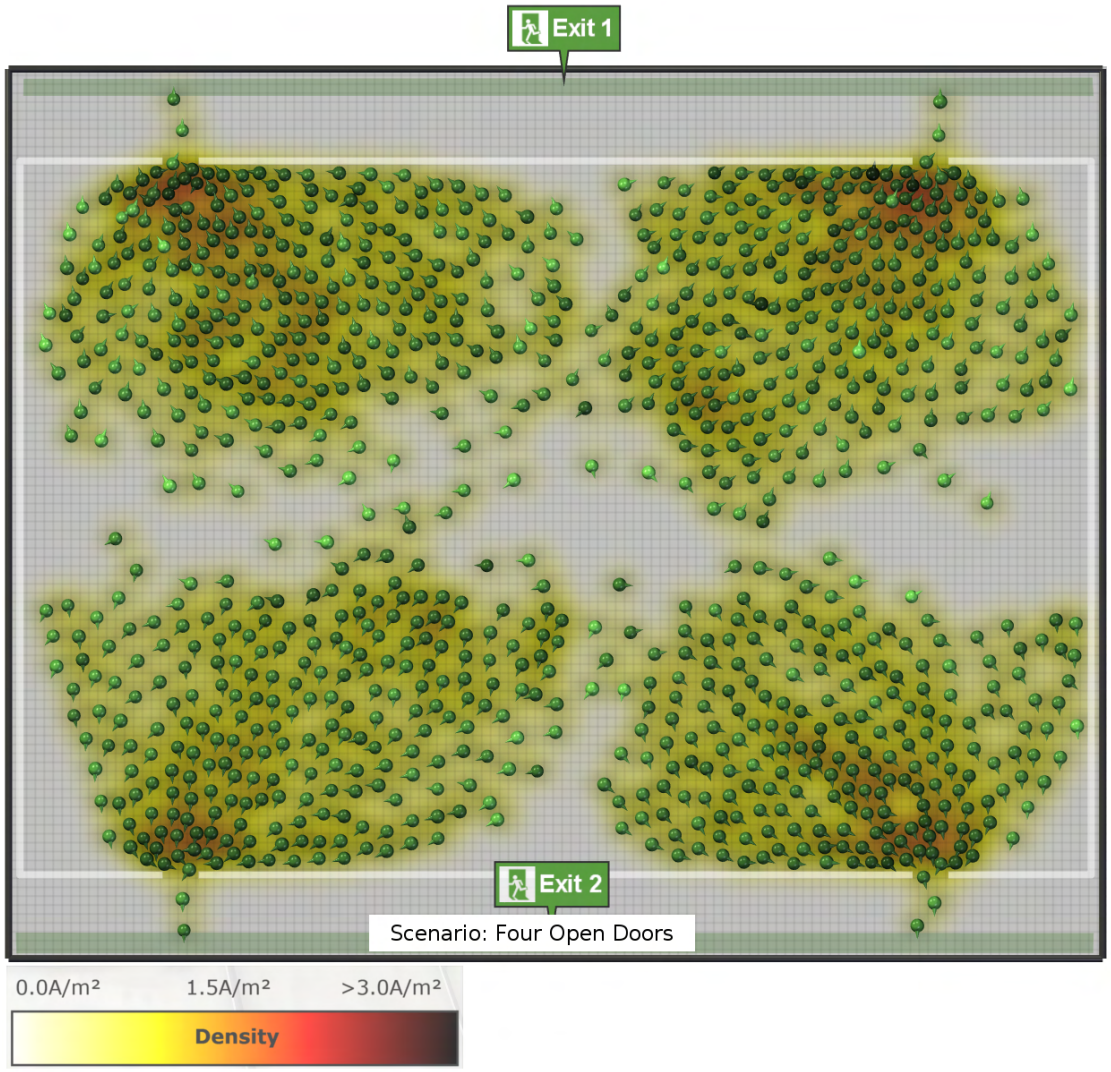


Figure 6.2: RiMEA Test Case 9 Setup with four open doors. For the other test case the lower two doors are closed.

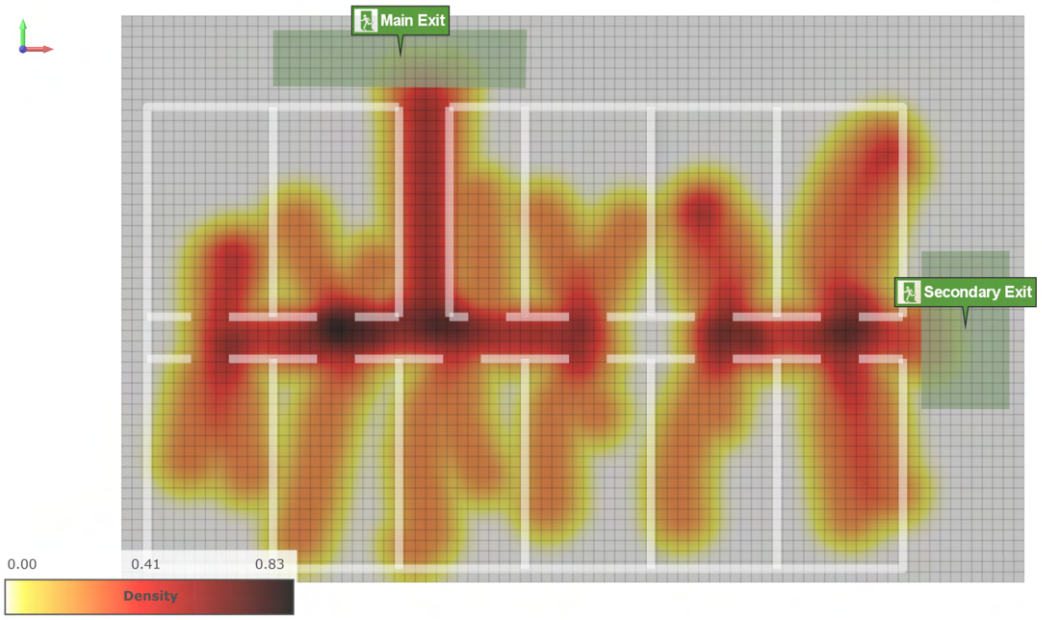


Figure 6.3: RiMEA Test Case 10 Path Traces

6.1.5 Test Case 12

Two rooms are connected by a one meter wide corridor and 150 persons have to walk through Room 1, the corridor and then cross Room 2 to reach the final exit. The expected result is a congestion when the agents try to enter the corridor, but none at the exit of Room 2 as the flow is already constricted by the narrow corridor.

6.1.6 Test Case 15

The behavior of agents when walking around corners is tested in Test Case 15. Three corridors are constructed. One with a bend, a straight one that has the same length as the innermost (shortest) path of the bent corridor and a second straight one with a length equal to the outermost (longest) path of the bent one. The corridor with the corner should be evacuated before the long straight, but after the short straight one has been. Our tests showed that this is the case with our approach.

6.2 Variations on the Tests

We conducted the RiMEA test cases also with different settings. The first run was the default run with the default settings: enabled speed model and quickest path model for the tactical layer. Then one run with disabled speed model and quickest path was simulated. The last variation was configured to run with enabled speed model, but with a simple shortest path model for the tactical layer. The differences between these variations

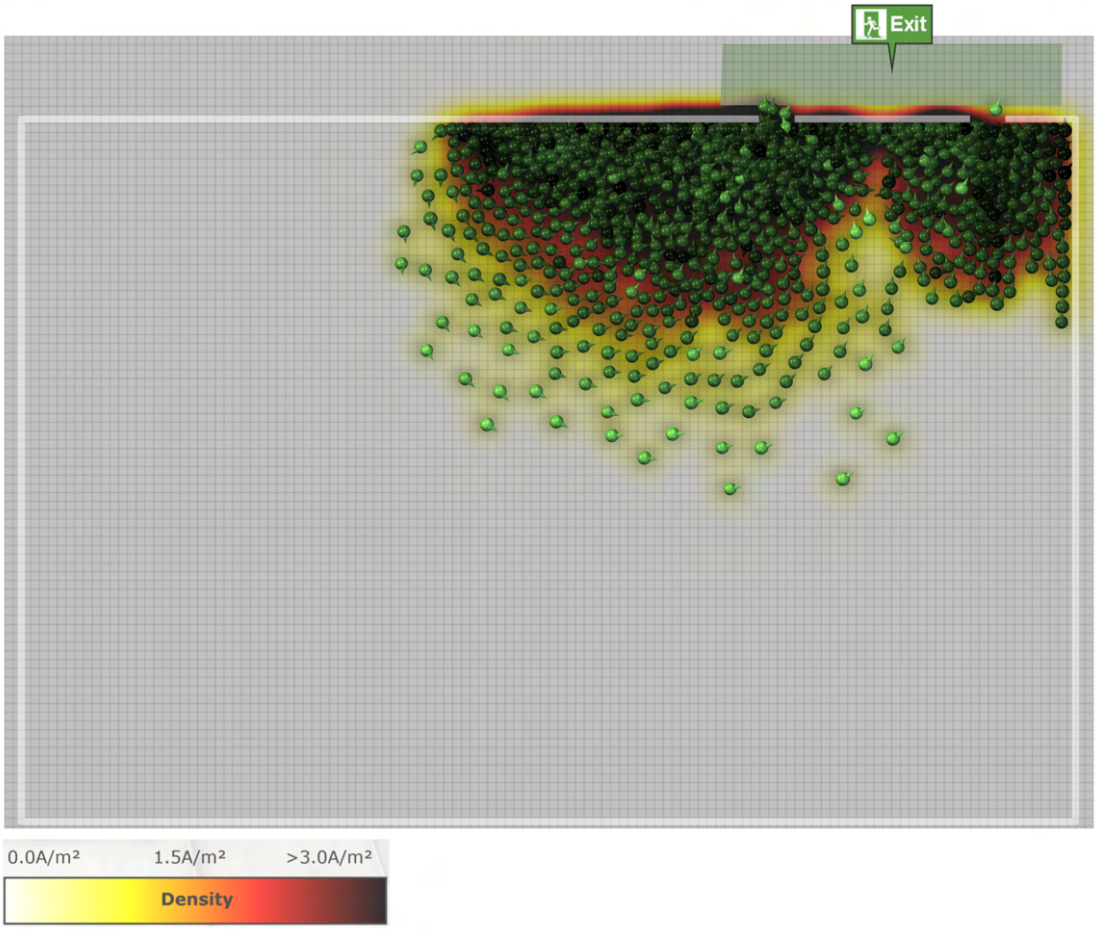


Figure 6.4: RiMEA Test Case 11 after 50 seconds

Test Number	Evacuation Time (s)
6	21.2
9 - 4 Doors	206.4
9 - 2 Doors	428.2
10	19.6
11	318.8
12	155.1
15	96.2

