On Weighted Regions and Social Crowds:
Autonomous-agent Navigation
in Virtual Worlds

NORMAN JAKLIN

© 2016 Norman Jaklin

Cover design by Norman Jaklin

ISBN: 978-90-393-6521-2

On Weighted Regions and Social Crowds: Autonomous-agent Navigation in Virtual Worlds

Gewogen Gebieden en Sociale Menigtes:
Navigatie voor Autonome Karakters in Virtuele Werelden

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op maandag 18 april 2016 des middags te 12.45 uur

door

Norman Simon Jaklin

geboren op 9 januari 1982
te Keulen, Duitsland

Promotor:	Prof. dr. M.J. van Kreveld
Copromotor:	Dr. R.J. Geraerts

The work described in this thesis has been supported by the Dutch national program COMMIT/.

CONTENTS

1 Introduction **3**
1.1 Virtual worlds 3
1.2 A five-level agent-navigation planning hierarchy 4
1.2.1 Related frameworks 6
1.2.2 High-level planning 7
1.2.3 Global-route planning 8
1.2.4 Route following 8
1.2.5 Local movement 9
1.2.6 Animation 9
1.3 Main contributions of this thesis 10
1.3.1 Contributions Part I 11
1.3.2 Contributions Part II 11
1.3.3 Contributions Part III 12

2 Motivation and preliminaries **13**
2.1 Representing traversable space in virtual worlds 14
2.1.1 Grids 16
2.1.2 Waypoint graphs and road maps 18
2.1.3 Navigation meshes 21
2.2 A^* and its variants 24
2.3 The weighted region problem 28
2.3.1 Definition and first approximation algorithm (Mitchell and Papadimitriou, 1991) 28
2.3.2 Unsolvability in ACMQ (De Carufel et al. 2012) 29
2.3.3 Existing approximation algorithms 29
2.4 Crowd simulation 31

I Path Planning in Weighted Regions **33**

3 Grid-paths in weighted regions **35**
3.1 Related work 36

3.2 Region-homotopic paths 37
3.3 Path-length analysis of 8-neighbor grid paths in grid-aligned regions 39

4 Vertex-based pruning (VBP): A Hybrid Method 45
4.1 The Steiner-point method (SPM) by Aleksandrov et al. 1998 46
4.2 The VBP method 48
  4.2.1 Triangle-based pruning 49
  4.2.2 Edge-based pruning 50
  4.2.3 Vertex-based pruning 51
4.3 Experiments 52
  4.3.1 The tested scenes 53
  4.3.2 VBP vs. SPM in small scenes 55
  4.3.3 Empirical analysis of path differences 59
  4.3.4 VBP vs. SPM in a large scene 60

5 Conclusion Part I 65

II Path Following in Weighted Regions 69

6 The MIRAN method 71
6.1 Related work 72
6.2 Preliminaries 74
6.3 Method details 75
  6.3.1 Computing a reference point 77
  6.3.2 Computing the candidate attraction points 78
  6.3.3 Choosing the attraction point 80
  6.3.4 Moving the agent 81
  6.3.5 Proof of correctness 82
6.4 Experiments 85
  6.4.1 The tested scenes 86
  6.4.2 MIRAN paths vs. IRM paths 86
  6.4.3 The impact of σ and d on path-following behavior 87
  6.4.4 Performance 93
6.5 Limitations 97

7 MIRAN for disc-based agents 101
7.1 Candidate attraction points for disc-based agents 102
7.2 Weight function for disc-based agents 105
  7.2.1 General concept 106
  7.2.2 Computation of interval points 106

7.3 Results and future extensions 109

8 Conclusion Part II 113

III Crowd Simulation 117

9 The Stream model: coordinating dense crowds 119
9.1 Related work 120
9.2 Preliminaries 123
9.2.1 Agent representation 123
9.2.2 Overview of the Stream model 123
9.3 Stream behavior 124
9.3.1 Computing local density information 125
9.3.2 The perceived stream velocity 127
9.3.2.1 Perceiving a single agent 127
9.3.2.2 Perceiving the local stream 128
9.3.3 Incentive 130
9.3.3.1 Computing the incentive 130
9.3.3.2 Using the incentive 131
9.4 Experiments 132
9.4.1 Scenarios 132
9.4.2 Modeling various agent profiles 133
9.4.3 Comparing different collision-avoidance methods 135
9.4.4 Testing the effect of streams 137
9.4.5 Performance 138
9.5 Limitations 140

10 Social Group Behavior 143
10.1 Related work 144
10.2 Preliminaries 146
10.2.1 Basic settings 146
10.2.2 Overview of the SGN method 147
10.2.2.1 SGN initialization 147
10.2.2.2 The SGN simulation loop 148
10.2.3 Integration of SGN into a crowd simulation framework . . . 149
10.3 The *Social Groups and Navigation* method 150
10.3.1 Leader and last member 150
10.3.2 Coherence and sociality 150
10.3.3 Coordination mode 151
10.3.4 Group-walking mode 152
10.3.5 Collision avoidance 153

10.3.6 Social-force model 154
10.3.6.1 Physical-contact force with another agent 154
10.3.6.2 Physical-contact force with obstacles 155
10.3.6.3 Group force 155
10.3.6.4 Differences to the original model 156
10.4 Experiments 157
10.4.1 Experimental setup 157
10.4.2 Effects of SGN on coherence and sociality 161
10.4.3 Evacuation times 161
10.4.4 Performance 163
10.5 Limitations 165

11 Combining it all: the ECM crowd-simulation framework 167
11.1 The Explicit Corridor Map 168
11.2 Implementation details 171
11.2.1 Input and output 171
11.2.2 Computing navigation meshes 172
11.2.3 Algorithms of the planning hierarchy 174
11.2.4 Architecture 174
11.2.5 Visibility checks 175
11.3 Experiments 177
11.3.1 Computing visibility polygons 179
11.3.2 Computing indicative routes 180
11.3.3 Crowd simulation 180
11.4 Conclusion and future work 182

12 Conclusion Part III 185

13 Overall conclusion and future research 189

Bibliography 191

Samenvatting 203

Acknowledgements 205

Curriculum Vitae 207

"If the path before you is clear, you're probably on someone else's."

- Joseph Campbell

CHAPTER 1

INTRODUCTION

1.1 | VIRTUAL WORLDS

Virtual environments have gained in importance in many aspects of the world we live in today. With the rise of the internet, the possibility to connect people and to transfer data on a global scale within milliseconds, the virtualization of the real world as well as imaginative fictional worlds have become an important aspect of modern life.

When most people hear terms such as *virtual world* or *virtual environment*, they tend to think of immersive worlds for entertainment purposes such as movies, digital games, or online communities. Such entertainment applications are most popular and ubiquitous in today's society. However, they are by far not the only areas in which virtual worlds have become increasingly important. Virtual worlds are also important in non-entertainment applications such as training and education software, simulations of mass events, evacuation scenarios, human factor analysis, and urban city planning. Training and education software itself comprises various application areas, ranging from teaching children with the help of virtual characters, to training policemen and firefighters, or training soldiers in virtual environments for military operations. Other applications such as online mapping services (e.g. *Open Street Map*, *Google Street View*, or *Mapillary*) are also examples of virtualizing the real world. Adding virtual humans and crowds to such applications can increase their believability, usability and overall impact on society, potentially leading to the creation of complete virtual copies of real-world locations.

Virtual worlds are usually not environments of vast empty space, but rather highly dynamic and lively. They do not only contain huge numbers of user-controlled characters or avatars, but also autonomous virtual humans or similar entities that navigate through the virtual environment. Thus, a key aspect of creating an immersive virtual world is the development of algorithms that handle the navigation of such entities. This involves the creation of believable paths that are smooth, do not contain unnecessary detours, keep clearance from obstacles, respect terrain and region information, and avoid collisions with other moving entities. Furthermore, it involves the coordination of large virtual crowds in both sparse and dense situations, and the generation of social behavior among virtual groups.

Over the past few decades, advances in computer hardware and algorithms have radically changed and shaped the overall appearance of virtual worlds. Aspects such as graphics, modeling or physics simulation have received much attention, especially in the movie and game industries. This led to a wide range of novel techniques to generate believable pictures, $3D$ models and animations. By contrast, the *paths* traversed by virtual characters are often not visually convincing in the aforementioned application areas. This problem is less apparent in movies, games, or simulations that do not allow characters to deviate from their predetermined or scripted paths. For these applications, a designer can manually create believable paths that are of high quality. By contrast, applications that are more flexible and allow unpredictable interactions among characters rely on algorithms to handle path planning. In serious simulations that try to mimic real-life behavior of humans, flexibility and a potentially infinite number of interactions between characters is even more important. However, even state-of-the-art algorithms and crowd-simulation models struggle with particular computational tasks, and the range of possible character behaviors is still limited up to the present day.

In this thesis, we will focus on three computational tasks, with which state-of-the-art algorithms still struggle: Region-based path planning, region-based path following, and coordinating dense virtual crowds and social groups. We will show why these tasks are difficult to solve with existing algorithms when using grids or graph-based representations of the traversable space in a virtual environment. Furthermore, we will show that these tasks can be solved efficiently on a surface-based representation when using novel methods. These novel methods on region-based planning and coordinating crowds and social groups will be presented in detail, and they form the main contributions of this thesis.

Throughout this work, we mainly refer to an autonomously moving entity as an *agent* because all novel algorithms we will discuss are agent-based methods as part of a larger agent-based crowd simulation framework. We will sometimes use the term *character*, when the context is entertainment games or educational games that feature autonomous virtual humans.

1.2 | A FIVE-LEVEL AGENT-NAVIGATION PLANNING HIERARCHY

All novel algorithms that we present in this thesis are embedded in a five-level planning hierarchy [56, 142]. This hierarchy enables the simulation and combination of particular aspects of agent navigation to generate complex and thus more believable agent behaviors. In the past, the term *path planning* has been widely used as a variant of the more abstract field of *motion planning* when discussing autonomous-agent navigation. From a modern perspective, however, path planning is only one aspect of autonomous-agent navigation. In the context of modern simulations and

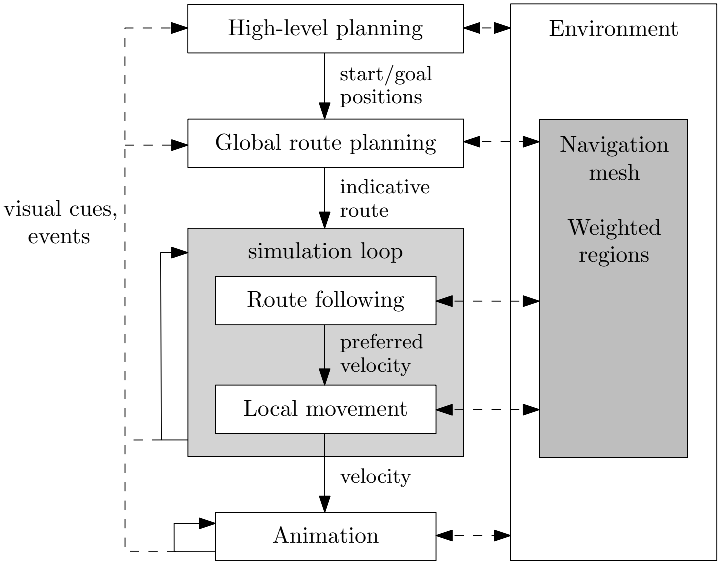


FIGURE 1.1: A five-level agent-navigation planning hierarchy.

gaming applications, planning a path is only the first step to simulate believable navigation behavior for a virtual agent. An agent also has to follow the path and react to dynamic changes and events it encounters on its way. For instance, such an event can be the encounter with other autonomous agents, for which collisions need to be avoided, and dynamic changes in crowd density and flow need to be handled appropriately. As such, an agent-navigation and crowd-simulation system comprises more than just path planning, which highlights the need for multiple levels of planning.

In this section, we describe the five levels of planning that we propose for modern agent-navigation and crowd-simulation systems. An illustration of such a planning hierarchy is given in Figure 1.1. Note that planning in such a hierarchy is not purely serial. For instance, when an agent has reached its goal position, it returns to the global-planning level or the higher-planning level to determine its next action. An agent might also be forced to re-plan its global path while traversing it, e.g. when crowd density increases or parts of its path are unexpectedly blocked by a dynamic obstacle. As such, the levels of the proposed hierarchy need to be able to communicate with all other levels at any time.

Note that the proposed hierarchy does not depend on a particular navigation-mesh data structure, but rather serves as a general paradigm for tackling problems related to autonomous-agent navigation. The work in this thesis, while strongly based on the proposed hierarchy, is described in an abstract way that is independent of the particular data structures that are being used. Still, all algorithms presented in this thesis have been developed within the *Explicit Corridor Map* (ECM) framework that was presented by Geraerts [31]. We will give a brief overview of the ECM in Chapter 2, where we discuss navigation meshes in general. For a more detailed discussion of the ECM, together with more practical implementation details of all novel algorithms that we present in this thesis, we refer the interested reader to Chapter 11. We will now start with discussing related frameworks and the most important related work for each level.

The remainder of this section is based on the following publication:

[56] N. Jaklin, W. van Toll, and R. Geraerts. Way to go – a framework for multi-level planning in games. In *Proceedings of the 3rd International Planning in Games Workshop (ICAPS'13 | PG2013)*, pages 11–14, 2013.

[142] W. van Toll, N. Jaklin, and R. Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN*, 2015.

1.2.1 | RELATED FRAMEWORKS

Several frameworks exist for autonomous-agent and crowd simulation. Some of these are rooted in the agent-navigation research community, while others are commercial products for the industry.

A wide range of frameworks and software packages exists for serious-gaming applications such as safety training and evacuation studies, e.g. Legion[1], Massive[2], MassMotion[3], Pedestrian Dynamics[4], SimWalk[5], Steps[6], or VisWalk[7] Most of these require manual work from the user because they do not automatically compute a navigation data structure.

In the entertainment industry, the *Unity3D* game engine[8] has recently adopted the *Recast*[9] and *Detour* systems for automatic navigation meshes and agent simulation. Golaem Crowd[10] has shifted its focus to high-quality plug-and-play crowds for entertainment applications. *Massive* is another software package that has been used for crowd generation in various movies and entertainment games.

---

[1] Legion; `http://www.legion.com/`; accessed January 13, 2016.

[2] Massive; `http://www.massivesoftware.com/`; accessed January 13, 2016.

[3] Mass Motion; Oasys Software; `http://www.oasys-software.com/`; accessed January 13, 2016.

[4] Pedestrian Dynamics; Incontrol Simulation Solutions; `http://www.pedestrian-dynamics.com/`; accessed January 13, 2016.

[5] Sim Walk; `http://www.simwalk.com/`; accessed January 13, 2016.

[6] Steps; `http://www.steps.mottmac.com/`; accessed January 13, 2016.

[7] VisWalk; PTV Group; `http://vision-traffic.ptvgroup.com/`; accessed January 13, 2016.

[8] Unity3D; `http://www.unity3d.com/`; accessed January 13, 2016.

[9] Recast navigation; M. Mononen; `https://github.com/recastnavigation/recastnavigation`; accessed January 13, 2016.

[10] Golaem Crowd; `http://www.golaem.com/`; accessed 13 January, 2016.

In the research community, *SteerSuite* [120] has been proposed for evaluating steering methods. *ADAPT* [119] is a platform for developing agent behavior with an emphasis on animation. *SimPed* and *NOMAD* are models for passenger flows, based on real-world observations [19, 49]. The *Menge* framework by Curtis et al. [17] uses a similar subdivision into levels as the ECM framework, on which the work in this thesis is based. *Menge* and the ECM framework were developed independently, and both are loosely based on a subdivision of tasks that has been presented by Funge et al. [29] and Ulicny and Thalmann [132] in the context of cognitive and behavioral modeling. Curtis et al. describe the subtasks as mathematical functions from an abstract point of view, and they focus on four main subtasks: goal selection, plan computation, plan adaptation, and motion synthesis. In general, these subtasks correspond to the same subtasks that are used in the ECM framework. However, the ECM framework subdivides the plan adaptation task further into path following and local movement, and it emphasizes the need of an indicative route as a rough guidance path that is computed in the global-planning level. Furthermore, *Menge* is not based on a particular representation of the environment. It treats the choice for a particular environment representation as a black box, whereas the ECM framework focuses on an ECM as its underlying data structure with its particular advantages (such as the ease of computing arbitrary-clearance paths).