Table 6.1: Simulated evacuation times for the RiMEA Test Cases

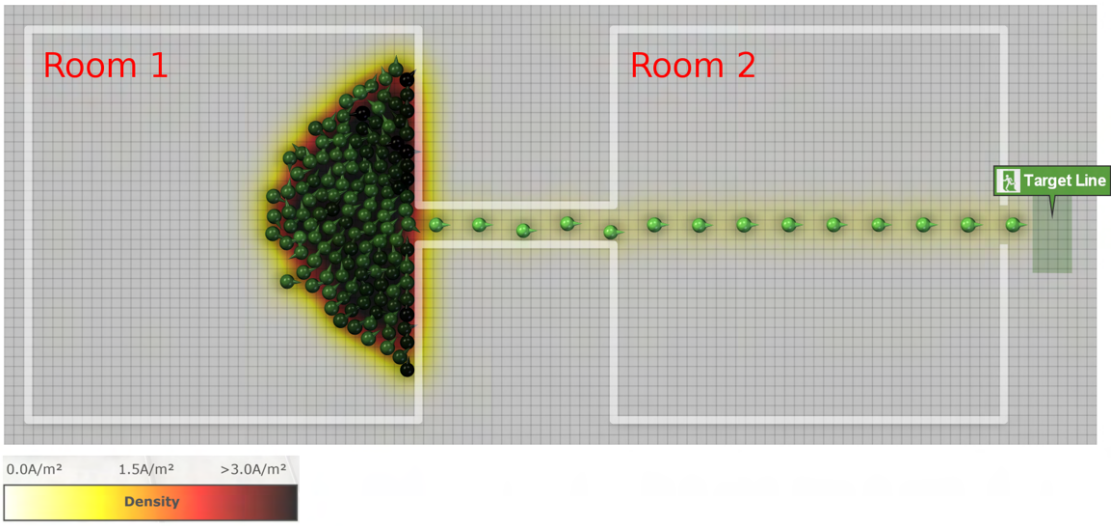


Figure 6.5: RiMEA Test Case 12 Setup

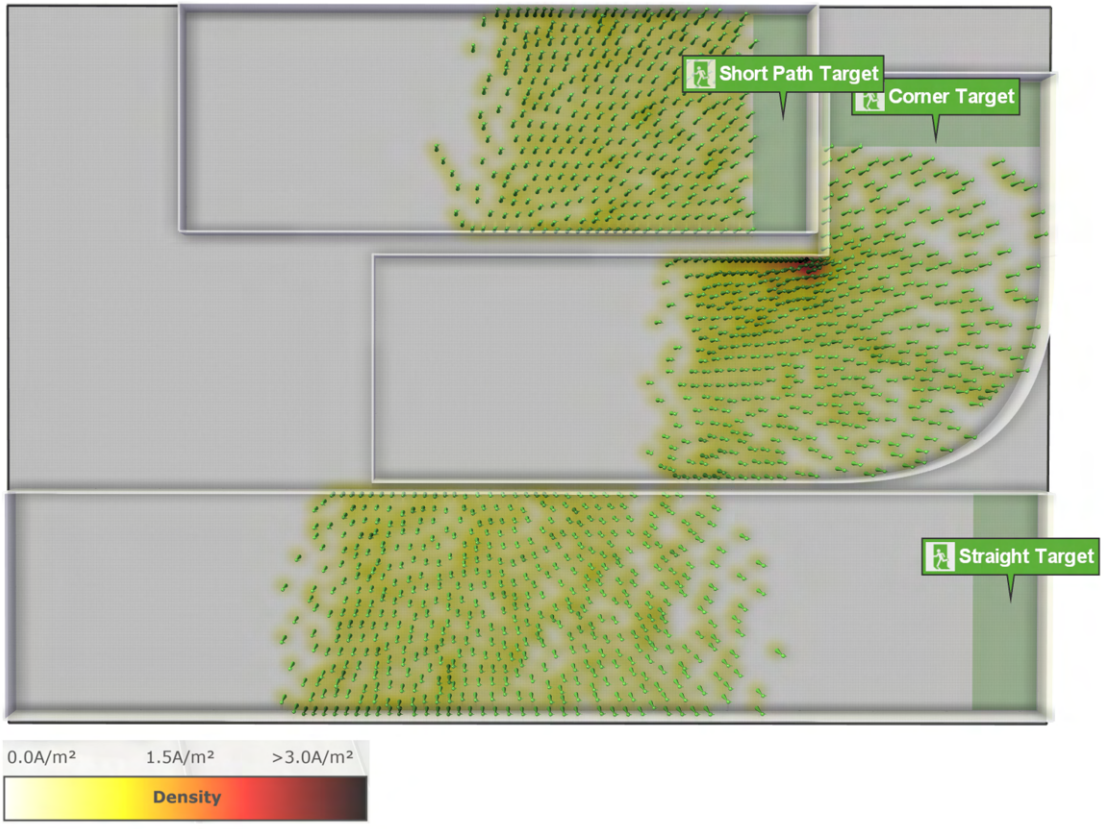
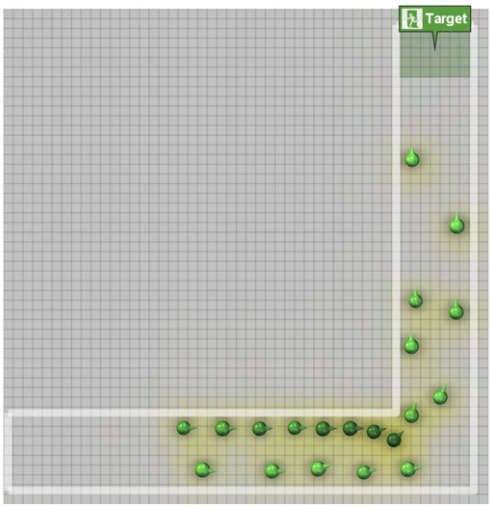
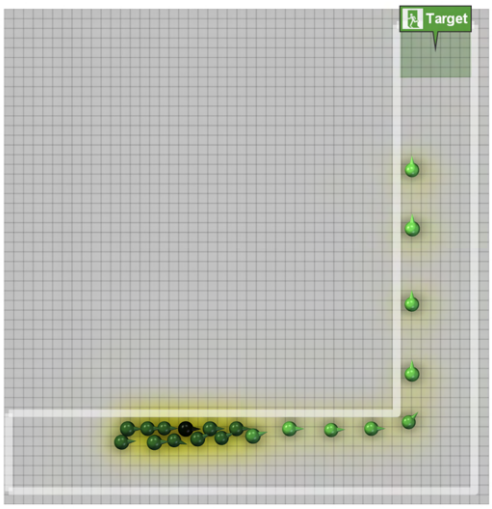


Figure 6.6: RiMEA Test Case 15 Setup



(a) The default variation with the speed model enabled and the quickest path model

(b) Variation 2 with the speed model disabled and the quickest path model



(c) Variation 3 with the speed model enabled and the shortest path model

Figure 6.7: A comparison between three different setting variations for Test Case 6.

are consistent throughout all tests. Without the speed model agents evacuated the rooms much quicker than with it. This is expected as the speed model only decreases the speed of pedestrians. Figure 6.7 shows a comparison between all three variants on Test Case 6.

With the shortest path model instead of the quickest path model agents created seemingly unnecessary congestion and for example did not walk through the second door in Test Case 11 as can be seen in Figure 6.8.

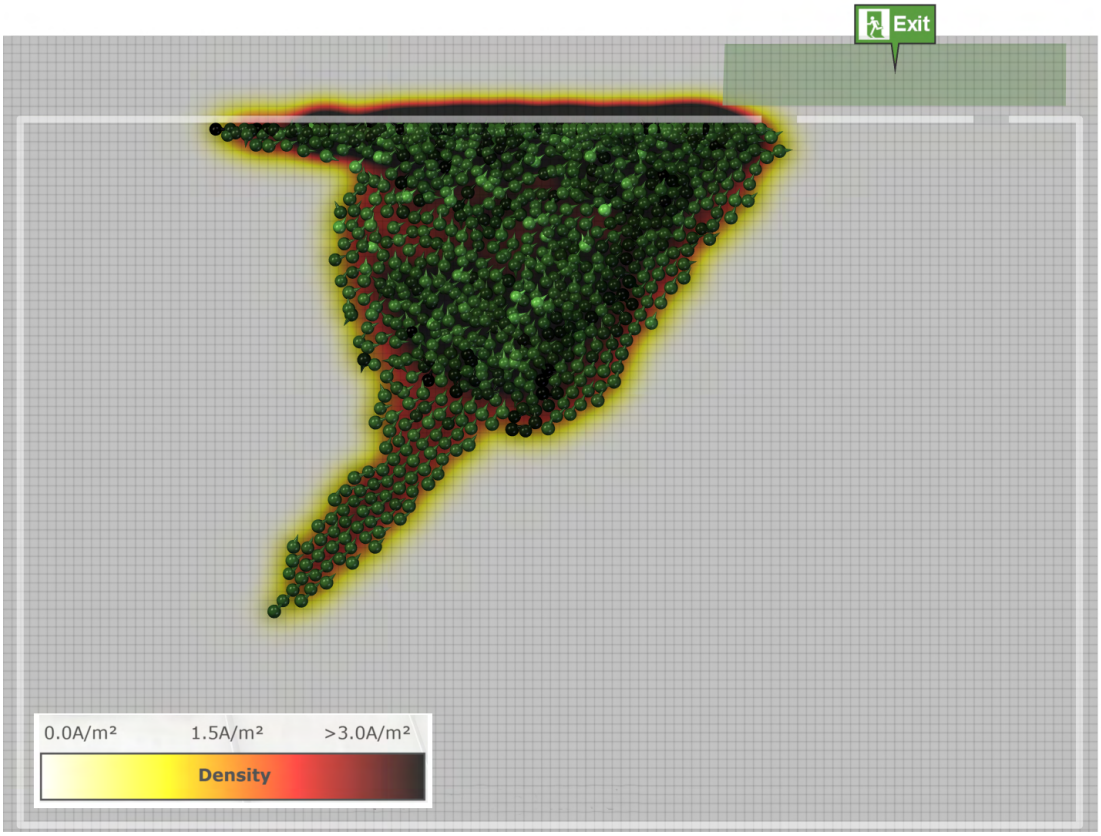


Figure 6.8: RiMEA Test Case 11 with the shortest path model

6.3 Real World Case Study

6.3.1 Setup

To test the system in a real-world scenario the Tanzbrunnen in Cologne was picked as a location for an imaginary open-air concert. The system was given the height data of the Tanzbrunnen and its immediate surroundings. With the help of a plan all obstacles have been added by projecting this plan onto the terrain and redrawing the lines by hand. The plan also had all of the emergency exits marked, so all of them were modeled as open. The evacuation zones for the pedestrians were located in the park to the north and on the street to the south of the event site. See Figure 6.9 for the overlay in the edit mode that was used to draw the walls.

The official homepage of the Tanzbrunnen [tan] states that a maximum of $12,500$ visitors are permitted on an area of $30,000m^2$, so we added $\approx 12,000$ agents to the scenario. After viewing photographs and videos of previous events that happened at the Tanzbrunnen, a pattern emerged in the density of the crowds. As one would expect the most dense place was right in front of the stage and the density decreased with respect to the distance to

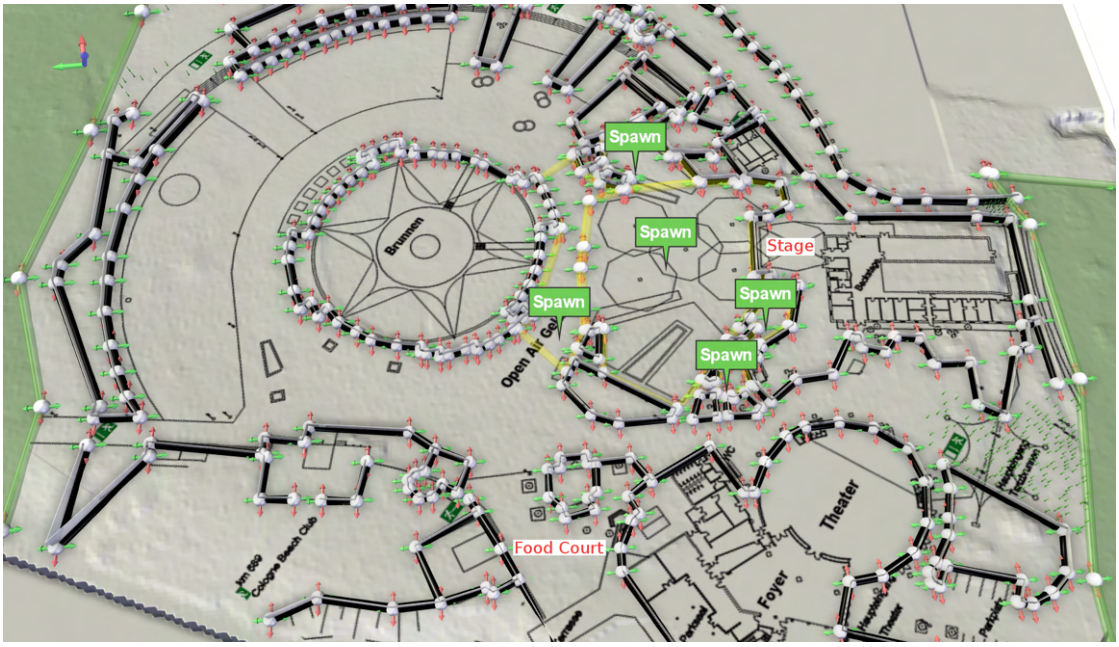


Figure 6.9: The overlay of the floor plan of the area in edit mode

the stage. This was also reflected in the setup of the simulation.

6.3.2 Results

The evacuation took 400 seconds (6,66 minutes). In this time all pedestrians reached the designated evacuation zones (targets). Figure 6.10 shows agents evacuating and in Figure 6.11 the time field of the simulation can be seen.

Subfigure 6.11a shows the unaltered time field of the simulation. This visualizes the expected time an agent needs to reach an exit without any other agents blocking it. It is very useful to find regions that are far away from exits. In our case the region in front of the stage and to the side of the food court are the regions with the highest expected times of around 130 (unitless).

Subfigure 6.11b shows the time field in the middle of the simulation. The time field has changed to accommodate for the agents, because large densities of agents might indicate a congestion or at least a slowdown. Here the largest values are around 340 in front of the stage. This estimation of the evacuation time changes with each time step, because the density of the agents is taken into consideration.

A second variation of this scenario was made and can be seen in Figure 6.12. In this version all small exits next to the stage are blocked, so only the large exits remain. The Tanzbrunnen itself is sealed off additionally so the little pockets, shown in the figure as red striped areas, are not accessible any more.