1.2.2 | HIGH-LEVEL PLANNING

At the top of the hierarchy, *high-level planning* (level 5) translates the desired *semantic* behavior of an agent to a *geometric* path-planning problem. First, an agent's abstract task such as 'take the train to work' can be converted to a list of more concrete tasks, e.g. 'go to the train station, buy a train ticket, go to the correct platform, enter the train', and so on. Based on this plan, an agent should compute a list of goal positions. Such a list is usually ordered, and the goal positions are usually specific points in the environment [118]. Interesting open research questions in this area are how to simulate the behavior of agents that have no clear ordering of goals, and goals that are more general than specific geometric positions in the environment. An example situation would be a strolling agent in a shopping mall with no specific goal other than looking around and maybe spontaneously deciding to buy something here and there.

High-level planning is a research topic of its own, involving techniques such as *STRIPS* [25] and *Hierarchical Task Networks* [68]. Cognitive decision-making models have also been applied to crowd simulation [101, 118]. In the context of this thesis, we focus on geometric planning, and we deliberately treat high-level planning as a black box.

1.2.3 | GLOBAL-ROUTE PLANNING

On level 4 of the hierarchy, *global route planning* uses an agent's current goal position to compute a global path through the environment. Following Karamouzas et al. [63], we refer to such a global path as an *indicative route* because it is a preliminary indication of an agent's final trajectory. Compared to using a path that is followed exactly, using an indicative route that is only roughly followed yields greater flexibility in the lower levels of the planning hierarchy. In theory, an indicative route can be any curve through the walkable space. In practice, it is usually a sequence of connected straight-line segments.

Commonly desired properties for an indicative route are an overall short curve length, the avoidance of unnecessary detours, and clearance from obstacles in the environment [31]. However, indicative routes can also be computed based on other criteria. For instance, local crowd-density information can be used to make agents prefer routes that are less congested, which consequently spreads a crowd over multiple routes [141]. Visibility information can also be used to compute indicative routes along which an agent is not seen by other agents [34]. For a more extensive overview of data structures and algorithms for global-route planning, we refer the interested reader to Chapter 2.

Another desired property that is particularly important for the contributions made in this thesis is the avoidance of undesired regions or terrain in the environment. For example, a pedestrian might prefer walking on a sidewalk while avoiding roads, puddles or muddy terrain. Planning and following approximate cost-optimal global routes, which are based on an agent's individual region preferences, is one of the major research challenges we tackle in this thesis. We refer the interested reader to Chapters 3 through 5 for path **planning** in weighted regions.

1.2.4 | ROUTE FOLLOWING

On level 3 of the hierarchy, an agent's route-following behavior is being handled. The purpose of this level is to compute a *preferred velocity* for an agent in each time step of the simulation.

Many researchers and software systems do not treat route following as a separate level. However, we believe that a clear separation between route planning and route following is crucial for the real-time simulation of autonomous agents. The main reasons for this are as follows: An indicative route is generally not smooth and natural looking. Furthermore, modern collision-avoidance methods (see Section 1.2.5) require a preferred velocity as an input. Unless the agent can walk towards its goal in a straight line, we need an algorithm that chooses a desired walking direction at any point in time. Due to collision-avoidance maneuvers, an agent is usually not located exactly on its indicative route. In addition, a route-following algorithm can take region preferences into account, which is an aspect that is particularly relevant in this thesis. We refer the interested reader to Chapters 6 through 8 for path following in weighted regions.

Among the first authors to acknowledge route following as a separate step was Reynolds [115], who introduced steering behaviors for autonomous agents as an extension to his seminal work on simulating flocks of birds and schools of fish [114]. The *Indicative Route Method* (IRM) by Karamouzas and Overmars [63] uses an attraction point that moves along an indicative route, which is then combined with a force-based model to make an agent approach its attraction point in each step of the simulation. Our novel path-following method that we present in Chapter 6 is based on the same idea, but extends and improves on the IRM by simulating region preferences and giving the user control over how closely an indicative route should be followed.

1.2.5 | LOCAL MOVEMENT

On level 2 of the hierarchy, *local movement* enables an agent to temporarily deviate from its route to resolve local events such as collisions with other agents. An agent may also coordinate its direction and speed with the agents in its vicinity, e.g. to simulate social groups of agents [65, 115].

In early collision-avoidance methods, agents exerted attractive and repulsive *forces*, and physical laws of motion yielded new velocities for each agent [45, 115]. A disadvantage of these methods is that they are inherently reactive.

More recent algorithms are based on *velocity selection* [64, 88, 133]. These algorithms let an agent pick the best speed and direction from a sampled range of options (i.e. candidate velocities), based on a cost function. The cost of a candidate velocity depends on the difference to an agent's preferred velocity, and on the predicted collisions with other agents when choosing the candidate velocity.

Most local collision avoidance models cannot solve scenarios that feature high crowd densities. Furthermore, they handle agents as individuals with no social bond. These are two important aspects in the context of this thesis: Improving the coordination between agents in high-density situations and simulating social-group behavior are open research challenges that we tackle in Chapters 9 and 10 of this thesis, respectively.

1.2.6 | ANIMATION

Finally, the animation level 1 produces visual output by animating and translating an agent's $3D$ model in the environment [7]. Generating smooth and physically plausible animations without requiring a large database of motion clips is an active research topic that is outside the scope of this thesis.

Note that an animation and the overall simulation usually have different framerates. Crowd simulations often use a fixed timestep of $0.1s$ (i.e. 10 frames per second) [63], whereas smooth animation requires a much higher framerate. We therefore assume that the animation level uses a separate loop. Whenever a simulation step ends and all agent positions have been updated within the simulation model, the animation level of the planning hierarchy is notified and generates animations that take the agents to their new positions over the next few animation frames.

This concludes the description of the underlying planning hierarchy that we assume throughout the remainder of this thesis. At the beginning of each subsequent chapter, we will highlight the level(s) in which the described work is embedded. We now give an overview of the main contributions of this thesis.

1.3 | Main contributions of this thesis

The main contributions of this thesis comprise novel real-time algorithms for virtual-agent navigation in simulations and gaming applications, as well as theoretical results related to path planning, path following and the navigation of autonomous virtual crowds.

All novel algorithms are designed in a modular way. This means that they can be independently plugged into a larger crowd-navigation framework that follows a hierarchical structure as described in the previous section. Some of the proposed algorithms are directly linked to each other in such a way that the output of one algorithm can be used as an input for another. For instance, the *Vertex-based Pruning* method as described in Chapter 4 computes an indicative route from a given start to a given goal position. The *Modified Indicative Routes and Navigation* method as described in Chapter 6 takes such an indicative route as an input and computes a smooth trajectory for an agent based on the given indicative route as a rough guidance path.

All algorithms were developed within the *Explicit Corridor Map* (ECM) framework [31, 56], which follows the proposed hierarchical structure of five different planning levels. However, specific properties and features of the underlying ECM navigation mesh are used only for ease of implementation, and they are not fundamental parts of the described algorithms. As such, the algorithms can be used in any crowd-navigation and simulation framework that handles path planning, path following, and local avoidance-behavior as separate steps, e.g. the framework by Curtis et al. [17].

Overall, the thesis is subdivided into three distinct parts, each of them focusing on one of the center levels of the five-level crowd-navigation hierarchy. Part I focuses on path **planning** in weighted regions for an individual agent, Part II focuses on path **following** in weighted regions for an individual agent, and Part III focuses on simulating the navigational aspects of multiple agents in small social groups and large crowds.

1.3.1 | Contributions Part I

In the first part of this thesis (Chapters 3 through 5), we address the problem of computing a path in a $2D$ virtual environment that features multiple weighted regions. Such regions either resemble a particular terrain type such as *road*, *grass*, or *water*, or they resemble a psychological influence on an agent's navigation, such as *dangerous* or *attractive*. The weight for each region encodes how difficult it is for a virtual agent to traverse it. Given an agent with a set of particular region preferences, the task is to compute a path from a start position to a goal position that minimizes the weighted Euclidean length among all possible paths. From a computational geometry perspective, this is known as the *Weighted Region Problem*, and we discuss it in Section 2.3 in more detail.

In Chapter 3, we analyze 8-neighbor grid paths. To this end, we define the term *region-homotopy* and contribute to existing literature on path-cost analyses. We prove an upper bound on the costs for grid paths under the assumption that all regions are aligned with the grid. To the best of our knowledge, this is the first path-cost analysis for grid paths in weighted regions. In Chapter 4, we present a new path planning method called *Vertex-based Pruning* (VBP) that combines an A^*-search on a grid with an existing ϵ-approximation method. We show that VBP decreases the computation times of ϵ-approximate paths in weighted regions of existing methods when the environment is sufficiently large, while still guaranteeing acceptable overall path costs. As such, VBP is a next step towards *real-time* path planning in weighted regions for simulations and gaming applications. The paths computed with VBP can be used as *indicative routes* that form a rough guideline of an agent's preferred path through the environment. In Chapter 5, we draw conclusions on the path-planning part of this thesis.

1.3.2 | Contributions Part II

In the second part of this thesis (Chapters 6 through 8), we present a new path-following method named *Modified Indicative Routes and Navigation* (MIRAN). MIRAN takes as an input an indicative route in a scene with multiple weighted regions. Such a path can be either computed with our new VBP method, or with a simpler path planning approach, or it can be manually drawn by a user. The MIRAN method samples the given route and makes an agent follow it based on its region preferences to create smooth curves. MIRAN is the first method to allow path-following behavior that is based on region preferences for real-time simulations and gaming applications.

In Chapter 6, the agent is assumed to be represented as a point. Two user-controlled parameters can be set to determine how densely the indicative route should be sampled and how far ahead along the indicative route the agent is allowed to see. The latter determines how much of the route the agent is allowed to shortcut when following it. In Chapter 7, we extend the MIRAN method to handle agents that are represented as discs with an arbitrary radius. We adjust the computation of attraction points and the underlying weight-function to account for the agent's radius. In addition, we discuss how we can extend existing collision avoidance methods to also account for an agent's region-preferences as future work. In Chapter 8, we draw conclusions on the path following part of this thesis.

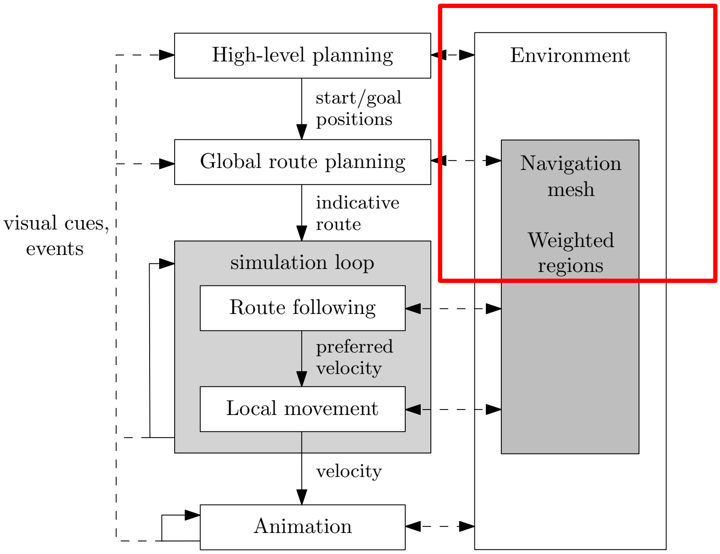
1.3.3 | Contributions Part III

In the third part of this thesis (Chapters 9 through 12), we present a new crowd simulation model and a new method to simulate the walking behavior of small social groups. Similar to the contributions made in Part I and Part II, these contributions are agent-based, but they can be used to simulate large numbers of virtual agents at interactive rates on modern hardware.

In Chapter 9, we introduce a novel crowd simulation model, the *Stream* model. It combines the advantages of agent-based and flow-based models using only local rules. The Stream model is the first to significantly improve the coordination and crowd flow in arbitrarily-changing density scenarios for real-time simulations and gaming applications. In Chapter 10, we present a new method called *Social Groups and Navigation* (SGN). It can be used to simulate social groups of virtual pedestrians that display social formations and waiting behavior. SGN is the first social-group method that adds sociality to real-time simulations on the global-planning level, the route-following level, and the local-movement level of the underlying hierarchy as proposed in Section 1.2. In Chapter 11, we discuss how to combine all our contributions in a framework based on the *Explicit Corridor Map* [31] and discuss implementation details. Lastly, we draw overall conclusions and discuss future work in Chapters 12 and 13, respectively.

Chapter 2

Motivation and preliminaries



This chapter provides motivation and preliminaries for the contributions we make in the three parts of this thesis. We motivate why classical representations of traversable space in a virtual environment are not well-suited for advanced methods to generate complex behavior such as region-based path planning, region-based path following, coordinating dense crowds, and adding social behavior to small agent groups.

To this end, we start with discussing different ways to represent traversable space in a virtual environment; see Section 2.1. We discuss classical representations such as grids or waypoint graphs, and more recent surface-based representations that allow the extraction of relevant information such as closest-obstacle information or local crowd density. We show why such surface-based representations are well-suited for solving advanced path planning and crowd simulation problems and thus form the basis for the contributions we make in this thesis. In Section 2.2, we give an overview of the popular and widely-used A^* method. This method is well-suited for classical path planning problems on a grid or graph structure. The A^* method is also used within the context of the novel *Vertex-based Pruning* method, which we present in Chapter 4. In Section 2.3, we discuss the *Weighted Region Problem* (WRP) and its variants. Our novel region-based methods in Parts I and II can be seen as solutions to particular variants of the WRP. In Section 2.4, we discuss existing crowd simulation models. We show why these models are not well-suited for solving the problems we tackle in Part III of this thesis.

This chapter is based on the following publication:

[53] N. Jaklin and R. Geraerts. Navigating through virtual worlds: From single characters to large crowds. In D. Russel and J. M. Laffey, editors, *Handbook of Research on Gaming Trends in P-12 Education*, chapter 25, pages 527–554. IGI Global, 2015.

2.1 | REPRESENTING TRAVERSABLE SPACE IN VIRTUAL WORLDS

How should traversable space in a virtual world be represented? The choice for a certain representation is strongly connected to the complexity and required efficiency of the application at hand. In this section, we give an overview of several representations and their pros and cons. We start with discussing how traversable space is represented in one of the most popular application areas of virtual worlds: entertainment games.

As an example for which representing traversable space is a trivial task, consider the classic $2D$ arcade game *Space Invaders*[1]. Here, the enemies move in a predetermined and scripted way, which does not involve any path planning. The player's traversable space is only a $1D$ straight-line segment at the bottom of the screen for which no complex representation is necessary. Note that there is no automated path planning for the directly-controlled player character either, but the player's traversable space is still important for collision avoidance with the left and right screen limits. A game that is more complex with respect to path planning is *Pac-Man*[2]. The enemies follow different strategies to catch the player. Here, the $2D$ game world consists of rectilinear tiles that are either traversable or blocked. Thus, a simple tile-based grid approach is a sufficient way to represent all traversable space. By contrast, many modern games tend to simulate an open world (e.g. *Grand Theft Auto V*[3]), or they provide a sandbox environment, which allows players to generate their own content (e.g. *Minecraft*[4]. Both types of games feature highly dynamic multi-layered 3D environments with different terrain types. Autonomous characters in future iterations of such game genres should not only plan collision-free paths, but also detect and use areas where climbing or jumping over gaps is possible to access

---

[1] Space Invaders; Taito Corporation; 1978.