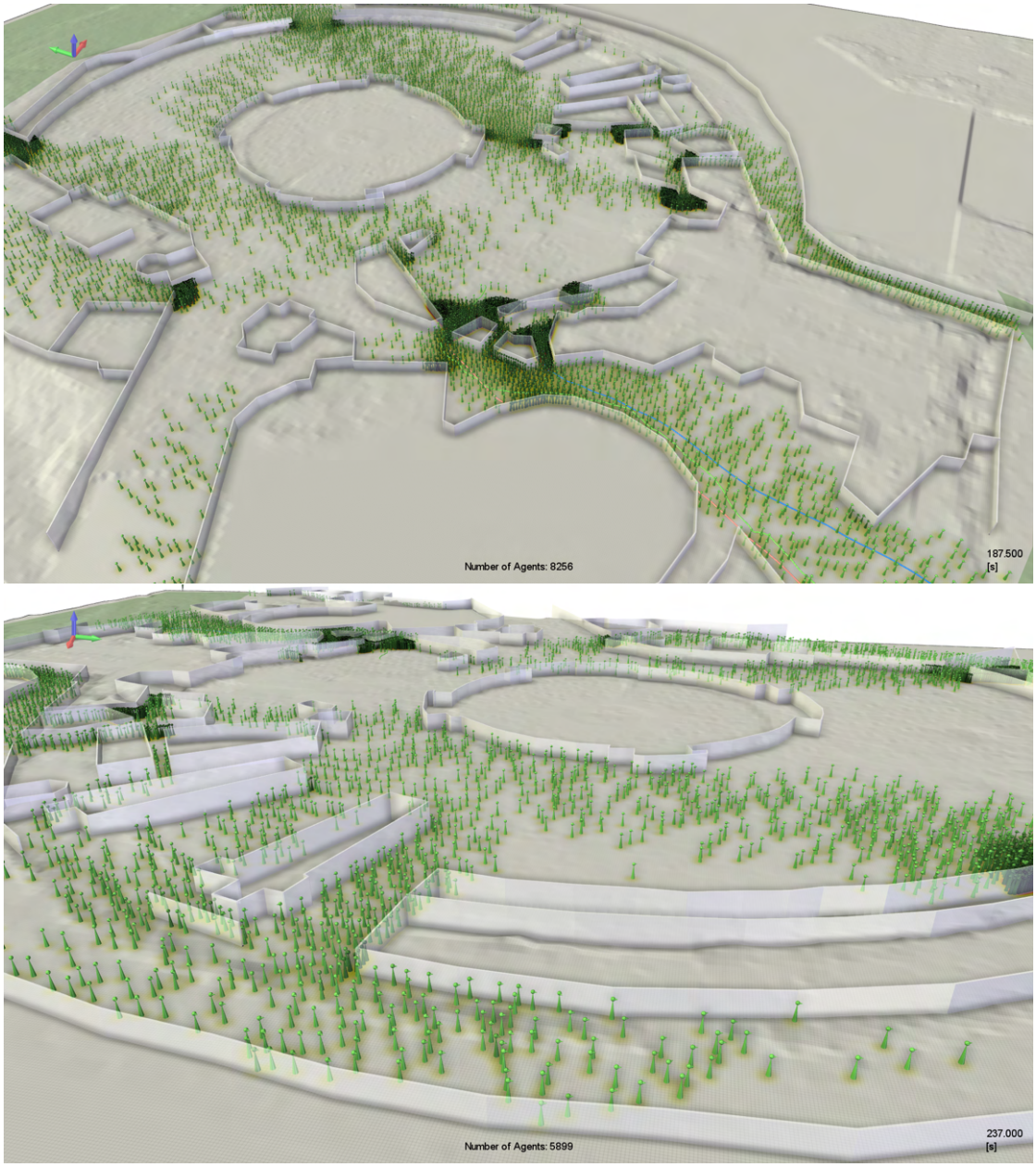
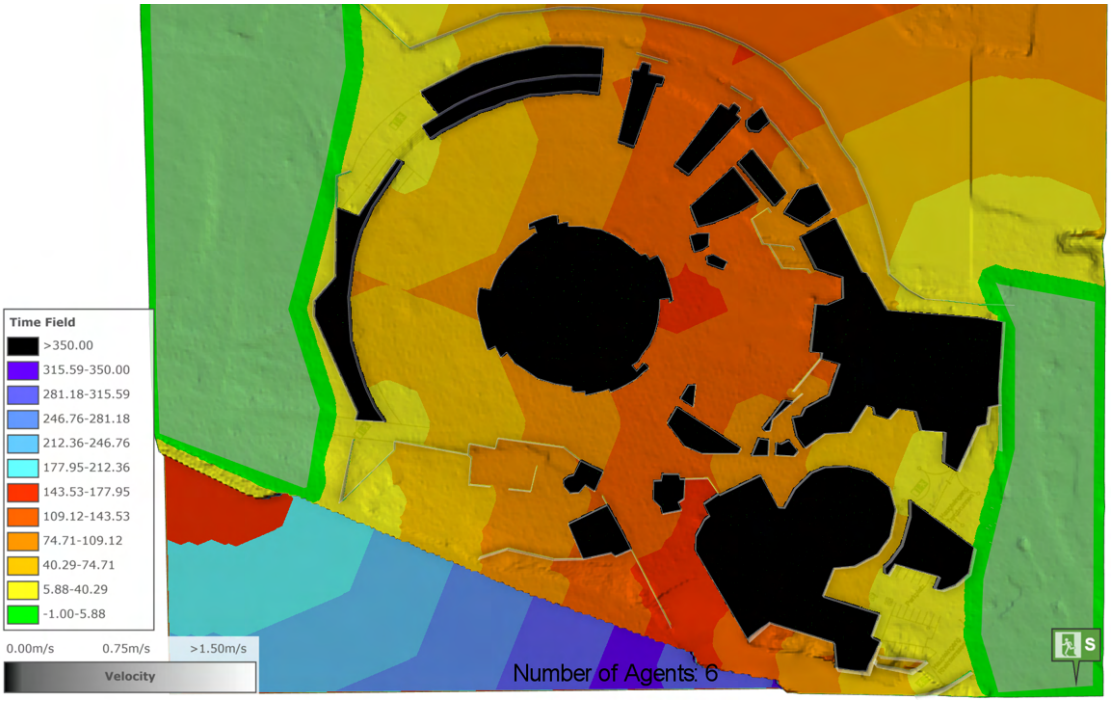
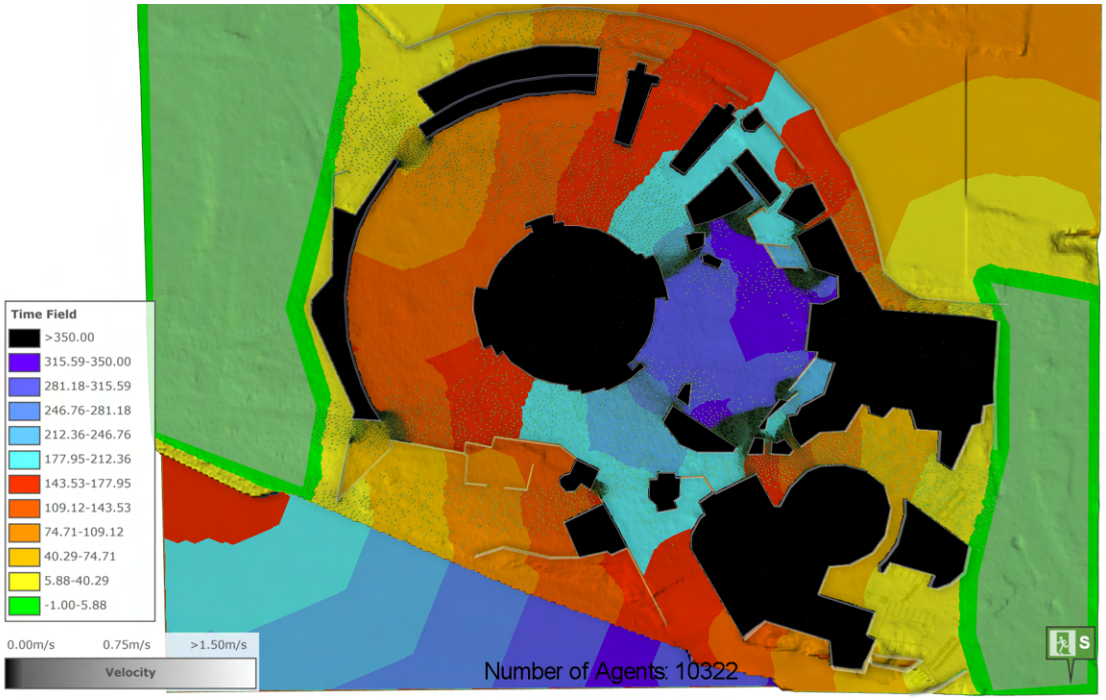


Figure 6.10: The evacuation of the Tanzbrunnen. The number of agents at the bottom is the current number of agents that have not reached an exit yet. At the bottom right the passed simulated time since the beginning of the simulation is shown.



(a) The unaltered time field at the end of the simulation (397s) - it becomes the distance field.



(b) The time field during the evacuation (138.5s). Please note how the distances changed compared to the distance field in (a).

Figure 6.11: The time field at different times of the evacuation of the Tanzbrunnen.

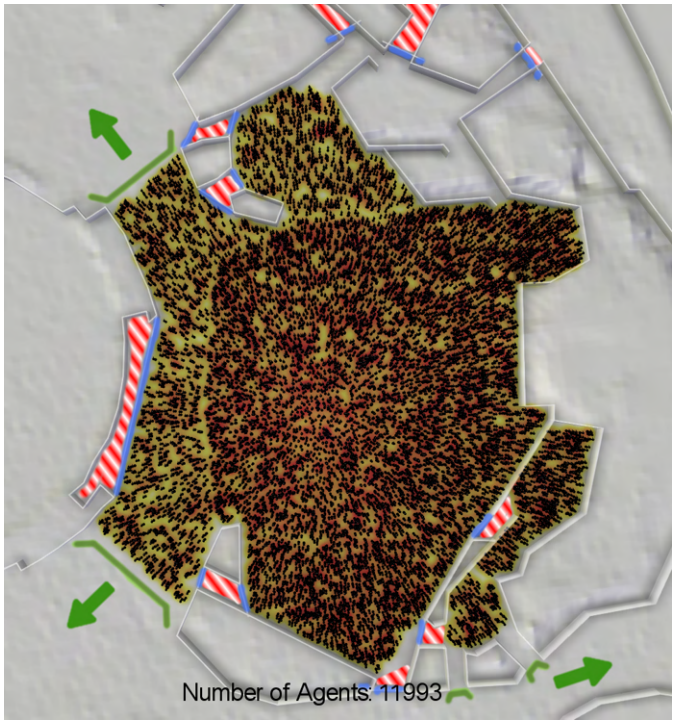


Figure 6.12: The setup of the second variation of the scenario: The new walls are colored in blue while the remaining exits are visualized by green arrows and green lines shaped like trapezoids with the longest side missing. All blocked small exits and small pockets are colored red striped.

This scenario was created based on the data of the first simulation. As can be seen in Figure 6.13 there are very high concentrations of pedestrians in front of the secondary exits. In comparison, the two main exits have a relatively low density. The question was: if the secondary exits are sealed off and only the main exits remain will this decrease the maximum density overall?

The result can be seen in Figure 6.14. While the maximum density at the outer exits hasn't changed compared to the previous simulation, the maximum density at the main exits has changed significantly. With the secondary exits closed the two main exits get totally overcrowded and have a maximum density that is far higher than 6 agents per m^2. In particular, the top exit seems to be very dangerous as it's corridor starts very wide and then narrows. This seams to increase the pressure on the agents at the front, because of the greater mass of agents pushing from behind. Also in the tests this exit was the one with the highest level of density.

Still some changes worked better than in the variant with all exits opened. Blocking off dead ends helped agents not getting stuck there.

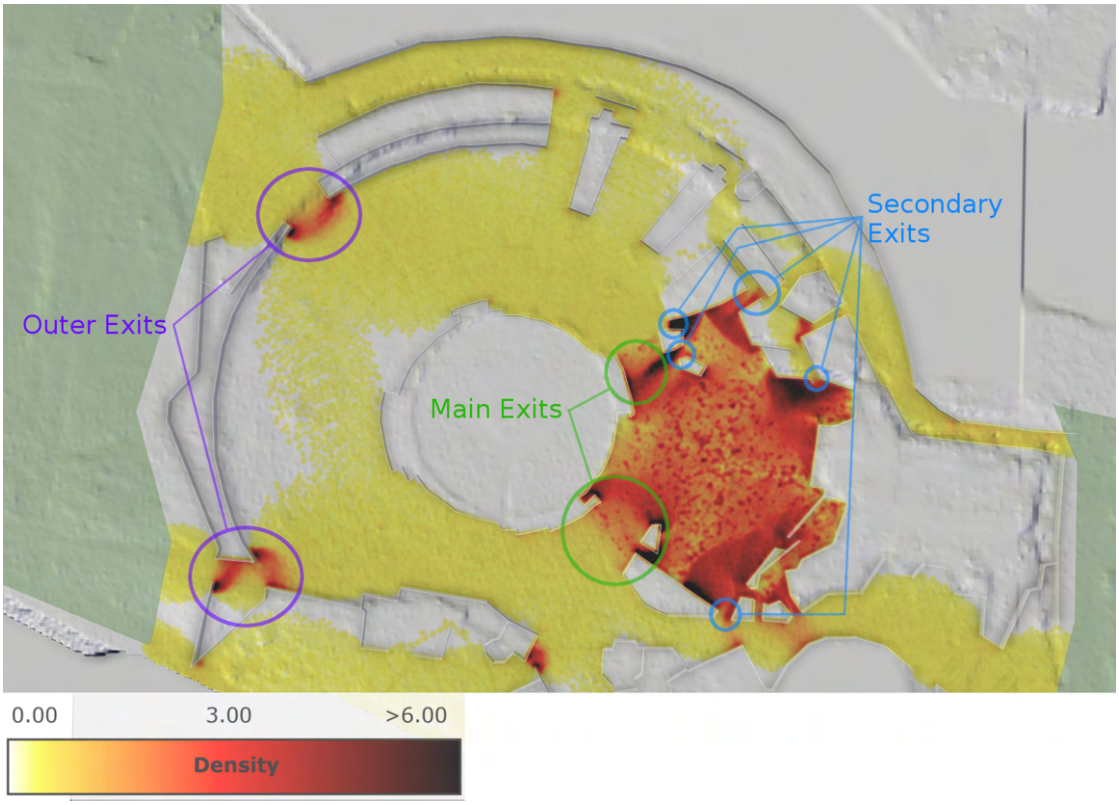


Figure 6.13: The maximum density of agents over the entirety of the first simulation. The unit of the legend is in agents per m^2

6.4 Performance

Performance is important for an interactive simulation, because the user has to be able to see the effects of her/his changes as fast as possible. In this section we will discuss performance measurements and results. Three values were measured: the overall computation time of a simulation step, the time it took the tactical model to finish its computation and the time it took the operational model to finish. With these values, the overhead may be computed. This is the time that is not spend within the models, but with tasks like preparing data or copying data. It is always very close to $0ms$. All tests were conducted on an Intel(R) Core(TM) i7-6700K @ 4 GHz (8CPUs) with 64GB of RAM and an NVIDIA GeForce GTX 1080 Ti running Windows 10 64-bit.

In the following a performance diagram will be used to visualize the performance data. To make the diagram more readable the data has been smoothed: Normally there are ten measurements per second, but they were reduced to one mean measurement per second. In the diagram the mean values are represented by the line and the lighter area behind the lines represent the standard deviation from the mean. Furthermore the horizontal axis represents the time in the simulation, while the vertical axis represents the time it

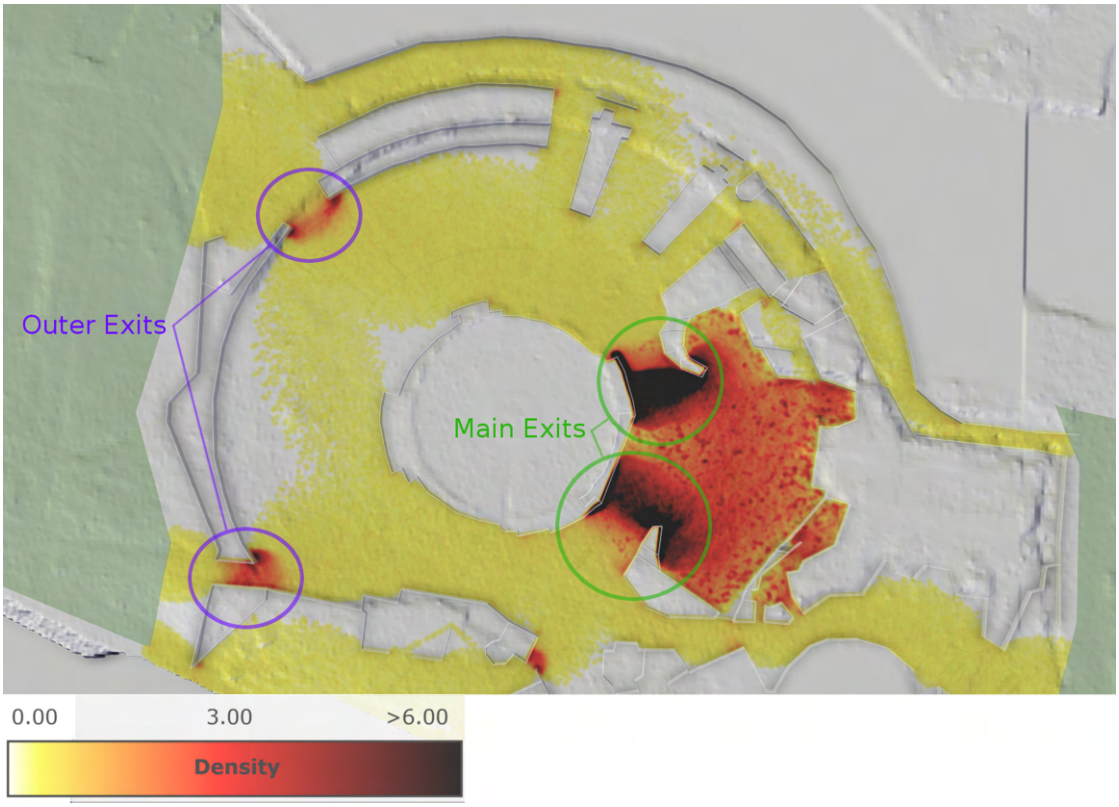


Figure 6.14: The maximum density of agents over the entirety of the second simulation. The unit of the legend is in agents per m^2

took to compute the next simulation step.

Because of the way the algorithm is constructed, we know the dependencies of the models we used. The tactical model is dependent on the (reachable) domain size and the number of agents. It uses a flood-filling algorithm that depends on the domain size (number of cells) and then has to compute the density of agents in that domain. The operational model depends on the density of agents and obstacles as every agent has to compute a path around other agents or obstacles that are near to the agent. In the worst case the performance is $\Theta(a^2)$ where a is the number of agents. This may occur in dense situations if each agent has to check every other agent. In the best case the algorithm has a performance of $\Theta(a)$, if all agents are so far away from each other that they do not have to check other agents at all.

6.4.1 RiMEA Performance

Most of the test cases' diagrams look very similar. Test Case 11's diagram is shown in Figure 6.15 and represents this class of diagrams. The tactical model takes up most of the computational time and the operational layer is computed very quickly. Both values

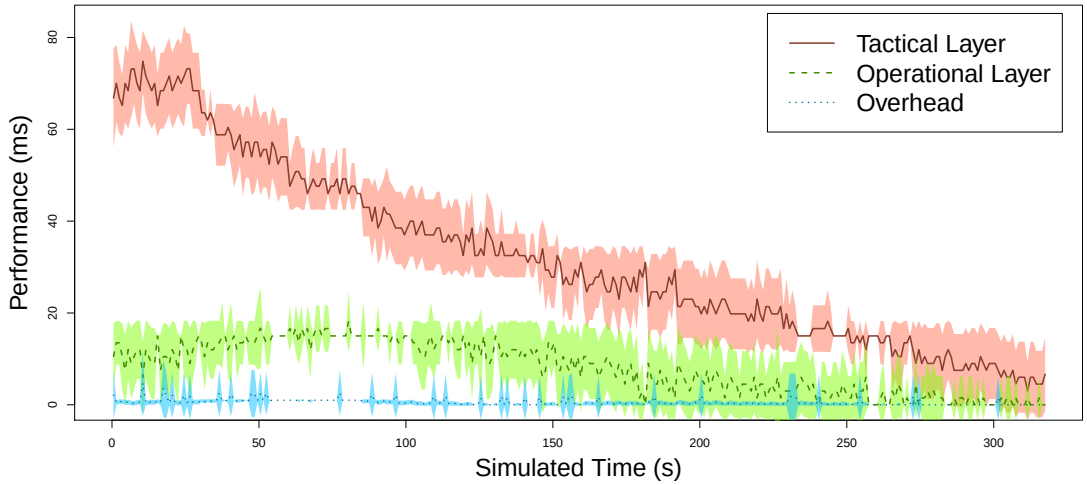


Figure 6.15: RiMEA Test Case 11 performance

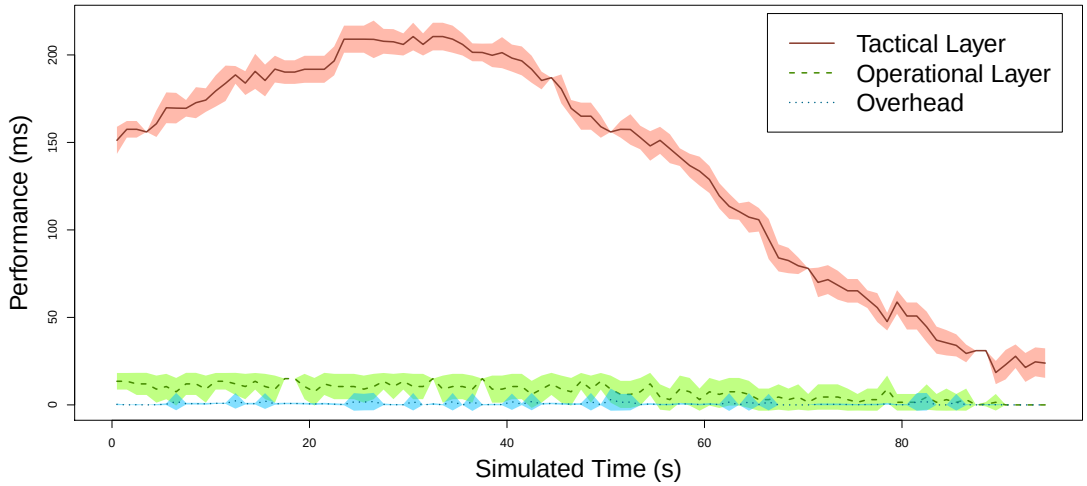


Figure 6.16: RiMEA Test Case 15 performance

go down eventually over time as less and less pedestrians are inside of the simulation.