[2] Pac-Man; Namco; 1980.

[3] Grand Theft Auto V; Rockstar Games; 2013.

[4] Minecraft; Mojang; 2009.

difficult-to-reach areas in the game world. Such features require a representation of traversable space that is far from trivial and still open to future research.

The topology of the virtual world also influences the choice for a particular representation of traversable space. While the topology of the world in *Space Invaders* is a $2D$ plane, there are games with more complex topologies. In *Pac-Man*, characters are allowed to exit to the left and right screen edges to appear on the opposite side. This behavior follows the topology of a cylinder. An example of a game with the world topology of a torus is *Asteroids*[5]. The player's ship as well as asteroids and flying saucers can exit the screen to all four edges and appear on the opposite side. $3D$ Games that allow a character to fly, such as *Descent*[6], feature a $3D$ space topology. In *Super Mario Galaxy*[7], the player can fully circumnavigate small planets and, hence, the world topology equals a sphere. In *Super Paper Mario*[8], the player has to interactively switch between $2D$ and $3D$ perspectives to solve puzzles. In *Portal*[9], the player can create traversable connections between arbitrary points in particular areas of the game world. The game *Monument Valley*[10] features dynamic Escher-like worlds that are physically impossible. Other exotic topological spaces such as the *Möbius Strip*[11] or the *Klein Bottle*[12] require different path planning strategies, and they may enable novel gameplay elements.

Note that in this thesis, we only discuss virtual worlds that require two-dimensional traversable space representations. Even when we discuss (multi-layered) $3D$ environments, a $2D$ representation is sufficient as long as virtual characters need to traverse $2D$ surfaces only. For actual $3D$ path planning (e.g. for autonomous flying characters), the problems we discuss in this chapter require different approaches beyond the scope of this thesis.

For more information on world topologies and the concept of traversable space in digital games, we refer to the discussion *Theorizing Navigable Space in Video Games* [148]. Its main focus is on space that the *player* can traverse, viewed from an abstract level. Traversable space in games and simulations that only feature directly-controlled player characters usually require simpler representations. Such representations only need to account for handling collisions with static obstacles and providing connectivity information between particular parts of the world, e.g. between different game levels. In this thesis, by contrast, we are interested in representing space for fully autonomous or user-controlled agents that are steered in an indirect way (e.g. by assigning a goal position via a mouse click or a finger press

---

[5] Asteroids; Atari Inc.; 1979.

[6] Descent; Parallax Software; 1994.

[7] Super Mario Galaxy; Nintendo EAD Tokyo; 2007.

[8] Super Paper Mario; Nintendo SPD; 2007.

[9] Portal; Valve Corporation; 2007.

[10] Monument Valley; Ustwo; 2014.

[11] Möbius Strip; F. Möbius and J. B. Listing (independently); 1885; `http://mathworld.wolfram.com/MoebiusStrip.html`; accessed January 13, 2016.

[12] Klein Bottle; F Klein; 1882; `http://mathworld.wolfram.com/KleinBottle.html`; accessed January 13, 2016.

on a touchpad). The autonomy of an agent thus requires artificial decision making that requires an agent to extract information from the virtual world that a directly-controlled player character naturally has due to a human being in control. We will now list several methods to represent traversable space that have been used in past, and we discuss their advantages and drawbacks.

2.1.1 | GRIDS

Grids can be intuitively described as regular subdivisions of the plane into cells of a particular shape and with a particular grid resolution. From an abstract point of view, grids can be seen as a special case of *lattice structures.* This means that by adding so-called *generator vectors* to a given cell point in the grid, any neighboring cell point can be obtained. For a $2D$ grid, two generators representing the up and right directions are sufficient to obtain all grid cell points [76].

Grids are intuitive and easy to implement, which makes them a popular representation of traversable space. The most common types are rectangular or square grids. However, other types have been widely used in simulations and games, too. Hexagonal grids are common, as well as grids with isometric diamond-shaped tiles; see Figure 2.1. From a topological point of view, isometric grids and rectilinear grids are equal. They are commonly used in $2D$ games to simulate an isometric view on pseudo-three-dimensional game worlds in which correct clipping is easily achieved by rendering objects on the grid from top to bottom along the screen. Rectangular grids are also used in many traditional and modern board games, and have also been widely used in pen and paper role-playing games to simulate combat scenarios.

When traversable space is represented using a grid, the actual path finding is usually performed on the dual graph of the grid. This is because a wide range of graph-search algorithms exists that can compute shortest paths efficiently (see Section 2.2). Furthermore, game objects are usually placed in the center of a grid cell. In the dual graph, each grid cell is represented as a vertex, and it is connected via an edge to each vertex that corresponds to an adjacent cell. While a rectangular, square or isometric grid keeps its structure when considering its dual graph (except for an offset translation), hexagonal grids become triangular and vice versa, see Figure

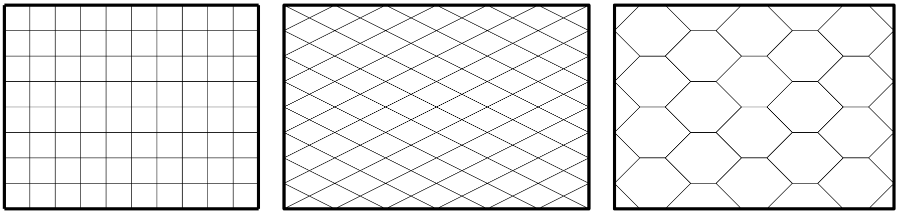


FIGURE 2.1: Rectilinear, isometric and hexagonal grids.

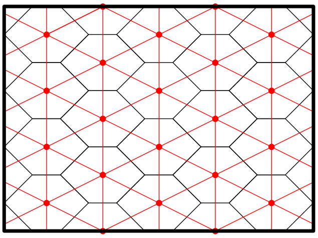


FIGURE 2.2: Hexagonal grids and triangular grids are dual graphs of each other. Each point is a vertex in the triangular graph (red), and it corresponds to a hexagonal cell in its dual graph representation.

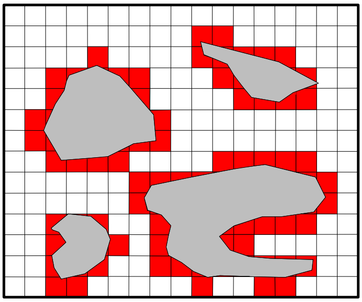


FIGURE 2.3: Squared grid overlaid on a polygonal environment: Red (dark) squares count as obstacles because they are partially covered by polygons. Some passages are not traversable due to the too coarse grid resolution.

2.2. Thus, using the center points of hexagonal cells as possible character positions is technically the same as performing path-finding on a triangular graph. In the case of square grids, typical variants are 4-neighbor and 8-neighbor square grids, depending on whether diagonal movement from one cell to another is allowed or not.

Path planning can also be done on the edges or vertices of the grid itself. *The Settlers II*[13], for instance, uses a hexagonal grid. The player can build roads along the edges and place flags along the vertices of the grid. Goods are then distributed and moved along the roads. The edges and vertices of the grid are also used in adaptations of abstract board games such as *Go* or in many puzzle games.

While grids are easy to implement, a major problem is that grids may not cover all of the traversable space that is visually displayed to the user. Some corners of the virtual world and important passages between obstacles might not be traversable due to a too coarse grid resolution; see Figure 2.3.

---

[13] The Settlers II; Blue Byte Software (today: Ubisoft Blue Byte); 1996.

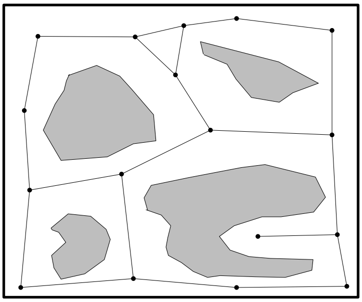


Figure 2.4: A waypoint graph for an environment with four obstacles.

The question of how much a grid path deviates from a shortest path on the exact geometry has been answered via path-length analysis proofs. Nash [98] presented a unified proof structure for upper bounds on the length of square, triangular, hexagonal and cubic $3D$ grid paths. One of the contributions of this thesis is a corresponding result for 8-neighbor grid paths in environments with weighted regions, when all regions are aligned with the grid; see Chapter 3.

Grids or their dual graphs are also inflexible because character motions are hard to coordinate when two or more characters follow the same edge bi-directionally. In addition, the edges of a graph are only a discrete subset of all possible trajectories in a continuous space. Resulting motions may not be visually convincing because the underlying graph edges are not smooth and do not cover energy-optimal paths. By energy-optimal paths, we refer to paths that minimize the expended energy of an agent while traversing them. How the term energy is defined depends on the application context and which real-world factors are being simulated. For instance, the overall path length and roughness of the underlying terrain could be modeled to determine the required energy an agent has to spend. Smoothing paths in a graph can be expensive and requires a global approach. This is undesired because entertainment games and simulations usually require real-time responses. Furthermore, the dual graph of a grid requires computationally expensive updates when obstacles are inserted, deleted, or moved [140]. Note that these issues also occur with other graph-based approaches such as waypoint graphs, which we discuss in the following section.

2.1.2 | Waypoint graphs and road maps

Waypoint graphs and road maps are two interchangeable terms. The nodes (or waypoints) resemble locations in the virtual world where an agent can be located. An edge between two nodes in the graph corresponds to a straight-line path which agents can traverse without hitting obstacles; see Figure 2.4.

Waypoint graphs are a general concept that is used in various research fields and in a wide range of applications. A common way to build a waypoint graph is to manually determine waypoints and edges for a given environment in the design phase of a game level or simulation environment. Some games such as real-time strategy games (e.g. Starcraft II[14]) even allow players to set waypoints themselves and use this as a tactical gameplay element.

Probabilistic Road Maps (PRMs) [66] and *Rapidly-Exploring Random Trees* (RRTs) [75] are commonly used in the robotics community. Besides generating waypoint graphs from manually determined points, they can also be defined based on a decomposition of the traversable space, e.g. induced by the medial axis of the environment (see Section 2.1.3 and [8]). Another example of a waypoint graph is the *visibility graph* of an environment and its generalized variant [74]. We now discuss the three mentioned examples in more detail.

PRMs [66] can be used to solve classical path planning problems, but also higher-dimensional motion planning problems. An example of such a problem is planning the motion of a robot arm with several joints through a $3D$ environment with obstacles. The basic idea is to randomly sample the given configuration space and build a roadmap graph of nodes that resemble collision-free configurations. A local planner is used to connect feasible nodes in the roadmap graph. After the start and goal configurations have been inserted and the roadmap graph is constructed, a graph-search method is used to find a collision-free path between the given configurations. PRMs are proven to be probabilistically complete. This means that the probability of finding a path (if one exists) converges to 1 when more and more valid samples are added to the roadmap graph over time. This basic idea leaves room for filling in particular details such as how to sample the configuration space or what local planner to use. Thus, many variants of the PRM approach have been presented, some of which can also handle non-holonomic constraints [122]. This means that they can be used to compute paths for autonomous agents that are restricted to particular movements in a given state, and are thus not able to move in any direction at any time (e.g. a virtual car). For further details, we refer the reader to the comparative study of different PRM approaches by Geraerts and Overmars [32].

RRTs [75] are similar to PRMs because they also randomly sample the (potentially high-dimensional) configuration space to build a graph of configurations, on which a graph search can be performed. Contrary to PRMs, RRTs are tree-structures with the root being the given start configuration, which makes RRTs a single-query approach, whereas PRMs can be used to answer multiple queries. RRTs can handle non-holonomic constraints, and they are designed to expand new nodes towards the largest Voronoi regions of the yet unexplored configuration space. We refer the interested reader to the survey by LaValle and Kuffner [77] and the book by LaValle on planning algorithms [76].

---

[14] Starcraft II; Blizzard Entertainment; 2010.

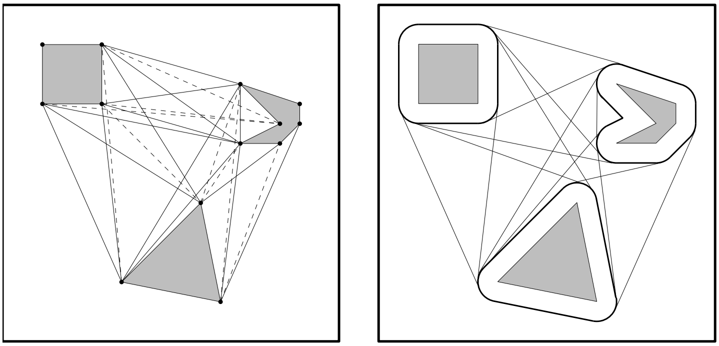


Figure 2.5: *Left*: A visibility graph for a scene with three obstacles. Dashed edges are pruned in the generalized visibility graph because their corresponding vertices in the generalized graph are not mutually visible in the generalized definition of *visibility*. *Right*: A generalized visibility graph for the same scene with inflated obstacles and corresponding visibility edges.

In a *visibility graph*, each node corresponds to a vertex of a polygonal obstacle in the environment. An edge between two nodes is added to the visibility graph, if the straight-line segment between them does not intersect any obstacles, or, in other words, if the two nodes are mutually visible; see Figure 2.5 (left). Note that the complexity of the visibility graph can be quadratic in the number n of polygon vertices. The graph can be computed in $O(n^2)$ worst-case optimal time [36, 146]. In addition, there are output-sensitive algorithms that can construct a visibility graph in $O(n \log n + E)$ time, where E is the total number of edges in the resulting visibility graph [8, 37]. Note that $E = \binom{n}{2} = O(n^2)$ for the worst-case scenario of a complete graph.

Laumond [74] introduced the concept of the *generalized visibility graph*, which has been used for path planning by several authors [85, 145, 152]. For an agent that is represented as a disc with a radius r, the obstacles in the generalized visibility graph are first inflated by r using Minkowski's operations. There are two types of nodes in the resulting graph. Vertices of the first type are concave corners of the inflated obstacles. Vertices of the second type are *fictitious* vertices that correspond to convex arcs of the inflated obstacles. In general, a shortest path in a polygonal scene consists of segments that are tangent to the borders of the obstacles. Therefore, it can be shown that all vertices of type one need not be connected by additional edges because they are never part of a shortest path. For the set of fictitious vertices, a generalized notion of visibility is used: Two fictitious vertices *see* each other, if and only if there is at least one pair of points on the corresponding two arc segments such that their straight-line connection is tangent to both arc segments. See Figure 2.5 (right) for an example. The generalized visibility graph is not only smaller in the number of edges, but it also makes an agent automatically keep clearance from obstacles. Since the search for tangents can be performed in $O(n^2)$ time [74], the

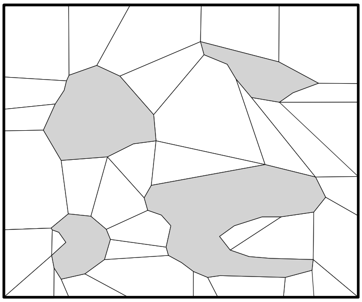


Figure 2.6: *Left*: A navigation mesh for the environment shown in Figure 2.3 and Figure 2.4.

overall time complexity to compute the generalized visibility graph is the same as for the visibility graph.

As mentioned before, waypoint graphs are generally smaller in the number of nodes and edges compared to the dual graph of a grid. This decreases the time to perform a path-finding algorithm on that structure (see Section 2.2). However, all graph-based techniques suffer from a range of problems that are inherent to these techniques. A graph does not provide information about the actual geometry of the scene, and it does not allow agents to deviate from their paths induced by the graph edges. This becomes an even bigger problem for a large crowd and collision-avoidance among the crowd members. The resulting paths may neither be natural nor visually convincing.

Given all these issues, it becomes apparent that graph-based representations are not sufficient for advanced path planning and crowd simulation tasks. We now discuss navigation meshes as a more recent representation to overcome the aforementioned problems.