Test Case 15 is a bit different as can be seen in Figure 6.16. The performance of the tactical layer peaks around 35 seconds as more pedestrians try to go around the corner. While the tactical layer has a clear peak, the operational layer stays the same throughout the simulation. This indicates that the tactical layer cannot handle dense simulations as well as the operational layer.

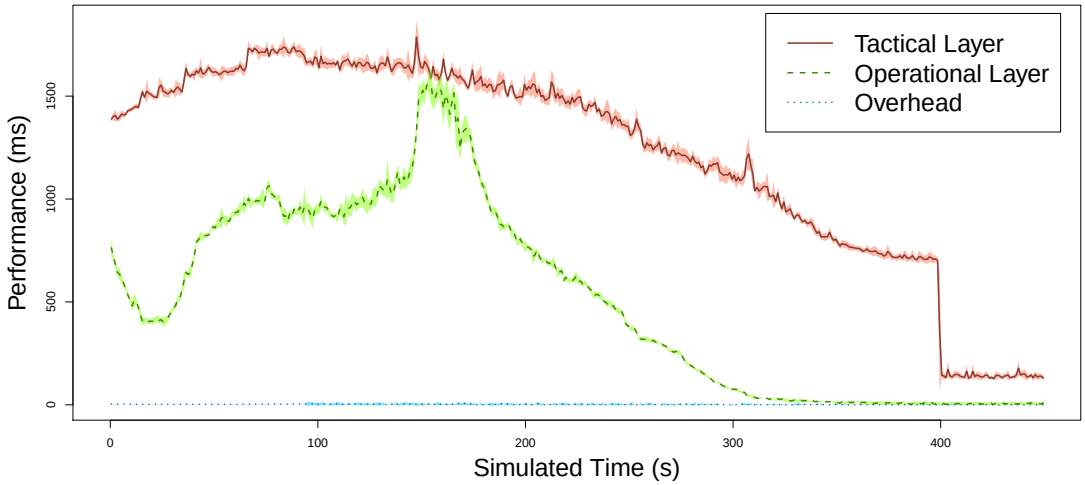


Figure 6.17: Performance graph of the real world scenario Tanzbrunnen

6.4.2 Real World Scenario Performance

The real world scenario is a very complex simulation with a big domain and a large number of pedestrians (11,993 agents). Because of this the performance is slower than real-time and the diagram of it can be seen in Figure 6.17. The slowest part of the simulation occurs between 150 and 175 seconds of simulation time. There the time it takes to compute a time step of 0.1 seconds exceeds three seconds. The simulation is down to zero agents after 400 seconds. This is the time when the tactical model's simulation time drops significantly.

The peak in the operational layer's performance curve might be due to a large congestion of pedestrians that happens at that time. This congestion dissolves over time as agents try to reach other, less used exits and more agents leave the simulation.

6.4.3 Performance Tests

In order to identify performance bottlenecks a few tests were conducted. The same simple setup was measured for different domain sizes and agent numbers. In this setup all agents start on the left side of a square domain of a certain size and have their target at the right side of this domain. Table 6.2 shows the detailed results of the test.

The following graphs show the performance of the tactical and operational layer in addition to the overhead. The overhead is produced by copying the data between the layers. As can be seen in Figure 6.18 the tactical layer is the most expensive part of the simulation for varying agent numbers on a small domain. It is also notable that the tactical layer seems to follow a near-linear relationship between the number of agents and the performance while the operational layer seems to have a quadratic growth rate.