2.1.3 | Navigation meshes

Intuitively, a *navigation mesh* is a set of two-dimensional simple polygons with connectivity information, which represents the traversable space in a virtual world. The term has been coined mainly in the gaming community in the context of game engines. There is no exact and formal definition, even though the term has been used in scientific papers on path planning. With the above informal description, the aforementioned grids can also be seen as special types of navigation meshes. However, navigation meshes are usually associated with polygonal representations that cover all traversable space exactly and that allow the extraction of additional information about the polygonal regions such as terrain information or clearance from obstacles for any given point. Figure 2.6 shows an example of a navigation mesh.

Entertainment games that come with level editors and support path planning with navigation meshes either use an automatic navigation mesh generation algorithm, or they let the user define traversable space for the level geometry within the editor. For example, in Counter-Strike: Source[15], automatic navigation mesh generation for user-generated maps is done by a flood-filling algorithm. The user defines spawn-points for players on traversable parts of the game map. From those points, the algorithm computes all traversable space that can be reached from the initial positions. Additionally, the user can explicitly mark traversable areas in case the algorithm fails to detect some parts due to steep stairs or ramps.

The project *Recast navigation*[16] is an open source library to automatically construct navigation meshes out of $3D$ level geometry. Recast generates a voxel mold from the $3D$ level geometry and automatically detects and prunes non-traversable areas. The resulting walkable space is then divided into simple overlaid $2D$ regions with one single non-overlapping contour. By tracing the boundaries of the regions, the algorithm produces a set of traversable convex polygons as a final output. The resulting navigation mesh can then be used for path planning. For example, it can be fed into *Detour*, which is a path-finding toolkit that accompanies Recast.

Kallmann [59] introduced a navigation mesh called a *Local Clearance Triangulation* (LCT). It can be used to answer path planning queries for agents of different size. Locally shortest paths can be computed in optimal time. If global optimality is required, an extended search is used to gradually improve the path. Furthermore, Kallmann discussed several algorithms and operations that are based on generic triangulation-based navigation meshes and on the LCT in particular [58]. Among these are methods for automatic agent placement, tracking moving obstacles, and ray-obstacle intersection queries. Kallmann's navigation mesh yields an exact representation of the environment and can handle dynamic updates efficiently. However, (multi-layered) $3D$ environments are not discussed.

Pettré et al. [108] introduced a navigation mesh based on discs and cylindrical areas. It supports multi-layered $3D$ environments and agents of variable size. However, it does not support an exact representation of the navigable space and efficient dynamic updates of the environment.

A navigation mesh that combines the advantages of the aforementioned approaches is the *Explicit Corridor Map* (ECM) [31]. It is an annotated data structure based on the medial axis of the environment, which is the set of all points that are equidistant from at least two distinct closest obstacle points; see Figure 2.7.

The medial axis can be seen as a special type of waypoint graph in which all edges run through the center of the free space between pairs of obstacle polygons. For each vertex of the medial axis graph, there are either at least three obstacle polygons

---

[15] Counter Stike: Source; Valve Corporation; 2004.

[16] Recast navigation; M. Mononen; `https://github.com/recastnavigation/recastnavigation`; accessed January 13, 2016.

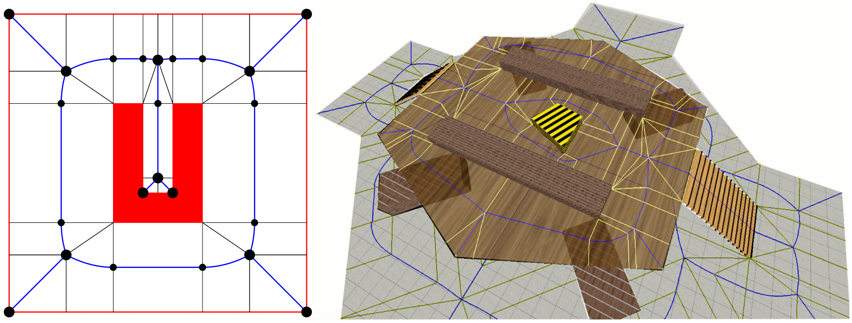


FIGURE 2.7: *Left*: Obstacles (shown in red; the center U-shape and the four bounding line segments), medial axis (blue), ECM vertices (large discs), event points (small discs), and connections of ECM vertices and event points with closest obstacle points (black line segments), subdividing the free space into ECM cells. *Right*: A multi-layered $3D$ environment and its medial axis [140].

that have the same distance from that vertex, or the vertex is placed in a non-convex corner of an obstacle. An edge between two vertices of the medial axis consists of a sequences of straight-line segments and parabolic arcs, depending on the type of corresponding obstacles to its left and right (with respect to a given orientation of the edges); see Figure 2.7 (left). With each element in this sequence, its left and right closest obstacle points are stored. This partitions a $2D$ environment into a set of walkable areas in $O(n \log n)$ time and $O(n)$ space, where n is the total number of obstacle vertices. Each area corresponds to one particular obstacle polygon, as all points in that area are closer to that obstacle than to all other obstacles.

In the field of Computational Geometry, a similar structure is known as the *Generalized Voronoi Diagram* (GVD), and it has been widely studied during the last decades. Approximations of a GVD can be efficiently computed using graphics hardware [48]. First, for each two-dimensional site (i.e. the obstacle polygons, lines or points) a three-dimensional distance mesh is computed and drawn by the graphics hardware, each mesh in a different color. By projecting the distance meshes back onto the $2D$ plane and tracing the boundary lines of the different regions in the color buffer, a feasible approximation of the GVD can be obtained. This approach requires the obstacle polygons to be convex, so concave polygons are first subdivided into convex ones. The GVD depends on how the given Voronoi sites are defined. For the exact same geometrical scene, different variants of the GVD can be obtained depending on whether obstacles polygons are treated as separate lines or as whole (convex) polygonal sites. The medial axis, by contrast, is independent of this choice because it is defined for any two distinct obstacle points, no matter what the structure of the obstacles is. Technically, the medial axis can be seen as a variant of the GVD, in which all edges are pruned that the GVD might contain due to treating obstacles as sets of line segments.

There are software libraries available for computing GVDs: *Vroni* [47] and *Boost*[17]. As a pre-processing step, undesired intersections and overlaps caused by imprecision in the geometry data can be detected and removed using corresponding functions of the Boost library. Vroni or Boost can then be used to robustly compute an ECM.

The ECM has many advantages. All traversable space is represented with respect to the correct topology of the environment. This resolves the issues that are inherent to all approximated representations that we discussed before. Furthermore, the ECM is space-efficient and supports time-efficient extraction of global paths with any desired amount of clearance from obstacles. The ECM works both for $2D$ and multi-layered $3D$ environments; see Figure 2.7. In addition, the ECM can be dynamically updated in real-time by only applying local changes to the mesh whenever an obstacle is inserted or deleted [140].

This concludes the discussion on representing traversable space. We have discussed grids and waypoint graphs, and we have shown why such representations are not well-suited for advanced path planning tasks. Examples of such tasks are the computation of smooth and energy-optimal trajectories, or the simulation of higher-level behaviors for autonomous agents that require the extraction of local information about the virtual environment. What is needed instead is a surface-based representation such as a navigation mesh, which allows to extract additional information about the scene, the geometry, nearby agents, or terrain types. The novel algorithms presented in the three main parts of this thesis are based on surface-based representations, either on an annotated triangulation of the scene or on the Explicit Corridor Map (ECM) [31]. These methods, however, are designed in a flexible way independent of a particular representation. They can be used on similar data structures, too, as long as the needed information can be extracted efficiently. For more algorithms on automatic navigation mesh generation and related path planning topics, we refer the interested reader to the *AI Game Programming Wisdom* book series [112].

In conclusion, we believe that surface-based representations of the traversable space are a promising next step into the future of simulating immersive virtual worlds. In the following section, we discuss the popular A^* graph-search algorithm, which will be part of our novel *Vertex-based Pruning* method for computing ϵ-optimal paths in weighted regions, which we present in Chapter 4.

2.2 | A^* AND ITS VARIANTS

The A^* algorithm [42] is one of the best-known and probably the most-used pathfinding algorithm. This is because it can efficiently compute shortest paths in a

---

[17] The Boost C++ Library; `http://www.boost.org/` ; accessed January 13, 2016.

graph and combines the advantages of *Dijkstra's algorithm* [21] with a greedy *Best-First-Search* strategy. With only small and easy-to-implement variations, a wide range of variants (e.g. shorter computation time to find good paths rather than optimal ones) can be obtained. This makes the A^* algorithm a flexible and powerful tool for many path-finding applications [131].

The main idea of A^* is to combine actual traversal costs from a start node with heuristic values that estimate the distance to a target node. Given an edge-weighted graph with two designated nodes s and t, a function g assigns to each node n in the graph the costs of the currently known shortest path from s to n. Furthermore, a heuristic function h assigns to each node in the graph the estimated costs of a path from n to t. A^* then starts to search for a path from s to t by computing $f = g + h$ for each node under consideration and expanding a node with minimum f-value in each step. The g-value of a node n is updated whenever a shorter path than the current one from s to n is detected. A* manages two lists of nodes, the *open list* and the *closed list*. While the open list stores all nodes that are currently under consideration, the closed list stores all nodes that have already been expanded and do not need to be visited again.

It can be proven, if the heuristic function h does not overestimate the actual costs for each node, that A^* will always find an optimal (i.e. shortest) path. If h overestimates the costs for some or all nodes, the computation time of the search may be decreased, but the path may not be optimal.

If h is an exact estimation of the actual costs, then A^* has the nice property to find a shortest path in optimal time. In this case only the nodes that are contained in a shortest path as well as their neighbors are expanded; see Figure 2.8 (bottom). Even if all paths in the graph are shortest, A^* will expand only one of them depending on the sorting strategy of the open list. A theoretical approach to exploit this property could be to compute all costs of optimal paths for all pairs of nodes in the graph as a preprocessing step (without storing the paths themselves to save space). These costs could then be used as an exact heuristic to make A^* compute optimal paths in optimal time. However, this is only practicable for small graphs and is usually not a feasible solution in gaming or simulation applications.

If we drop the heuristic function h and let $f = g$, we get *Dijkstra's algorithm* [21]. If we only consider h and ignore all g-values in each node (i.e. $f = h$), we get a greedy *Best-First-Search* strategy. Therefore, A^* can be seen as generalization of both strategies. Figure 2.8 illustrates the three different strategies on a grid with uniform costs. It shows that Dijkstra's algorithm finds an optimal path, but expands a large number of nodes. The greedy strategy expands only a few nodes, but the resulting path is far from optimal. A^* combines both strategies, finding an optimal path while expanding only a few nodes. Note that in the example, we assume to have an exact heuristic h, which reduces the number of expanded nodes to a minimum.

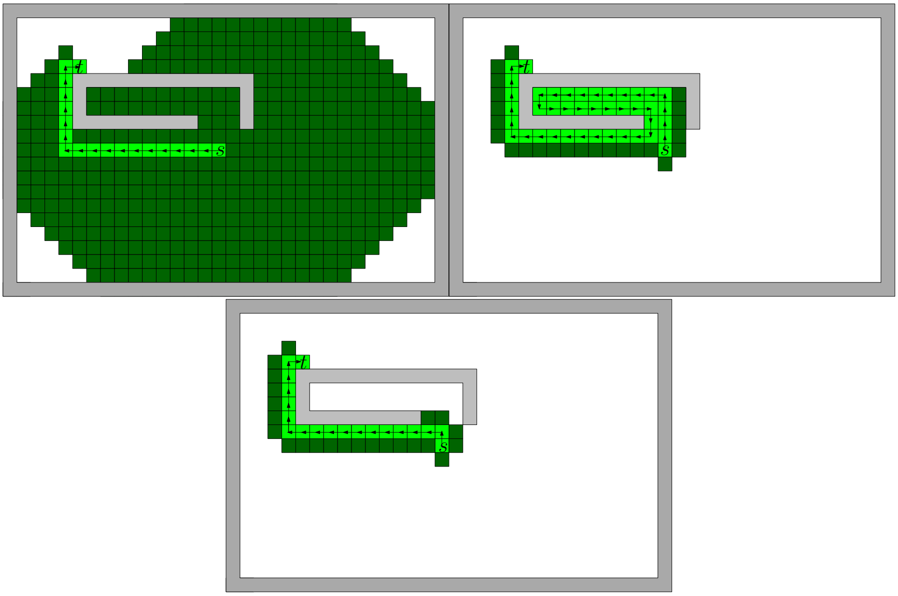


FIGURE 2.8: Comparison of different search strategies on a grid with uniform costs. Obstacles are shown in grey, free space is shown in white. Expanded cells are shown in dark green. Cells that are part of the final path are shown in bright green (with arrows pointing towards the next node on the final path). *Top-Left*: Dijkstra's algorithm expands many nodes, but finds an optimal path. *Top-Right*: Best-First-Search expands only a few nodes, but finds a non-optimal path. *Bottom*: A^* combines both advantages (an exact heuristic is chosen to illustrate the strength of A^*).

Finding a feasible heuristic can be difficult because the quality of the heuristic depends on the environment. The Euclidean distance is a popular choice. However, it is not well-suited for maze-like environments in which the length of a path between two points may differ significantly from the Euclidean distance between them.

There are many more variants and modifications of A^*. As the size of the open list strongly influences the computation time for the final path, many variants aim at keeping the number of nodes in the open list small. For example, the size of the open list can be limited by a constant number of nodes, and nodes with the highest f-values can be dropped whenever the open list becomes bigger than that number (*beam search* [83]).

Instead of generating one large set of opened and closed nodes, performance can also be improved by searching from s to t and from t to s simultaneously and stop when the two paths meet in the same node. However, the result is not guaranteed to be optimal.

Another concept is to inflate the h-values by some given weight $w \geq 1$ and to compute the f-values as $f = g + wh$. With this *weighted* A^* variant [110], additional emphasis is put on the heuristic. In this way, the expansion of nodes that appear to be closer to the goal are preferred, thus yielding a trade-off between computation time and quality of the path. This idea is further extended to *anytime* variants of A^*, which start with a high weight for the heuristic values to compute a first rough solution quickly. This first solution is then improved over time by adjusting the weights. One of those anytime variants is ARA^* [81], in which the weight w is based on a linear trajectory and two additional user-controlled parameters. The user has to set the initial value of w together with a fixed step size Δw by which the weight is decreased between the computation of solutions. Another anytime variant of A^* is called *Anytime Non-parametric* A^* (ANA^*) [134]. It uses a novel criterion for deciding which node to expand next in each step. Instead of expanding the node with lowest weighted f-value, it expands the node that maximizes the term $e = (G - g)/h$, with G being the costs of the currently best solution (which is set to infinity in the first iteration). The term e can be intuitively understood as the ratio between the "budget" that is left to improve the current-best solution and the estimated costs between the node and the goal. Maximizing e corresponds to picking the largest weight w such that $f \leq G$, and to expanding the node that is most promising to improve the current-best solution. ANA^* overcomes the user's problem in ARA^* of finding appropriate parameters, and has comparable and in some cases better performance than ARA^*. Another range of related methods are A^* variants in which the costs can dynamically change over time, such as D^* Lite [71] or GAA^* [126].

Despite the fact that A^* and its variants are widely used for general graph-search problems and path planning in simulations and games in particular, it is not the final answer. Many problems – such as making agents use all the free space in the environment for collision avoidance or dense-crowd coordination – cannot be solved by using a search algorithm such as A^* on a graph alone. While such a search may still be part of finding an overall solution, recent approaches – including the novel methods we present in this thesis – use surface-based representations such as the aforementioned navigation meshes and the possibility to store additional information about the virtual environment.

This concludes the discussion on A^*. In the following section, we discuss the Weighted Region Problem (WRP). The novel methods presented in Parts I and II of this thesis can be seen as solutions to particular variants of the WRP.