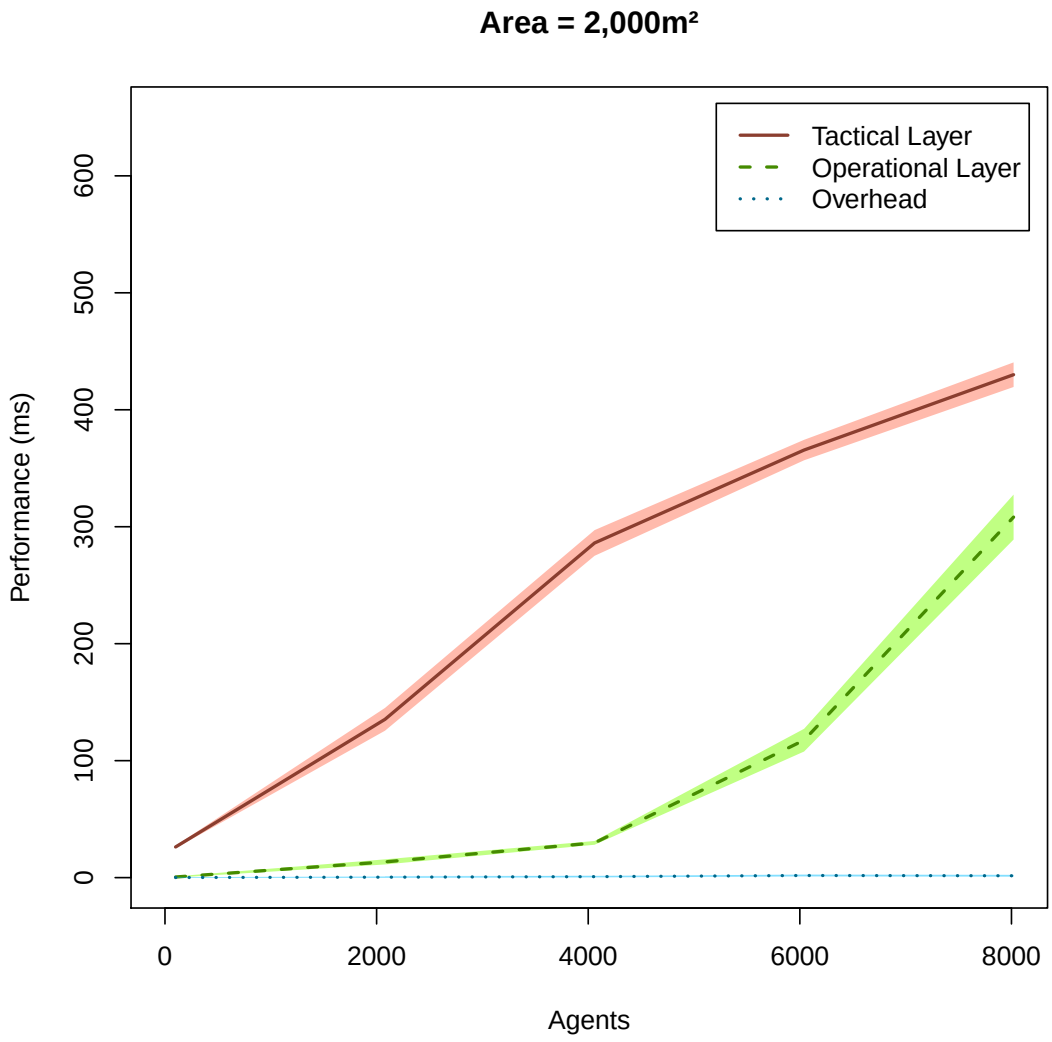


Figure 6.18: Performance graph of the mean performance for an area of $2,000m^2$

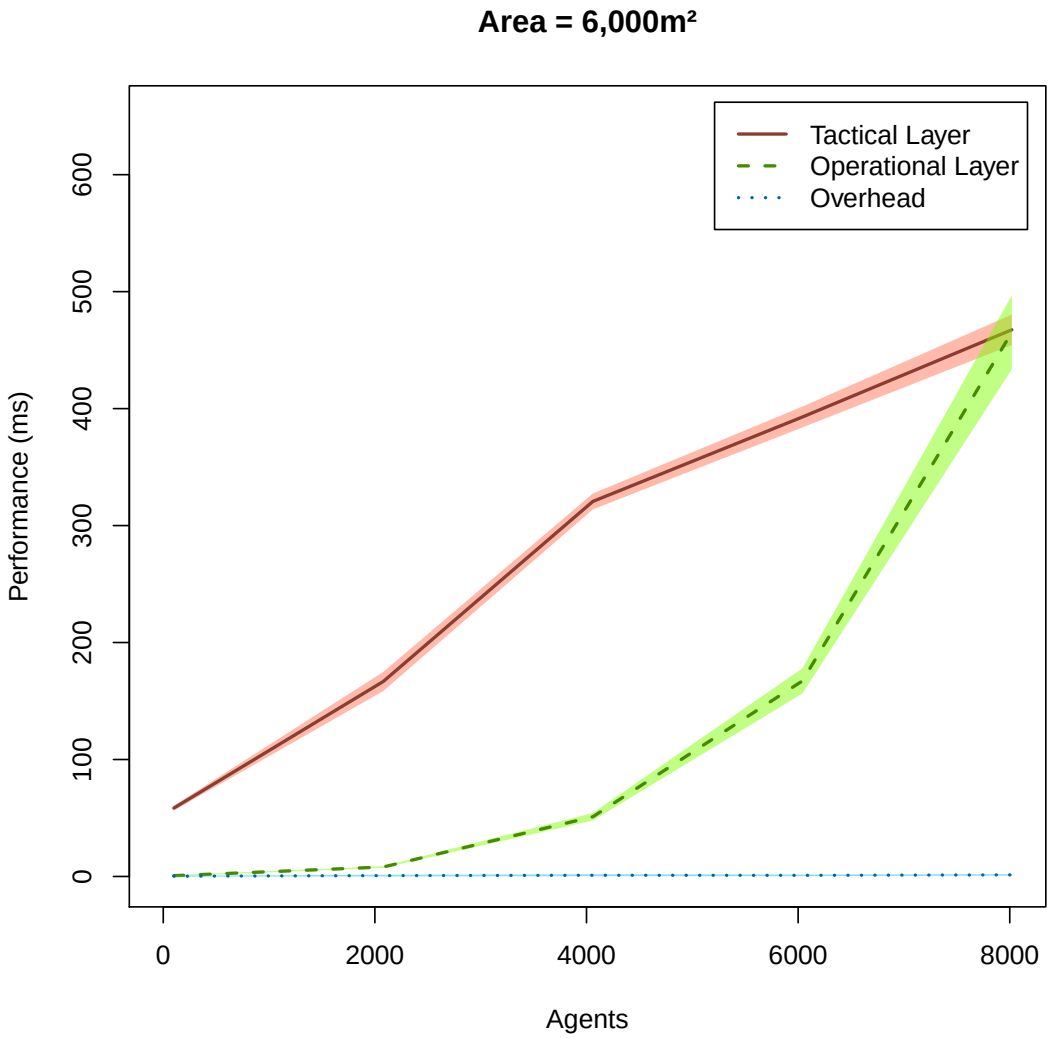


Figure 6.19: Performance graph of the mean performance for an area of $6,000m^2$

This holds true until the area reaches $6,000m^2$ (see Figure 6.19). At this size the initial cost of the tactical layer is by $32ms$ bigger than the cost for an area of $2,000m^2$, but the difference in cost for over $8,000$ agents is nearly the same, i.e., $38ms$. In contrast, the operational layer starts out with a difference of $0.1ms$ for 100 agents, but ends in a gap of $157ms$. This is also the point where the tactical and operational layer use up nearly equal amounts of computational time. After this point the operational layer takes more time to compute at high agent numbers than the tactical layer (see Figure 6.20).

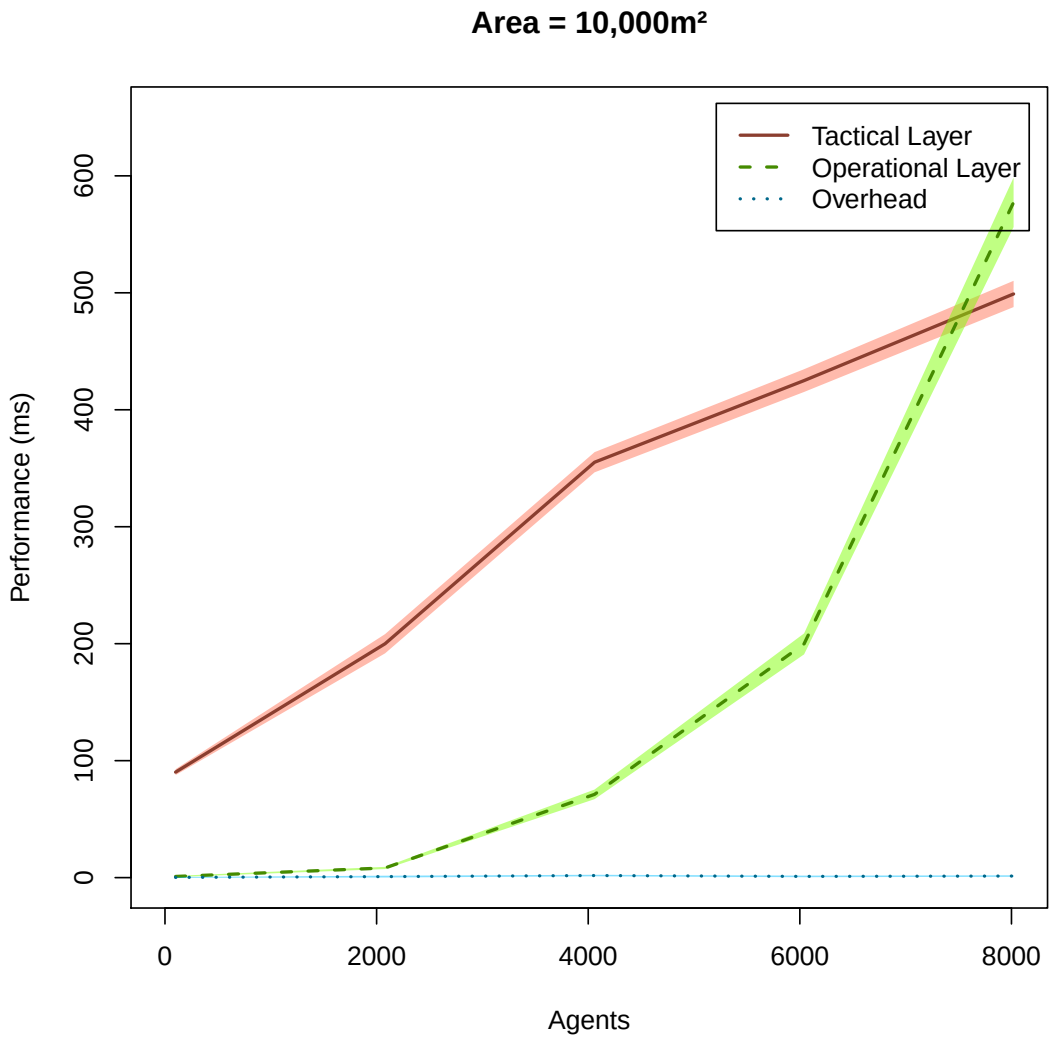


Figure 6.20: Performance graph of the mean performance for an area of $10,000m^2$

Agents	Domain Area	Tactical L.	Opera. L.	OH	Overall
100	$2,000m^2$	$26ms$	$1ms$	$0ms$	$27ms$
2080	$2,000m^2$	$135ms$	$13ms$	$0ms$	$149ms$
4060	$2,000m^2$	$286ms$	$30ms$	$1ms$	$317ms$
6040	$2,000m^2$	$366ms$	$117ms$	$2ms$	$485ms$
8020	$2,000m^2$	$430ms$	$308ms$	$2ms$	$740ms$
100	$4,000m^2$	$42ms$	$1ms$	$0ms$	$43ms$
2080	$4,000m^2$	$149ms$	$9ms$	$1ms$	$159ms$
4060	$4,000m^2$	$307ms$	$40ms$	$1ms$	$348ms$
6040	$4,000m^2$	$380ms$	$145ms$	$1ms$	$526ms$
8020	$4,000m^2$	$453ms$	$382ms$	$2ms$	$837ms$
100	$6,000m^2$	$58ms$	$1ms$	$0ms$	$59ms$
2080	$6,000m^2$	$167ms$	$8ms$	$1ms$	$176ms$
4060	$6,000m^2$	$321ms$	$51ms$	$1ms$	$373ms$
6040	$6,000m^2$	$393ms$	$167ms$	$1ms$	$561ms$
8020	$6,000m^2$	$468ms$	$465ms$	$1ms$	$934ms$
100	$8,000m^2$	$75ms$	$1ms$	$0ms$	$76ms$
2080	$8,000m^2$	$182ms$	$8ms$	$1ms$	$191ms$
4060	$8,000m^2$	$338ms$	$63ms$	$2ms$	$403ms$
6040	$8,000m^2$	$410ms$	$184ms$	$1ms$	$595ms$
8020	$8,000m^2$	$484ms$	$528ms$	$1ms$	$1013ms$
100	$10,000m^2$	$90ms$	$1ms$	$0ms$	$91ms$
2080	$10,000m^2$	$200ms$	$8ms$	$1ms$	$209ms$
4060	$10,000m^2$	$355ms$	$71ms$	$2ms$	$428ms$
6040	$10,000m^2$	$425ms$	$200ms$	$1ms$	$626ms$
8020	$10,000m^2$	$499ms$	$578ms$	$1ms$	$1078ms$

Table 6.2: Performance test with different domain sizes and agent counts. Performance measurements are the mean of a 100 steps simulation (10 Seconds). OH is Overhead

CHAPTER 7

Summary and Future Work

7.1 Summary

This thesis describes a plugin for Visdom that extends it by a pedestrian simulation module. The structure of a pedestrian simulation is discussed using the layered model by Hoogendoorn and Bovy [HB04]. For the tactical layer the quickest path model by Kretz et al. [KGH+11] is utilized and for the operational layer a modified version of the ORCA algorithm presented by Curtis and Manocha [CM14] is used. The strategic layer is modeled by the user in the form of target zones that the agents try to reach. Furthermore Visdom's emitter system is supported, which enables the user to add agents to the simulation as it goes on.

Two types of visualization are used: a coloring of the domain based on properties assigned to it and change in the color brightness of the agents based on their velocity so the user can identify slow moving agents more easily. Beside these two options, it is also possible to pick a single agent to look at its path so far and its current velocity. Another way to visualize data about agent's is via graphs. There are three types of graphs implemented. The first graph shows the velocity of selected agents over time, the second graph shows the number of agents in a certain area and the third graph shows the density of agents in a certain area.