2.3 | THE WEIGHTED REGION PROBLEM

2.3.1 | DEFINITION AND FIRST APPROXIMATION ALGORITHM (MITCHELL AND PAPADIMITRIOU, 1991)

The *Weighted Region Problem* (WRP) has been introduced by Mitchell and Papadimitriou [87] as a generalization of the classical path planning problem. Instead of a polygonal scene with static obstacles and traversable space, the input for the WRP is a polygonal planar subdivision with positive weights for each polygonal region. Each polygonal region is traversable, but yields different traversal costs depending on its weight. The goal is to compute a path between two query points which is optimal with respect to the traversal costs. Optimal paths in weighted regions consist of straight-line segments with their bending points lying on the borders of the region polygons. Thus, each straight-line segment runs through only one type of region. The costs for a path are defined as the sum of the weighted Euclidean lengths of each straight-line segment.

Mitchell and Papadimitriou presented the first ϵ-approximation algorithm for solving the WRP. An ϵ-approximation algorithm is an algorithm that does not compute an exact solution to a given problem instance, but that rather computes an approximate solution that is proven to be at most $1 + \epsilon$ times as expensive as an optimal solution with respect to a given cost function. Such algorithms are also referred to as *approximation schemes* when $\epsilon > 0$ can be chosen arbitrarily small by the user, thus allowing the approximation of an exact solution as closely as needed, usually at the cost of additional running time. The algorithm by Mitchell and Papadimitriou has a running time of $O(n^8 \log \frac{nNW}{w\epsilon})$, where n is the number of vertices, N is the maximum integer coordinate of any vertex of the triangulation, and w and W are the lowest and highest weight of the regions, respectively. Due to the high computational complexity, the algorithm is mainly of theoretical interest. The authors discuss several fundamental properties of the WRP. For instance, at its bending points, an optimal path in weighted regions obeys *Snell's law of refraction.* In other words, an optimal path crosses the borders of two adjacent regions in the same way as a ray of light crosses the border of two different materials.

The WRP has applications in many fields such as robotics [14], simulations [113], or entertainment games [61, 135]. Moving entities such as virtual humans or robots need to steer through a scene with various region types. However, the WRP is proven to be unsolvable in the *Algebraic Computation Model over the Rational Numbers* (ACM$\mathbb{Q}$) [11]. In other words, a solution to an instance of the WRP cannot be expressed as a closed formula in ACM$\mathbb{Q}$. We will now give a brief outline of the proof.

2.3.2 | UNSOLVABILITY IN ACMQ (DE CARUFEL ET AL. 2012)

The unsolvability of the WRP in ACM$\mathbb{Q}$ [11] is a fundamental theoretical result. It justifies further research on approximation algorithms rather than trying to solve the problem in an exact way. The proof uses in-depth algebraic arguments from Galois theory [23]. The authors show that the unsolvability can already be shown on a small toy instance of the WRP with only three regions. The proof technique they used is based on a general technique by Bajaj [5].

The main idea of the proof is as follows: Let p be an irreducible polynomial of degree $d \geq 5$ over $\mathbb{Q}$. From Galois theory, it follows that $p(x) = 0$ is solvable by radicals if and only if the *Galois group* Gal(p) is solvable. The authors now define a particular irreducible polynomial p and an instance of the WRP such that solving this instance exactly in ACM$\mathbb{Q}$ is equivalent to solving p by radicals. Furthermore, they show that the Galois group Gal(p) is isomorphic to S_{12}, which is the symmetric group over 12 elements. It is known [23] that the symmetric group S_n is solvable if and only if $n \leq 4$. Thus, S_{12} is not solvable, which implies that p is not solvable by radicals, which again implies that the particular instance of the WRP is not solvable in ACM$\mathbb{Q}$.

The unsolvability result justifies the search for approximation algorithms to compute paths that are near-optimal within a certain costs-bound. A simple approach is to approximate the exact geometry of the scene with a rectilinear grid in which each grid cell is assigned the largest weight of all regions that intersect that cell. An efficient graph-search algorithm such as the A^*-algorithm (see Section 2.2) can then be used on this grid. This approach is fast and easy to implement, but as discussed in Section 2.1.1, a grid does not capture the exact geometry of the scene. A different approach is using approximation algorithms that work on a triangulation of the polygonal scene. We will now give an overview of such methods.

2.3.3 | EXISTING APPROXIMATION ALGORITHMS

After the first ϵ-approximation had been described by Mitchell and Papadimitriou [87], several approximation algorithms for the WRP were presented in the following years. Mata and Mitchell [86] created a graph called *Pathnet* to approximate optimal solutions to the WRP. The method makes use of cones around all vertices, which limit the paths that can extend from a vertex. The Pathnet can be constructed in $O(kn^3)$ time, where k is the number of such cones. Once the Pathnet is constructed, it can find ϵ-approximate paths in $O(n \log n)$ time. Note that ϵ cannot be chosen arbitrarily small for this method, but its value results from the given problem instance with $\epsilon = O(\frac{W/w}{k\theta_{min}})$. W/w is the ratio of the maximum and minimum weight, respectively, and θ_{min} is the minimum internal face angle of the subdivision.

Aleksandrov et al. [2] presented an ϵ-approximation scheme to solve the WRP up to an arbitrary constant $\epsilon > 0$. The method uses Steiner points that are added to the edges of all triangles in the scene with a logarithmic distribution. Because this distribution leads to infinitesimally small distances between Steiner points near the vertices, the authors define a *vertex vicinity*, which is a circle around each vertex that is void of Steiner points. Within each triangle, all Steiner points and vertices are connected by additional edges. Path planning queries can then be answered using Dijkstra's algorithm [21] (improved by using Fibonacci heaps). The running time is $O(mn\log(mn)+nm^2)$, where m is the total number of Steiner point, which increases for smaller values of ϵ. The placement of Steiner points is done in such a way that the distance between any two subsequent Steiner points q_i and q_{i+1} on an edge e_q is proven to be ϵ times the distance from q_i to an opposite edge e_p. For further details on this method, we refer the reader to Section 4.1, where we use it as part of our novel VBP method. Aleksandrov et al. [3] also presented a variant of the Steiner point method, in which the Steiner points are placed on the bisectors of the triangles. This variant has a better running time, but the paper lacks the description of some critical cases required for a thorough practical implementation.

Sun and Reif [127] presented an ϵ-approximation algorithm called *BUSHWHACK*. Similar to Aleksandrov et al. [2], they use Steiner points on the edges of the triangulation. Instead of searching the graph with Dijkstra's algorithm [21], they introduced a new graph-search method that exploits the underlying geometrical properties of the scene. Using an *interval* data structure, the BUSHWHACK algorithm can find ϵ-optimal paths in $O(\frac{n}{\epsilon}(\log\frac{1}{\epsilon}+\log n)\log\frac{1}{\epsilon})$ time.

Ferguson and Stentz [24] presented *Field D^**, an interpolation-based planning and replanning method well-suited for scenarios with weighted regions. The input for Field D^* is a grid representation of the weighted scene, in which all regions are aligned with the grid. Field D^* computes paths that may follow any direction and are thus not restricted to angles of $\frac{\pi}{2}$ at its bending points.

Research has also been conducted on variants of the WRP. Aleksandrov et al. [1] introduced a data structure called *All Points Query* (APQ), which can be used to efficiently find ϵ-optimal paths for all-pairs queries on an instance of the WRP. APQ has a high construction time, and it is therefore mainly useful for answering many queries on the same scene. Cheng et al. [13] presented a method to compute *homotopic* paths that are ϵ-optimal in an instance of the WRP. Gheibi et al. [35] recently considered a variant of the WRP with weighted arrangements of lines instead of bounded triangulated subdivisions.

The described approximation algorithms are computationally expensive and not suited for real-time applications with many virtual agents. One of the main contributions of this thesis is a novel method called *Vertex-based Pruning* (VBP) to compute approximated paths in weighted regions in real-time; see Chapter 4. VBP combines an efficient A^* search on a grid with the original Steiner point method by Aleksandrov et al. [2]. Furthermore, the method can handle arbitrary scenes given as planar polygonal subdivisions with weighted regions, and it therefore does not require all regions to be aligned with a grid.

The WRP forms the basis for the novel methods we present in Parts I and II of this thesis, which deal with region-based path planning and region-based path following for individual agents. In Part III, by contrast, we present new models to coordinate multiple agents in dense crowds and social groups. To this end, we will now briefly discuss crowd simulation models.

2.4 | CROWD SIMULATION

The overall goal of crowd simulation models is to mimic particular behaviors that can be observed in real-life crowds and display these in virtual environments. In real crowds, the overall behavior is determined by complex factors from different research fields. Not only physical factors such as the *Principle of Least Effort* [153] to minimize the expended energy play a role here, but also cultural factors and findings from fields such as sociology and psychology. The latter especially holds for small social groups, in which factors such as social formations and social territories are involved. A crowd simulation model that considers all relevant factors, if that were possible, can be seen as the holy grail in this research field. Existing models are still far away from that ideal state, and they usually focus on a few particular aspects.

The level of realism that is needed strongly depends on the application at hand. To show two fighting armies in the background of a blockbuster movie, more effort needs to be spent on the animation and the lighting to match the overall action and atmosphere of the scene, rather than on realistic navigation and coordination of the individual crowd members. In a simulation to train safety personnel evacuate a building, by contrast, the animation and visual appearance of the crowd is less important than realistic navigation and coordination of the crowd.

Existing crowd simulation models can be subdivided into agent-based models and flow-based models. Agent-based models treat each character in a crowd as an individual autonomous entity. Among these are models to simulate small social groups [60, 65, 89, 91, 111]. Flow-based models, by contrast, treat the whole crowd as one big entity, and the simulation of its members follows particle-based approaches from fluid simulations or gas kinetics [69, 130].

Each of the two paradigms has its advantages and drawbacks. Agent-based models are a good choice to simulate crowds in low-density and medium-density scenarios in which the individual actions of crowd members determine the overall crowd behavior to a great extent. However, agent-based models are computationally expensive and struggle in high-density scenarios when a large number of agents needs to be simulated and coordinated. Flow-based models can be seen as the complement paradigm. They are designed to work well in high-density scenarios, and they require a large number of crowd members to display their advantages. They are not well-suited to simulate only a few agents in low-density and medium-density scenarios.

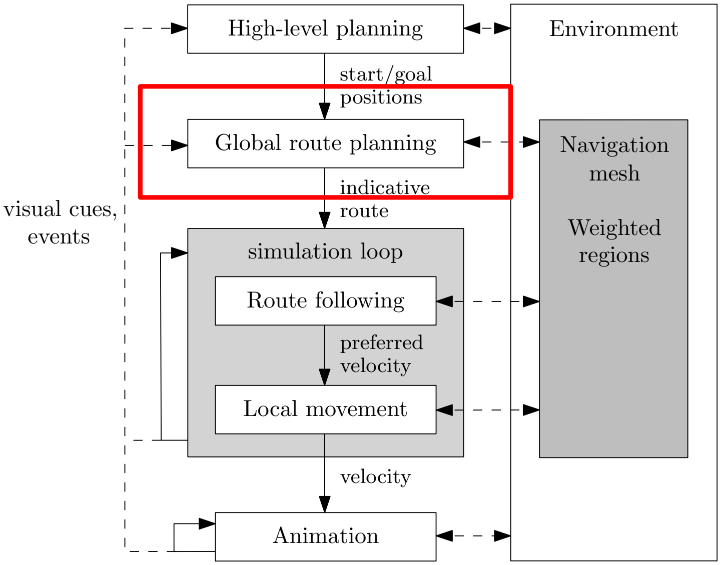
What follows from the above observations is that it is difficult to decide on a particular crowd simulation model when the application at hand features highly dynamic crowd scenarios in which the crowd density may change arbitrarily and frequently. To address this, a recent trend in crowd simulation research is to design hybrid models that try to combine the advantages of agent-based and flow-based models. The new model we present in Chapter 9 can be seen as such a hybrid. It is technically an agent-based model, but uses simple local rules to improve the coordination of dense crowds while still being real-time applicable for a large number of agents.

For an overview of existing crowd simulation models and social group methods related to the contributions in this thesis, we refer the reader to Sections 9.1 and 10.1, respectively. For a more general overview of crowd simulation topics, we refer the reader to the books by Pelechano et al. [107] and Thalmann and Musse [128]. The latter also includes topics such as virtual human animation, crowd rendering and populating environments.

Part I

Path Planning in Weighted Regions

GRID-PATHS IN WEIGHTED REGIONS



In this chapter, we analyze the *Weighted Region Problem* (WRP), which we have discussed in Section 2.3. In short, the WPR is defined as the problem of finding a cost-optimal path in a weighted planar polygonal subdivision. A fast and easy-to-implement solution to the WRP is searching for cost-optimal paths on a grid representation of the scene, in which each grid cell is assigned with a weight. This weight could, for instance, be the highest weight among all polygonal regions that intersect a grid cell. However, grid representations are only a rough approximation of the actual scene and do not capture its exact geometry. Hence, grid paths can be inaccurate or might not even exist at all. Methods exist that work on an exact representation of the scene, and such methods can approximate an optimal path up to an arbitrarily small ϵ-error [2, 86, 87, 127]. However, these methods are computationally inefficient and thus not well-suited for real-time applications.

The main contribution of this chapter is a path-cost analysis proof on the quality of 8-neighbor-grid paths in weighted regions. We prove that – in the general case – the costs of such a grid path can be arbitrarily high compared to the costs of an optimal path that is not restricted to lie on the grid. In the literature, such a non-restricted path is sometimes called an *any-angle* path [92, 98]. If all regions are aligned with the grid, we prove that the costs of an optimal grid path are at most $2\sqrt{2}$ times the costs of an optimal *any-angle* path. In the paper on which this chapter is based (see below), we have claimed a higher upper bound. That proof turned out to contain an error[1], which we have now fixed. The fixed proof now yields an improved upper bound of $2\sqrt{2}$.

We start with giving an overview of existing work that is related to our proof in Section 3.1. We then discuss preliminaries and definitions such as *region-homotopic paths*, which we are going to use within our proof in Section 3.2. Subsequently, we present the proof itself in Section 3.3.

This chapter is based on the following publication:

[55] N. Jaklin, M. Tibboel, and R. Geraerts. Computing high-quality paths in weighted regions. In *Proceedings of the 7th International ACM SIGGRAPH Conference on Motion in Games (MIG 2014)*, pages 77–86, 2014.

3.1 | RELATED WORK

Existing literature on the WRP lacks analytical path-cost analysis proofs for grid paths in weighted regions. For the classical path planning problem without weights, however, worst-case bounds for path costs on various grid types have been studied over the past few decades.

Luczak and Rosenfeld [84] introduced an integer-based coordinate system for 6-neighbor hexagonal grids. Based on that coordinate system, they presented a formula to compute the distance between two points on such a grid. Nagy [93–95] presented analytical discussions and algorithms for computing shortest paths on triangular and hexagonal grids. Björnsson et al. [9] proved analytically that graph-based search algorithms such as A* have smaller search complexities when run on 6-neighbor hexagonal grids than on 8-neighbor octile and 4-neighbor square grids. García and Garrido [30] provided a corresponding practical result. They proved empirically that searches on hexagonal grids can be faster than on square grids when the grids are large. Nash [98] presented in his PhD thesis a unified proof structure to derive upper bounds on the length of grid paths. Nash considers square, triangular, and hexagonal grids for non-weighted scenarios in $2D$ and cubic grids in $3D$. While some of the derived bounds had been proven before [95], the unified proof structure was a novel contribution that also led to new upper bounds for grids that

---

[1] Special thanks to Mark de Berg, who pointed out the flaw in the previous proof and helped with fixing it.

had not been analyzed up to that point. In particular, Nash showed that the additional costs of a path on a 3-neighbor triangular grid cannot be higher than 100% of the costs of an optimal path. For 4-neighbor square grids, the additional costs are bounded by 41% of the costs of an optimal path. Corresponding results are 15% for 6-neighbor hexagonal grids, 15% for 6-neighbor triangular grids, 8% for 8-neighbor square grids, 4% for 12-neighbor hexagonal grids, 73% for 6-neighbor $3D$ cubic grids and 13% for 26-neighbor cubic grids.

Other work that is related to paths and distances on particular grid structures comes from the digital imaging and pattern recognition community. In 1968, Rosenfeld and Pfaltz [116] introduced *digital distances* on *city-block* and *chessboard* neighborhood relations. These two terms correspond to 4-neighbor and 8-neighbor square grids, respectively. Such digital distances were later generalized by introducing weights for each direction of the underlying space, e.g. by Strand [125] and Nagy [96]. This means that a step from one grid cell to an adjacent cell does not have a uniformly distributed cost, but the cost depends on the direction of the step. Note that finding cost-optimal paths on a grid with such a weighted distance function is not the same as finding cost-optimal paths on a grid in the sense of the WRP, which we analyze in this chapter. In our context, the costs for a step from one grid cell to an adjacent cell depend on the underlying regions, and they can therefore change arbitrarily when traversing a path in one direction. For distance functions with directional weights, by contrast, the costs are the same for each step in a particular direction, but different directions can have different costs per step.

3.2 | REGION-HOMOTOPIC PATHS

In this section, we give definitions and general assumptions we will be using within our proof in Section 3.3. We start by introducing *region-homotopic* paths. Afterwards, we discuss grids and *grid-optimal* paths.

One important topological property in classical path planning without weighted regions are *homotopic* paths. Two paths with the same fixed start and goal positions are homotopic, if there exists a homotopic function that continuously maps one path into the other without having to cut open the path or intersect obstacle polygons in the scene. The question whether two paths are homotopic can be important when analyzing error bounds for approximations of optimal paths. Thus, it is useful to have a corresponding property for paths in the context of weighted polygonal scenes, which we can use in our analysis in Section 3.3. We generalize homotopy for paths in weighted regions in the following way:

Given the scene as a polygonal subdivision of the plane, let $\mathcal{R} = \{(r_1, w_1), (r_2, w_2),$ ..., $(r_n, w_n)\}$ be the set of its n non-overlapping region polygons r_i together with a weight $w_i > 0$. A region polygon in our context is a simple polygon with no further constraints. In particular, a region polygon does not contain any holes, and it does not need to be convex. For an edge that is shared by two adjacent region polygons, we count that edge as part of the region polygon that has a lower weight between the two. Thus, we define the higher-cost region as topologically open near that edge, whereas the lower-cost region is topologically closed near that edge. This allows us to let a path make use of region-boundary edges to circumnavigate a high-cost region. In case of a tie, that choice does not matter, and both options are equally valid.

Let π_1, π_2 be two paths connecting the same start and goal positions s and g in $\mathcal{R}$, respectively. In order to generalize the definition of homotopic paths, we map the given weighted scenario to an instance of the classical path planning problem by declaring all regions that either of the two paths intersect as free space. All remaining regions are declared as hard obstacles with an infinite weight. More formally, we let $\mathcal{R}(\pi_1, \pi_2) = \{(r, w) \in \mathcal{R} \mid \pi_1 \text{ or } \pi_2 \text{ intersects } r\}$. We then define $\mathcal{R}'(\pi_1, \pi_2) = \{(r_1, w'_1), (r_2, w'_2), ..., (r_n, w'_n)\}$ as the same set of region polygons r_i as in $\mathcal{R}$, but with the following weights:

$$w'_i = \begin{cases} 1 & \text{if } (r_i, w_i) \in \mathcal{R}(\pi_1, \pi_2) \\ \infty & \text{else .} \end{cases}$$

Definition 1. We say that π_1 and π_2 are *region-homotopic* in $\mathcal{R}$, if and only if they are homotopic in $\mathcal{R}'(\pi_1, \pi_2)$.

We consider a grid with square grid cells to approximate the scene. Each cell in the grid is weighted with the highest weight of all regions that intersect the cell. From a graph-search point of view, we use the center points of each grid cell to represent the cell as a vertex in the graph.

We focus on 8-*neighbor grids* that allow movement on the grid in up to 8 directions. Thus, any grid path consists of horizontal, vertical and diagonal straight-line segments. Furthermore, we only consider grid cell center points as input for path planning queries.

By $\mathcal{C}(\cdot)$, we denote the function to measure the costs of a path π. If π consists of k straight-line segments π_i, and each π_i runs through one region with weight w_i ($1 \leq i \leq k$), we let

$$\mathcal{C}(\pi) = \sum_{i=1}^{k} w_i ||\pi_i||,$$

with $|| \cdot ||$ being the Euclidean norm. Since an optimal (non-grid) path in weighted regions is proven to consist of straight-line segments [87], we can use this cost function for both grid paths and optimal paths. If a straight-line segment runs along the edge shared by two grid cells, the smaller weight of the two cells counts for this segment. This choice reflects paths that are tangent to a high-cost region without intersecting the interior of that region.

Let $\gamma \in \mathbb{R}^+$ be the side length of each grid cell. Let C_1 and C_2 be two adjacent grid cells with weights w_1 and w_2, respectively. Given the above cost function, we can conclude the following. If C_1 and C_2 are horizontally or vertically connected by a straight-line segment l, the costs for moving from one cell to the other are $\mathcal{C}(l) = \frac{1}{2}\gamma w_1 + \frac{1}{2}\gamma w_2 = \frac{1}{2}\gamma(w_1 + w_2)$. If C_1 and C_2 are diagonally connected, the costs are $\mathcal{C}(l) = \frac{1}{2}\gamma\sqrt{2}(w_1 + w_2)$.

In addition to the above, we use the following definition to distinguish between optimal paths on the grid and optimal paths on the exact geometry of the scene.

Definition 2. We call a grid path *grid-optimal*, if it is optimal among all other possible grid-paths with respect to the cost function $\mathcal{C}(\cdot)$.

3.3 | PATH-LENGTH ANALYSIS OF 8-NEIGHBOR GRID PATHS IN GRID-ALIGNED REGIONS

In this section, we analyze the quality of grid-optimal paths with respect to optimal paths in weighted regions. First, we show that a grid-optimal path can be arbitrarily more expensive than an optimal path between the same points, if a fixed cell size is given. Afterwards, we focus on scenarios in which all regions are aligned with a fixed grid. We show that even in this simplified scenario, a grid-optimal path and an optimal path are not necessarily region-homotopic. We can, however, prove an upper bound on the costs of grid-optimal paths compared to the costs of an optimal path when all regions are aligned with the grid. The latter is the main result in this section.

Let us assume a grid with a fixed resolution. If an optimal path and a grid path are not region-homotopic, the grid path can be arbitrarily more expensive than the optimal path. This already holds in the classical path planning situation in which no weighted regions are given, whenever the paths are not homotopic. Figure 3.1 shows an example in which a grid-optimal path Γ and an optimal path π^* that connect points s and g are not (region-)homotopic. This is due to using an overly coarse grid resolution. By increasing the height of the obstacle polygon P, we can make Γ become arbitrarily expensive. This also shows that a grid path does not necessarily need to exist at all in the general case.

The situation is different when all regions are aligned with the grid, i.e. when a grid cell contains exactly one region type. Remember that we count a cell edge that is shared by two adjacent cells as part of the cell that has a lower weight (in case of a tie it does not matter to which cell it belongs). A scene in which all regions are aligned

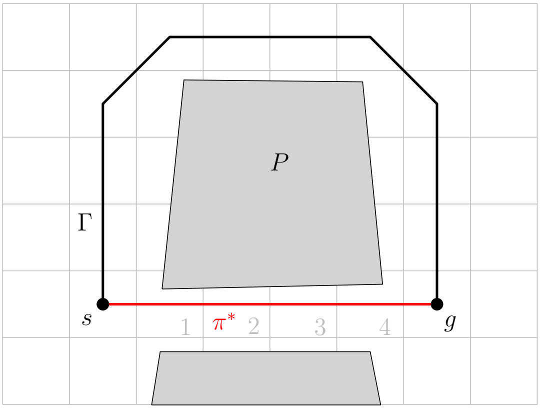


FIGURE 3.1: Example of two paths connecting points s and g. The grid-optimal path Γ can become arbitrarily expensive compared to the costs of an optimal path π^*, if the grid resolution is too coarse. Grid cells $1, 2, 3$ and 4 are not traversable for Γ because obstacle polygon P intersects them.

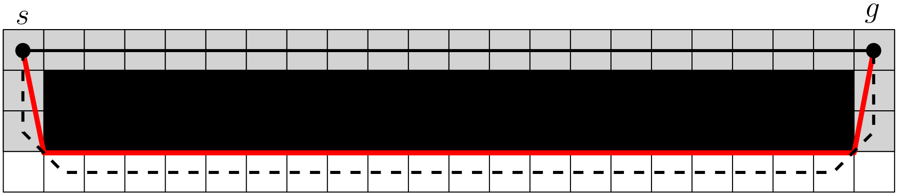


FIGURE 3.2: An example in which the grid-optimal path Γ (solid black) is not region-homotopic to the optimal path π^* (solid red). The grid path Γ' (dashed black) that is region-homotopic to π^* has slightly higher costs than Γ. The grid cell size is $\gamma = 1$. The white region has a low weight of 1, the gray region has a weight of 1.3, and the black region has a very high (infinite) weight. The path costs are: $\mathcal{C}(\pi^*) \approx 26.63, \mathcal{C}(\Gamma) = 27.3, \mathcal{C}(\Gamma') \approx 27.45$.

with the grid can occur in applications with simple rectangular shapes or when the exact geometric shape of a region is less important (e.g. in grid-based games, or when a region resembles an abstract feature such as *dangerous* or *attractive*), so that it is feasible to approximate the region polygons and thus reduce the complexity of the scene. We now discuss important properties and analyze the quality of grid-optimal paths in this special case of the Weighted Region Problem.

The example in Figure 3.1 might give rise to the assumption that a grid-optimal path and an optimal path are always region-homotopic if we ensure that all regions are aligned with the grid. However, this is not the case. Aligning all regions with the grid does ensure that there exists a grid path in the same homotopy class in which an optimal path is contained, but it does not need to be grid-optimal. Figure 3.2 shows an example in which the grid-optimal path Γ (solid black) and the optimal path π^* (solid red) are not region-homotopic, even when all regions are aligned with the grid. The grid path following the same regions as π^* is denoted as Γ' (dashed black), and its costs are approximately 27.45. The costs for Γ, in comparison, are approximately 27.3.

In the remainder of this section, we prove an upper bound on the costs of grid-optimal paths with respect to the costs of an optimal path when all regions are aligned with the grid.

Theorem 1. Let all regions be aligned with the grid. Let Γ be a grid-optimal path, and let π^* be an optimal path. It holds that $\mathcal{C}(\Gamma) \leq 2\sqrt{2}\ \mathcal{C}(\pi^*)$.

Proof. The main idea is to exploit the fact that an optimal path π^* consists of straight-line segments [87]. We construct a grid path for which we can show the claimed upper bound, as follows:

Since we define a cell edge that is shared by two adjacent cells as part of the cell that has a lower weight between the two cells, an optimal path π^* induces an ordered sequence of intersected grid cells $S = C_1, ..., C_k$. The cell C_1 contains s, and the cell C_k contains g. Note that contrary to the classical path planning problem without weights, an optimal path can visit the same grid cell more than once. Consequently, two grid cells C_i and C_j in S do not necessarily need to be distinct.

Given the sequence S, we construct a grid path Γ' that uses a subset $S' \subset S$ of the cells in S. We choose S' such that connecting the center points of any two successive cells in S' yields a grid path. Furthermore, we show that the costs of Γ' in each cell $C_i \in S'$ are at most $\sqrt{2}\, w_i$, whereas the costs of π^* in each cell $C_i \in S'$ are at least $\frac{1}{2}\, w_i$, from which we can derive the claimed upper bound for the overall path costs.

We define S' as all cells $C \in S$ for which the segments of π^* that intersect C are longer than $\frac{1}{2}$. Since S is an ordered sequence, S' is an ordered sequence, too. We now show that the center points of any two successive cells C'_i, C'_{i+1} in S' can be connected such that the resulting path Γ' is a valid grid path.

It holds that $C_1 \in S'$ because C_1 contains s as the cell-center point, and π^* leaves C_1 by crossing one of its cell edges. Consequently, the part of π^* that is inside C_1 has a length of at at least $\frac{1}{2}$. This shows that $S' \neq \emptyset$.

Given a cell $C_i \in S'$, we let p be the *last* point through which π^* leaves C_i and enters C_{i+1}. Without loss of generality, we can assume that p lies on the right edge of C_i and below the midpoint of that edge (all other situations are symmetric); see Figure 3.3. Let q be the point through which the segment of π^* starting in p leaves C_{i+1}, and let l be that segment. If $|l| \geq \frac{1}{2}$, then $C_{i+1} \in S'$, and the connection between the center points of C_i and C_{i+1} is a valid grid-path connection. We can then iterate the same arguments, with C_{i+1} taking the role of C_i. Let us assume that

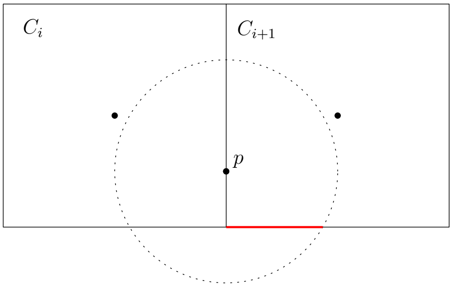


FIGURE 3.3: Possible positions (red) of point q, if the distance from p to q is smaller than $\frac{1}{2}$.

$|l| < \frac{1}{2}$. Figure 3.3 shows possible positions (red) of q. Note that other grid-edge positions within the dotted circle are not possible. Otherwise, this would mean that π^* leaves cell C_{i+1} and re-enters cell C_i, yielding a contradiction to our assumption that p is the *last* point through which π^* leaves C_i.

Now there are two cases, depending on which cell π^* enters after q. It can either enter the cell below C_i or the cell below C_{i+1}.

Case 1: The optimal path enters the cell below C_i. The only way to do so is via the grid vertex that is shared by all four grid cells. In this case, q equals that grid vertex. Let r be the next bending point of π^* after q, and let l be the segment between q and r. If $|l| \geq \frac{1}{2}$, then $C_{i+2} \in S'$, and the connection between the center points of C_i and C_{i+2} is a valid grid-path connection. If $|l| < \frac{1}{2}$, then the only possible position for r is below and co-linear with p and q; see Figure 3.4. Note that the bottom edge of C_i is not an option for r because then the path would re-enter cell C_i, yielding a contradiction to our assumption that p is the *last* point through which π^* leaves C_i. Assume r is indeed located below and co-linear with p and q; see Figure 3.5. This means that π^* leaves cell C_{i+2} and switches to C_{i+3} at r. Here we can use the same argument as before: If the segment of π^* between r and its next bending point is longer than $\frac{1}{2}$, then we know that $C_{i+3} \in S'$, and the diagonal connection between the cell centers of C_i and C_{i+3} is a valid grid connection. We can then iterate the same arguments, with C_{i+3} taking the role of C_i. If this is not the case, then the only possible positions of the next bending point would be at the top edge of C_{i+3} (shown in red). These positions, however, would contradict the fact that π^* is optimal because the segment from q to that next bending point would yield a cheaper path than first going down to r and then up again. This holds even when the weight for C_{i+2} is very low because the segment within C_{i+3} is always longer than the connection between q and the next bending point, which also counts as being inside C_{i+3} by construction.

Case 2: The optimal path enters the cell below C_{i+1}. Let r be the next bending point of π^* after q, and let l be the segment between q and r. First, r cannot lie to the left of the dashed line in Figure 3.6 that connects q with the bottom edge of C_{i+2}. If r was placed left of that line, we would get a contradiction to Snell's law of refraction.