The agents themselves are not models of humans, but glyphs that are designed to make it easier to identify agents and the direction they are currently facing. The usage of glyphs also removes unnecessary visual noise like the walking animation of models and their visual complexity.

Our model is validated with the help of the RiMEA test cases [rim]. While these do not have real world data to check against, they are designed in a way that a successful test adds to the credibility of the system. To be sure that the model can predict evacuations accurately, tests with real world data are necessary.

In order to validate the system in a real world scenario, an evacuation of the Tanzbrunnen in Cologne is simulated. Unfortunately we do not have any data to compare our results against, but this test shows that the system can handle real world scenarios.

We also discuss the computational performance of the system. We discover that a big bottleneck is the implementation of the tactical layer.

7.2 Future Work

The system has passed the RiMEA tests, but it is still advisable to validate it against real world data. This can be achieved by either taking existing real world data and model it in the simulation or by creating own studies and validate against these data. Existing data would be more desirable, because other systems might use the same data for validation, thous making both systems comparable.

Currently it is not possible to model buildings with multiple floors as the simulation is only 2.5D. To enable multi-storey buildings, it would be necessary to add portals that can connect two places with each other like a teleporter or a worm hole. This would allow the designer to arrange the floors next to each other and create portals at the staircases, which would connect the levels. A delay between agents entering and exiting portals could be added to simulated the pedestrians moving over the staircase. This feature would require the adaptation of both the operational and the tactical layer of the simulation.

Because the tactical layer is the slowest piece of the system, it should be improved. A possible optimization is the partial computation of the time field. As agents do not have to know all possible quickest paths to their destination it should be possible to limit the search space to the absolute minimum. It might be possible to implement an optimization that works similar to the A*-algorithm.

Another performance-heavy computation is the calculation the agent densities as this happens sequentially at the moment. It should be possible to rewrite the code to enable parallel code execution for each agent.

Currently all agents have global knowledge of the domain. This assumption is not very realistic as it includes real time data such as the current density of agents at each point of the domain. With bigger domains, the model might not accurately predict the evacuation any more. The solution to this problem would be to only use local information and old information the agent has observed, but cannot observe now (memory). This would open up new possibilities like placing signs to help agents to get to the exit quicker. To make the signs useful for planners they would have to have their own model that determines whether the agents even notice the signs or not.

The operational model can be improved as well. At the moment agents may change their velocities from one time step to the next from one extreme to another one. For example, the agent can go to the right and then suddenly move to the left with full speed,

without slowing down beforehand. The model behaves as if pedestrians have no mass and therefore no inertia. A small change in the setup of the ORCA halfplanes could change this. Two additional planes shall be added in such a way that the angle and the magnitude of velocity change is limited proportional to the current velocity. This should prevent the agents from accelerating as fast as they want and changing their direction of movement as strong as they want.

Another open topic is the height model. It is very simplistic at the moment and produces some unrealistic behavior. For example, agents are able to climb up steep cliffs which might not be desirable. A possible solution could be to handle steep slopes as impassable. Another flaw is that agents cannot fall down cliffs or holes at the moment. This should also be considered when reworking this system as well as using a better and more accurate speed model. This enhanced speed model should also takes the direction of movement (uphill, downhill) into account and assigns the speed accordingly.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BJR99] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The unified modeling language reference manual*. Addison-Wesley, December 1999.

[BT00] Srikanth Bandi and Daniel Thalmann. Path finding for human motion in virtual environments. *Computational Geometry*, 15(1-3):103–127, 2000.

[CGZM11] Sean Curtis, Stephen J Guy, Basim Zafar, and Dinesh Manocha. Virtual tawaf: A case study in simulating the behavior of dense, heterogeneous crowds. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference On*, pages 128–135. IEEE, 2011.

[CKB+16] Daniel Cornel, Artem Konev, Sadransky Bernhard, Horváth Zsolt, Brambilla Andrea, Viola Ivan, and Waser Jürgen. Composite flow maps. *Computer Graphics Forum*, 35(3):461–470, June 2016.

[CM14] Sean Curtis and Dinesh Manocha. Pedestrian simulation using geometric reasoning in velocity space. In *Pedestrian and Evacuation Dynamics 2012*, pages 875–890. Springer, 2014.

[CSM12] Sean Curtis, Jamie Snape, and Dinesh Manocha. Way portals: efficient multi-agent navigation with line-segment goals. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 15–22. ACM, 2012.

[DCSRO65] G. A. Dean (Commonwealth Scientific Research Organization). An analysis of the energy expenditure in level and grade walking. *Ergonomics*, 8(1):31–47, 1965.

[eig] Eigen. `http://eigen.tuxfamily.org`. [Online; accessed 10-November-2021].

[FC14] Bruno Fanini and Luigi Calori. 3D interactive visualization of crowd simulations at urban scale. In *9° Congresso Città e Territorio Virtuale,*

Roma, 2, 3 e 4 ottobre 2013, pages 276–284. Università degli Studi Roma Tre, 2014.

[FDCZ13] Shumin Feng, Ning Ding, Tao Chen, and Hui Zhang. Simulation of pedestrian flow based on cellular automata: A case of pedestrian crossing street at section in china. *Physica A: Statistical Mechanics and its Applications*, 392(13):2847–2859, 2013.

[Fra27] Philipp Frank. Über die Eikonalgleichung in allgemein anisotropen Medien. *Annalen der Physik*, 389(23):891–898, 1927.

[FS98] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[FT04] Taku Fujiyama and Nick Tyler. An explicit study on walking speeds of pedestrians on stairs. Japan Society of Civil Engineers/Transportation Research Board, USA, 2004.

[GCC+10] Stephen J Guy, Jatin Chhugani, Sean Curtis, Pradeep Dubey, Ming Lin, and Dinesh Manocha. Pledestrians: a least-effort approach to crowd simulation. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on computer animation*, pages 119–128. Eurographics Association, 2010.

[GCLM12] Stephen J Guy, Sean Curtis, Ming C Lin, and Dinesh Manocha. Least-effort trajectories lead to emergent crowd behaviors. *Physical review E*, 85(1):016110, 2012.

[GLM10] Stephen J Guy, Ming C Lin, and Dinesh Manocha. Modeling collision avoidance behavior for virtual humans. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 2*, pages 575–582. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[GWY+11] Hanqi Guo, Zuchao Wang, Bowen Yu, Huijing Zhao, and Xiaoru Yuan. Tripvista: Triple perspective visual trajectory analytics and its application on microscopic traffic data at a road intersection. In *Pacific Visualization Symposium (PacificVis)*, pages 163–170. IEEE, 2011.

[HB04] Serge P Hoogendoorn and Piet HL Bovy. Pedestrian route-choice and activity scheduling theory and models. *Transportation Research Part B: Methodological*, 38(2):169–190, 2004.

[HBK+10] Mario Höcker, Volker Berkhahn, Angelika Kneidl, Andre Borrmann, and Wolfram Klein. Graph-based approaches for simulating pedestrian dynamics in building models. *eWork and eBusiness in Architecture, Engineering and Construction*, pages 389–394, 2010.

[HGPA15] Oliver Handel, Ege Gümüş, Efthymios Papoutsis, and Julian Amann. Dynamic visualization of pedestrian simulation data. In *Forum Bauinf*, volume 2015, pages 1–9, 2015.

[HM95] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. *Physical review E*, 51(5):4282, 1995.

[IIC+96] A. Iserles, U.L.D.A. Iserles, D.G. Crighton, M.J. Ablowitz, S.H. Davis, E.J. Hinch, J. Ockendon, C.G. Crighton, and P.J. Olver. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 1996.

[JW07] Won-ki Jeong and Ross Whitaker. A fast eikonal equation solver for parallel systems. In *SIAM conference on Computational Science and Engineering*. Citeseer, 2007.

[KGH+11] Tobias Kretz, Andree Große, Stefan Hengst, Lukas Kautzsch, Andrej Pohlmann, and Peter Vortisch. Quickest paths in simulations of pedestrians. *Advances in Complex Systems*, 14(05):733–759, 2011.

[KLH14] Tobias Kretz, Karsten Lehmann, and Ingmar Hofsäß. User equilibrium route assignment for microscopic pedestrian simulation. *Advances in Complex Systems*, 17(02):1450010, 2014.

[rim] RiMEA e.V. - Richtlinie für Mikroskopische Entfluchtungs Analysen. `https://rimea.de/`. [Online; accessed 10-November-2021].

[RWF+13] Hrvoje Ribičić, Jürgen Waser, Raphael Fuchs, Günter Blöschl, and Eduard Gröller. Visual analysis and steering of flooding simulations. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):1062–1075, 2013.

[Set99] James Albert Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, volume 3. Cambridge university press, 1999.

[SSKB05] Armin Seyfried, Bernhard Steffen, Wolfram Klingsch, and Maik Boltes. The fundamental diagram of pedestrian movement revisited. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(10):P10002, 2005.

[SWR+12] Benjamin Schindler, Jürgen Waser, Hrvoje Ribičić, Raphael Fuchs, and Ronald Peikert. Multiverse data-flow control. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):1005–1019, 2012.

[tan] Tanzbrunnen Köln. `https://koelncongress.de/locations/tanzbrunnen-koeln/`. [Online; accessed 10-November-2021].

[VDBGLM11] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[vis] Visdom - Integrated Visualization. `http://visdom.at/`. [Online; accessed 10-November-2021].

[vrv] VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH. `http://www.vrvis.at/`. [Online; accessed 10-November-2021].

[Wei93] Ulrich Weidmann. Transporttechnik der fußgänger: transporttechnische eigenschaften des fußgängerverkehrs, literaturauswertung. *IVT Schriftenreihe*, 90, 1993.