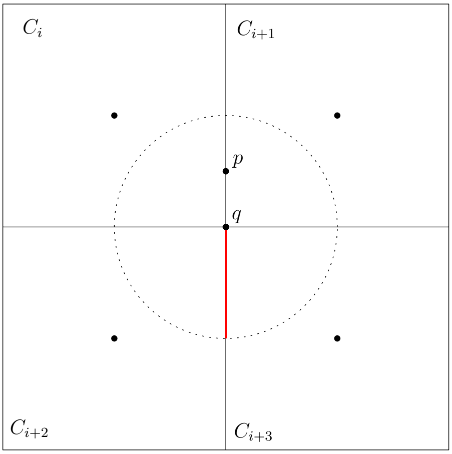


FIGURE 3.4: The situation in Case 1 with possible positions for r (red).

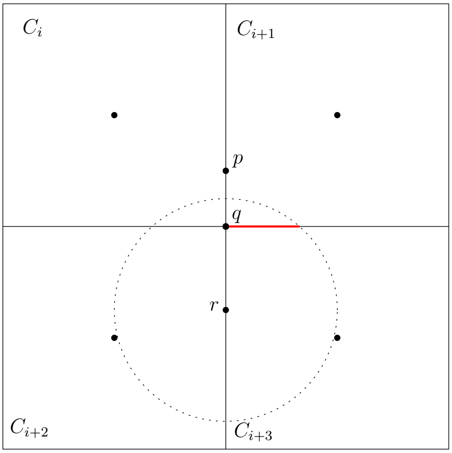


FIGURE 3.5: The situation in Case 1 with p, q, r being co-linear, and possible positions of the next bending point (red)

We can conclude that r lies right of that line. If $|l| \geq \frac{1}{2}$, then $C_{i+2} \in S'$, and the diagonal connection between the center points of C_i and C_{i+2} is a valid grid-path connection. We can then iterate the same arguments, with C_{i+2} taking the role of C_i. If $|l| < \frac{1}{2}$, then r must lie on the bottom edge of C_{i+1}, right of q. This means that π^* leaves C_{i+2} and re-enters C_{i+1}. Its next bending point after r lies either on the right or top edge of C_{i+1}. In any case, we know that the part of π^* that lies inside C_{i+1} is longer than $\frac{1}{2}$. Thus, $C_{i+1} \in S'$, and the connection between the centers of C_i and C_{i+1} is a valid grid connection.

From the above considerations, we can conclude that taking all cells from S that

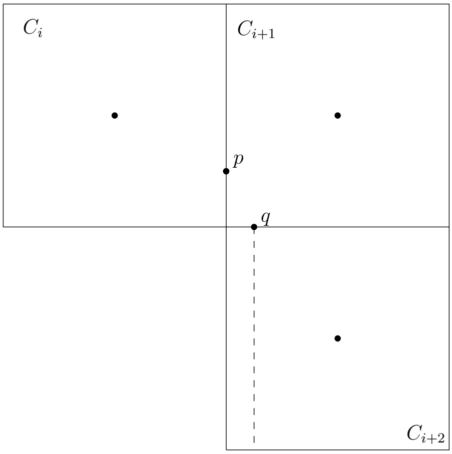


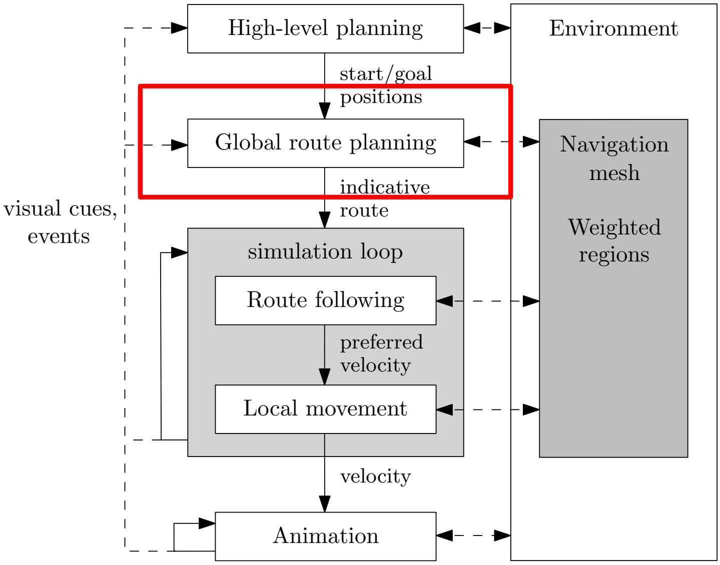
FIGURE 3.6: The situation in Case 2.

contain segments of the optimal path that are longer than $\frac{1}{2}$ yields a new sequence $S' = C'_1, ..., C'_{k'}$, for which we can connect all cell centers via valid grid-path connections. This gives as a grid path Γ', for which we know that its cost within a cell C_i with weight w_i are at most $\sqrt{2}\, w_i$. The cost of the optimal path in such a cell is at least $\frac{1}{2}\, w_i$. Thus, we can conclude that the cost per cell of Γ' is at most $2\sqrt{2}$ times the cost per cell of the optimal path, which concludes the proof of the thereom. $\square$

In the next chapter, we will continue with the topic of path planning in weighted regions. We will make use of the theoretical results of this chapter and present a novel real-time path planning algorithm named *Vertex-based Pruning* (VBP).

Chapter 4

Vertex-based pruning (VBP): A Hybrid Method



As we discussed in Section 2.3, a range of methods exist to solve the Weighted Region Problem approximately up to an arbitrarily small ϵ-error [2, 86, 87, 127]. These methods all work on the exact geometry of a scene, but they are computationally expensive and thus not suited for real-time applications with large numbers of virtual agents. By contrast, the A^* method [42] on a grid approximation of the scene is an efficient and easy-to-implement way to compute paths, but these paths do not account for the exact geometry of a weighted-region scene, and they are prone to errors induced by the grid approximation.

In this chapter, we combine the best of both worlds by presenting a novel hybrid method called *Vertex-based Pruning* (VBP). VBP efficiently computes a path that works on the exact geometry of a subset of a scene with weighted regions. The idea is to combine an efficient A^*-search [42] on a coarse grid representation of the scene with the ϵ-optimal *Steiner-point method* by Aleksandrov et al. [2].

We first summarize the method by Aleksandrov et al. [2] in Section 4.1. Subsequently, we present our new VBP method in Section 4.2, and we conduct experiments by testing the VBP method in several virtual environments; see Section 4.3.

This chapter is based on the following publication:

[55] N. Jaklin, M. Tibboel, and R. Geraerts. Computing high-quality paths in weighted regions. In *Proceedings of the 7th International ACM SIGGRAPH Conference on Motion in Games (MIG 2014)*, pages 77–86, 2014.

4.1 | The Steiner-point method (SPM) by Aleksandrov et al. 1998

In this section, we provide details on the Steiner-point method (SPM) by Aleksandrov et al. [2], which we have briefly discussed in Section 2.3.3. The method is part of our new VBP method, in which we use it on a subset of the polygonal subdivision.

The method works as follows: Given an instance of the WRP, i.e. a weighted polygonal subdivision, and an $\epsilon > 0$, the method transforms the problem into a graph-search problem by first triangulating the polygons and then adding Steiner points on the boundary edges of all triangles. The Steiner points serve as additional vertices for the graph, and additional edges are created between any two Steiner points that lie on two different edges of the same triangle. These new edges allow a path to cross a triangle in the same way as a theoretically optimal path would cross a region in a straight-line manner [87]. The points are placed in such a way that a path obtained from a graph-search method such as Dijkstra's algorithm [21] is proven to be an ϵ-approximation of an optimal path, which means that its costs are at most $(1 + \epsilon)$ times the costs of an optimal path. Since $\epsilon > 0$ can be chosen arbitrarily small, the method serves as an approximation scheme, which means that a user can control the error between the computed path and an optimal path. This has a price, though, because the smaller the error the more Steiner points are placed along the triangle edges. This yields an overall higher complexity of the graph and higher running times for the subsequent graph searches.

The key to achieve ϵ-approximate paths is to place the Steiner points in a logarithmic fashion along an edge. The points are placed in such a way that the distance between any two adjacent Steiner points along an edge is at most ϵ times the shortest possible path segment that can cross a triangle and intersect the edge between those two Steiner points. A theoretical problem that arises is that the shortest possible path segment near a vertex of a triangle becomes infinitesimally short. This problem is tackled by defining a *sphere* C_v, a circular area around each vertex v, in which no Steiner points are being added. The radius of C_v is denoted as

r_v, and it is defined as $r_v = \epsilon h_v$, where h_v is the minimum distance from v to the boundary of the union of its incident triangles; see Figure 4.1. Introducing C_v yields a lower bound on the length of the shortest possible edge that passes between two adjacent Steiner points, and it allows the placement of a finite number of Steiner points per edge.

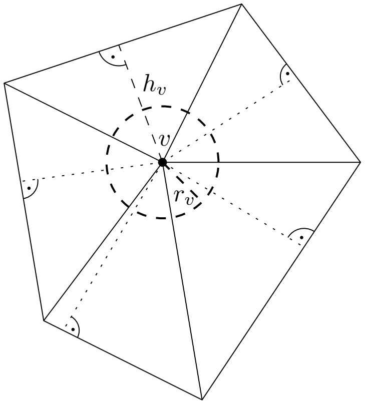


FIGURE 4.1: Example of a vertex v, h_v, and the resulting *sphere* around v with radius r_v.

The placement of Steiner points is done as follows: If v is a vertex with incident edges e_q and e_p and angle Θ_v between the edges, we place Steiner points $q_1, q_2, ...q_{\mu_q-1}$ along edge e_q and Steiner points $p_1, p_2, ...p_{\mu_p-1}$ along edge e_p, where $\mu_q = \log_\gamma(|e_q|/r_v)$ and $\mu_p = \log_\gamma(|e_p|/r_v)$. Here, $\gamma = (1 + \epsilon \sin \Theta_v)$, if $\Theta_v < \frac{\pi}{2}$, and $\gamma = (1 + \epsilon)$, otherwise. The Steiner points are placed such that the distance from v to each q_j is $r_v \gamma^{j-1}$. Furthermore, the distance between any two subsequent Steiner points q_i and q_{i+1} on edge e_q is proven to be ϵ times the distance from q_i to the opposite edge e_p; see Figure 4.2 (left).

The same placement strategy is done for all triangle vertices. This yields undesired overlaps of the intervals between Steiner points that are generated for opposite vertices of the same triangle edge. To resolve these overlaps, we determine the point on an edge where the interval sizes from each of the two sets are equal, and we eliminate all larger intervals.

Figure 4.2 shows an example of how the Steiner points are placed along the edges of a triangle. The placement of points for one edge are shown on the left, whereas the situation for all three edges with resolved overlaps is shown on the right.

With the above placement of Steiner points, the final graph is constructed by connecting any two Steiner points that lie on edges that belong to the same triangle. Aleksandrov et al. [2] show that ϵ-optimal paths can then be obtained by performing a classical graph search on the resulting Steiner-point graph.

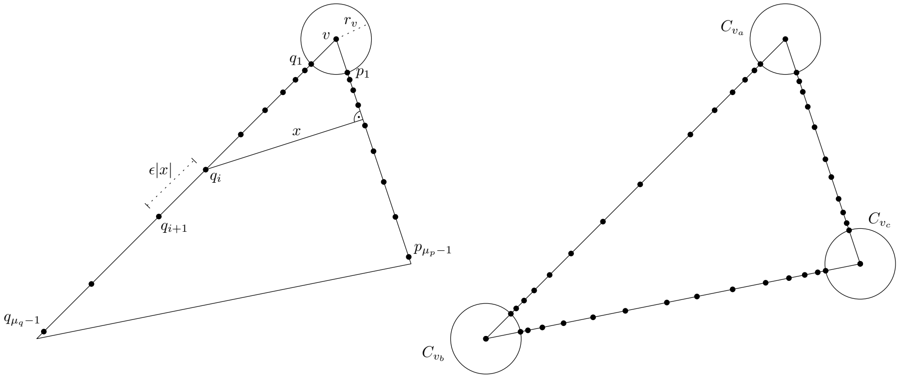


FIGURE 4.2: *Left*: The placement of Steiner points $q_1, q_2, ...q_{\mu_q-1}$ and $p_1, p_2, ...p_{\mu_p-1}$ for edges e_q and e_p that are incident to vertex v. The distance between q_i and q_{i+1} is ϵ times the distance from q_i to the opposite edge e_q. *Right*: The placement of points for all incident edges of all three vertices v_a, v_b, v_c, where overlaps of Steiner point intervals are resolved. By $C_{v_a}, C_{v_b}, C_{v_c}$, we denote the *spheres* around the corresponding vertices that remain void of Steiner points.

4.2 | THE VBP METHOD

In this section, we describe the idea behind our main contribution of this chapter: the *Vertex-based Pruning* (VBP) method. Assume we are given an instance of the Weighted Region Problem and an arbitrary error bound $\epsilon > 0$. The goal is to compute a path between a start and goal position for which the costs are at most $(1+\epsilon)$ times the costs of an optimal path between the same points. In addition, the computation of the path should be efficient such that the path can be computed in real time in a simulation or gaming application.

The intuitive idea behind the VBP method is as follows: We would like to exploit the efficiency of an A^* grid search to quickly determine regions that are interesting candidates for performing a more thorough search. By interesting, we mean that a theoretically optimal path will likely also cross the same regions. To be precise, we do the following. We first run an A^* search on a grid approximation of the scene. We then use the resulting grid path to determine a set of vertices and edges of the given polygonal subdivision that are close to the grid path. We have tested three different variants of how we define *close* in this context: triangle-based, edge-based, and vertex-based. Next, we prune the overall scene by only keeping the vertices and edges that we determined in the previous step. The Steiner-point method by Aleksandrov et al. [2] does not depend on a polygonal subdivision but works on a graph structure. As a consequence, we do not need to guarantee that the pruned graph is still a polygonal subdivision. It suffices to guarantee that we store the information of what edges belong to the same polygon in the overall scene and what weights each of these edges inherits from the polygonal subdivision.

VBP is described as pseudocode in Algorithm 1. To guarantee grid optimality for the grid path Γ in the first step of the algorithm, we need to use an admissible heuristic for the A^* search. A heuristic is admissible, if it never overestimates the actual costs from a node to the goal; see Section 2.2. If all weights are greater than 1, we can use the Euclidean distance to the goal as an admissible heuristic. If the given instance of the WRP features weights smaller than 1, we can multiply the Euclidean distance from each grid cell to the goal with the minimum weight w_{min} to obtain an admissible heuristic.

We will now describe three pruning heuristics. We start with discussing triangle-based pruning in Section 4.2.1. We then discuss edge-based pruning in Section 4.2.2. Finally, we discuss vertex-based pruning in Section 4.2.3, which yielded the best results and gives the method its name.

4.2.1 | TRIANGLE-BASED PRUNING

As a first pruning heuristic, we discuss triangle-based pruning. We trace the previously computed grid path Γ through the scene and collect all triangles of the polygonal subdivision that are intersected by Γ. If Γ runs along an edge or a vertex of the original scene, we collect both triangles that share this edge or all triangles that share this vertex. This is a straight-forward approach, which creates a subset of all triangles in the scene. Figure 4.3 shows an example of the resulting graph in a scene called *Puddle*, which we used in our experiments; see Section 4.3.

The approach trivially guarantees that the resulting graph is connected, and for each edge in the resulting graph, the other two edges of the same triangle are also contained in the graph. Thus, when we place Steiner points on the triangle edges according to the placement strategy by Aleksandrov et al. [2], we know that for each triangle edge e there is a corresponding edge e' of the same triangle in the pruned graph, which allows us to connect the Steiner points on e with the Steiner points on e'.

Algorithm 1 VERTEX-BASED PRUNING (VBP)

Input. weighted polygonal environment with edges E, vertices V, weights W; start position s and goal position g; error bound $\epsilon > 0$.
Output. a path π between s and g that is ϵ-optimal in the same homotopy class as a grid-optimal path between s and g.

```
$\Gamma \leftarrow$ run $A^*$ on a grid with weights $W$ and an admissible heuristic.
$E' \leftarrow$ PRUNEGRAPH($\Gamma, E, V$)
$\pi \leftarrow$ run the Steiner-point method on $E'$ with error bound $\epsilon$.
return $\pi$
```

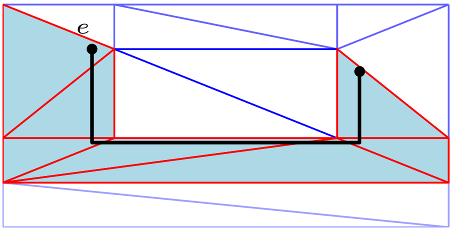


FIGURE 4.3: Example of triangle-based pruning.

While this pruning strategy is correct in the sense that the Steiner-point method can be performed on the resulting graph, it is not informed in the sense that the resulting graph still contains edges and Steiner points that are most likely not used by the final ϵ-approximate path. An example is edge e in Figure 4.3 because a path starting in the corresponding triangle will not use any of the Steiner points on edge e or on any other edge that forms the boundary of the pruned scene. If such an edge was used in an optimal path, it would also be used by the grid-optimal path Γ for a small enough grid resolution, and thus the pruned scene would be larger and contain the edge as a non-boundary edge.

4.2.2 | EDGE-BASED PRUNING

The second pruning heuristic we discuss is edge-based pruning. Instead of collecting all triangles that are intersected by the grid-optimal path Γ, we collect all edges that are closest to Γ. To be precise, for each grid vertex v that lies on Γ, we collect all edges that are closest to v in the Euclidean sense. Thus, if v coincides with a vertex of the triangulated scene, we collect all edges that are spawned from that vertex. Figure 4.4 shows the resulting graph of edge-based pruning in the *Puddle* scene.

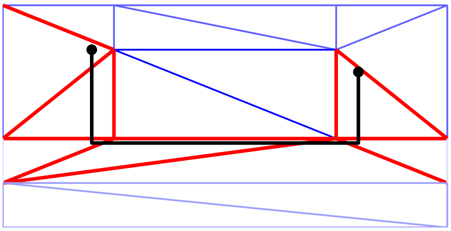


FIGURE 4.4: Example of edge-based pruning.

The graph obtained from edge-based pruning is connected. To see this, note that the grid-optimal path Γ induces a connected sequence of intersected triangles. For each such triangle, Γ leaves the triangle by intersecting one of the triangle edges or vertices. This edge, or all edges spawning from that vertex, are contained in the pruned graph. Since Γ can never leave a triangle via the same edge or vertex through which it entered a triangle, Γ leaves a triangle via an edge that is adjacent to the one through which it entered. As a result, the pruned graph is connected. In particular, similar to the triangle-based pruning heuristic, for each triangle edge in the pruned graph there is another edge from the same triangle in the graph. Thus, the heuristic is correct in the sense that the Steiner-point method can be performed on the pruned graph.

The graph obtained from edge-based pruning tends to contain less edges than the one obtained from triangle-based pruning. Though there are still triangles for which all three edges are contained in the pruned graph, there are also triangles for which only two edges are contained in the pruned graph. This happens whenever the third edge is never the closest to any grid vertex on Γ; see Figure 4.4. In this sense, edge-based pruning is more informed than triangle-based pruning.

However, both triangle-based pruning and edge-based pruning have a common disadvantage. Both heuristics might prune edges from the original scene that are neither used by a theoretically optimal path nor a grid-optimal path, but that are indeed used by an ϵ-approximate path when using the Steiner-point method. An example of such a case is a theoretically optimal path that runs close to a graph vertex v such that it intersects the *sphere* around v, which by definition does not contain any Steiner points [2]. The third heuristic, vertex-based pruning, overcomes this problem.

4.2.3 | VERTEX-BASED PRUNING

The third pruning heuristic works as follows: We iterate over all *bending points* of Γ. By bending point, we refer to any grid vertex on Γ at which the path changes its direction. The start and goal vertex could be seen as degenerate cases of bending points. However, defining them as such leads to computing unnecessary edges and vertices that are not relevant for the final solution, similar to the triangle-based pruning heuristic. Thus, we do not consider the start and goal vertices as bending points. For each bending point b, we compute vertices of the environment that are closest to b in the Euclidean sense. We then take all triangle edges incident to these vertices. Note that this set of edges does not yet need to form a connected graph. To address this, we also add all edges that Γ intersects, if they have not already been added during the first step of the pruning. This ensures that the resulting graph is connected. Figure 4.4 shows the resulting graph of vertex-based pruning in the *Puddle* scene.

The actual pruning step is described in Algorithm 2. After the pruning step, we run the Steiner-point method on the pruned graph. The Steiner-point method is proven to compute ϵ-optimal paths for any given $\epsilon > 0$ [2]. Thus, the VBP method

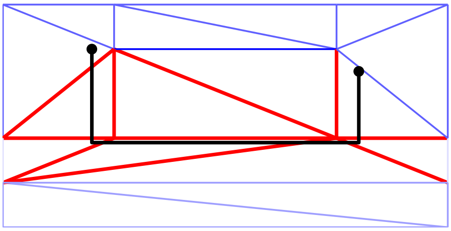


FIGURE 4.5: Example of vertex-based pruning.

Algorithm 2 PRUNEGRAPH

Input. grid path Γ; environment with edges E and vertices V.
Output. pruned environment as a set of edges E'.

```
initialize $E'$ as an empty set
for all bending points $b$ of $\Gamma$ do
    Find the set $closestV$ of closest vertices from $b$ in $V$
    for all vertices $v$ in $closestV$ do
        for all edges $e$ incident to $v$ do
            Add $e$ to $E'$
for all segments $s$ of $\Gamma$ do
    for all edges $e$ in $E$ do
        if $s$ intersects $e$ then
            Add $e$ to $E'$
return $E'$
```

will always compute a path that is guaranteed to be ϵ-optimal in the pruned scene induced by the grid-path Γ. As discussed in Section 3.3, an overall optimal path and a grid-optimal path do not always need to be region-homotopic. However, in our experiments in the next section, we empirically determined that the number of cases is small in which the two paths are not region-homotopic.

4.3 | EXPERIMENTS

In this section, we discuss the experiments we have conducted to validate our new VBP method. We have compared the Steiner-point method (SPM) by Aleksandrov et al. [2] with our new VBP method. To this end, we have compared the quality of the computed paths based on their costs and based on visual inspection. In addition, we have measured the performance of both methods. The experiments have been conducted on an *AMD Phenom II*$^{\mathrm{TM}}$ $x4$ 3.4 Ghz processor with 4 GB RAM and an NVIDIA GeForce GTX 650 graphics card.

In Section 4.3.1, we introduce the scenarios that we have tested. In Section 4.3.2, we compare the graph-construction times and the graph-query times of the SPM and VBP independently for a set of queries in four small and one medium-sized scene. In Section 4.3.3, we measure empirically how often our VBP method computes the same ϵ-optimal path as the SPM for a given query in the first five scenes. In Section 4.3.4, we use a large scene to compare the query times of the SPM with the combined computation times of VBP for constructing the pruned graph and searching the pruned graph.

4.3.1 | THE TESTED SCENES

We have tested the VBP method on six different scenes. The first four scenes are small with a size of 100×50 units. The fifth scene named *forest* is medium-sized with 410×290 units. The sixth scene named *abstractRegions16x16* is large with 800×800 units. It consists of a rectangular arrangement of 16×16 smaller tiles that each span 50×50 units. All weights were chosen manually. For the first five scenes, the weights were chosen to either simulate a real-world example scene or to test specific properties of our VBP method. For the large scene, the abstract regions and their weights were chosen arbitrarily.

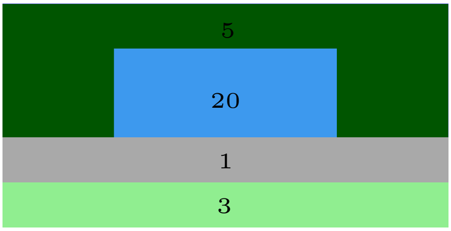


FIGURE 4.6: The *Puddle* scene.

The first scene is called *Puddle*; see Figure 4.6. It resembles a puddle of water in the center with weight 20, surrounded by a forest with weight 5. Below the puddle runs a road with weight 1, and below the road is a grass lawn with weight 3. This scene is small and simple, which makes it easy to visually check the computed paths. In addition, it resembles a typical scenario that could occur in a gaming or simulation application.

The second scene is called *Bars*; see Figure 4.7. It features several vertical bars with alternating weights of 1 and 5. The triangles span the whole height of the scene, which makes it an interesting test case for the VBP method: We expect the VBP method to result in a graph that is close to the Steiner graph of the whole scene.

The third scene is called *High-low*; see Figure 4.8. It features a high-cost region with weight 20 at the bottom, a medium-weight region in the center with weight 3, and

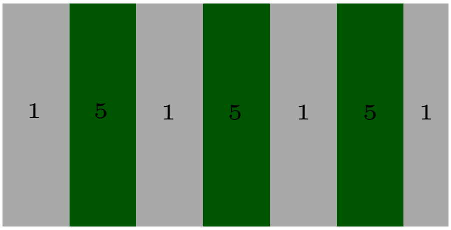


FIGURE 4.7: The *Bars* scene.

a low-cost region at the top with weight 1. The scene is well-suited to display a property of the Weighted Region Problem that does not occur in the classical Path Planning Problem: An optimal path can cross the same triangle multiple times. If s and g both lie in the high-cost region, an optimal path might use parts of the low-cost region at the top.

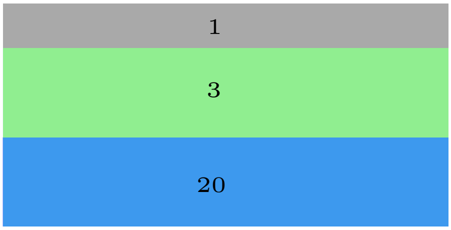


FIGURE 4.8: The *High-low* scene.

The fourth scene is called *Zigzag*; see Figure 4.9. It resembles a sandy road with weight 6, a grass field above it with weight 3, and alternating parts of road with weight 1 and water with weight 20. This is an interesting test case: An optimal path from left to right should follow the sandy road, but alternate between the top and bottom border of this region, resulting in a zig-zag path.

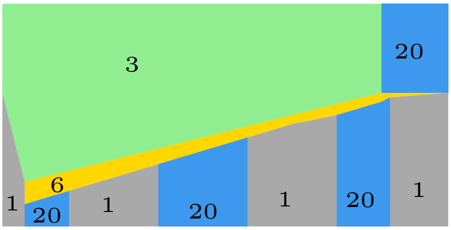


FIGURE 4.9: The *Zigzag* scene.

The fifth scene is called *Forest*; see Figure 4.10. It resembles a path with weight 3 that runs through a deep forest with weight 99. There are several puddles on the way with weight 20. At the top of the scene, there is an attractive spot such as a panoramic view over a valley with weight 1. The two parallel rectangular regions near the center resemble fallen tree logs that block the path, but can be traversed (by climbing or ducking) with weight 2. The scene is slightly larger than the other four, and it resembles a real-life scenario that could occur in a gaming or simulation application.

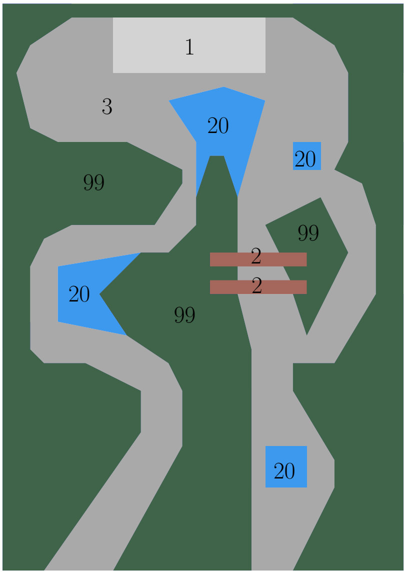


FIGURE 4.10: The *Forest* scene.

The sixth scene is called *abstractRegions16x16*. It consists of 16×16 tiles that each span 50×50 units; see Figure 4.11. The regions and their weights in this scene were chosen arbitrarily: gray 4, darkgreen 30, lightgreen 3, blue 30, yellow 6, and brown 2. The goal is to test whether we achieve an improved query-time performance of VBP over the method by Aleksandrov et al. when a scene contains a large number of triangles, which is usually the case in a gaming or simulation application.

4.3.2 | VBP VS. SPM IN SMALL SCENES

We have compared the computation times of the SPM against our VPB method in the first five scenes for one query per scene, which was run 50 times per query point. For each of the five tested scenes, we have used 5 different ϵ-error bounds ranging between 0.1 and 0.5, and we have compared the construction times of the graphs, the times needed to answer path planning queries on the graph, the number of nodes explored during the search, and the overall costs of the resulting paths. For the A^*-grid search of our VBP method, we used a grid-cell width and height

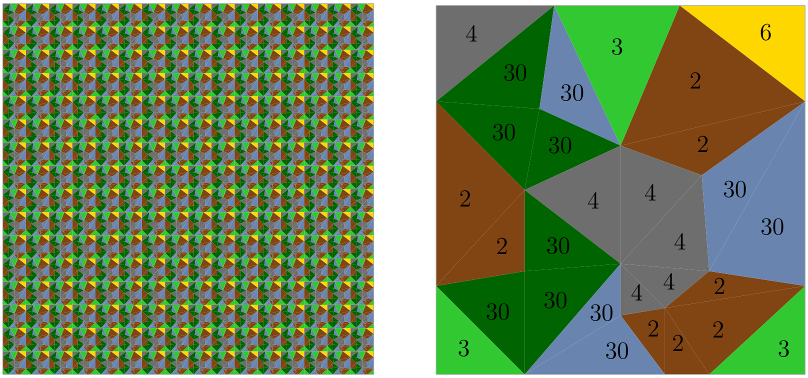


FIGURE 4.11: The *abstractRegions16x16* scene (left). It consists of 16×16 rectangular tiles that each span 50×50 units (right).

of 1 unit. The particular query points and the resulting paths for each scene are displayed in Figure 4.12.

The goal of the first experiment is to test our pruning heuristic and whether VBP shows the following expected behavior: For queries that affect a large portion of the overall number of triangles in a given scene, we expect VBP to not show any significant improvements – if at all – in both graph-construction times and graph-query times. This is particularly probable in small scenes such as *Bars* or *High-low*. When, by contrast, a query only affects a small portion of the total number of triangles in a given scene, we expect VBP to improve on both the graph-construction and graph-query times due to the overall smaller graph after the pruning step and the resulting lower number of Steiner points. Among the first five scenes, we expected this to be particularly the case in the *Forest* scene.

Table 4.1 shows the results of the first experiment. As expected, VBP does not outperform the Steiner-point method in the *Bars* scene. The scene contains triangles that span the whole height of the scene, and paths are planned from the left side to the right side. The pruned graph is therefore almost as big as the graph of the whole scene. In this case, the additional time to perform an A^* search for pruning the scene dominates the time that can be saved for querying the pruned graph. The larger the ϵ-error bound the fewer Steiner points are required. With fewer required Steiner points, the difference in the number of Steiner points on the whole graph and the pruned graph is small. This yields higher construction times for VBP and ϵ ranging from 0.3 to 0.5. Thus, the corresponding improvement in query times is comparably small.

VBP performs slightly better in the *High-low* scene. It shows a greater improvement on query times and the number of explored nodes than in the *Bar* scene. However, due to the geometry of the scene, the construction time is still slightly higher compared to the original Steiner-point method for $\epsilon = 0.5$.

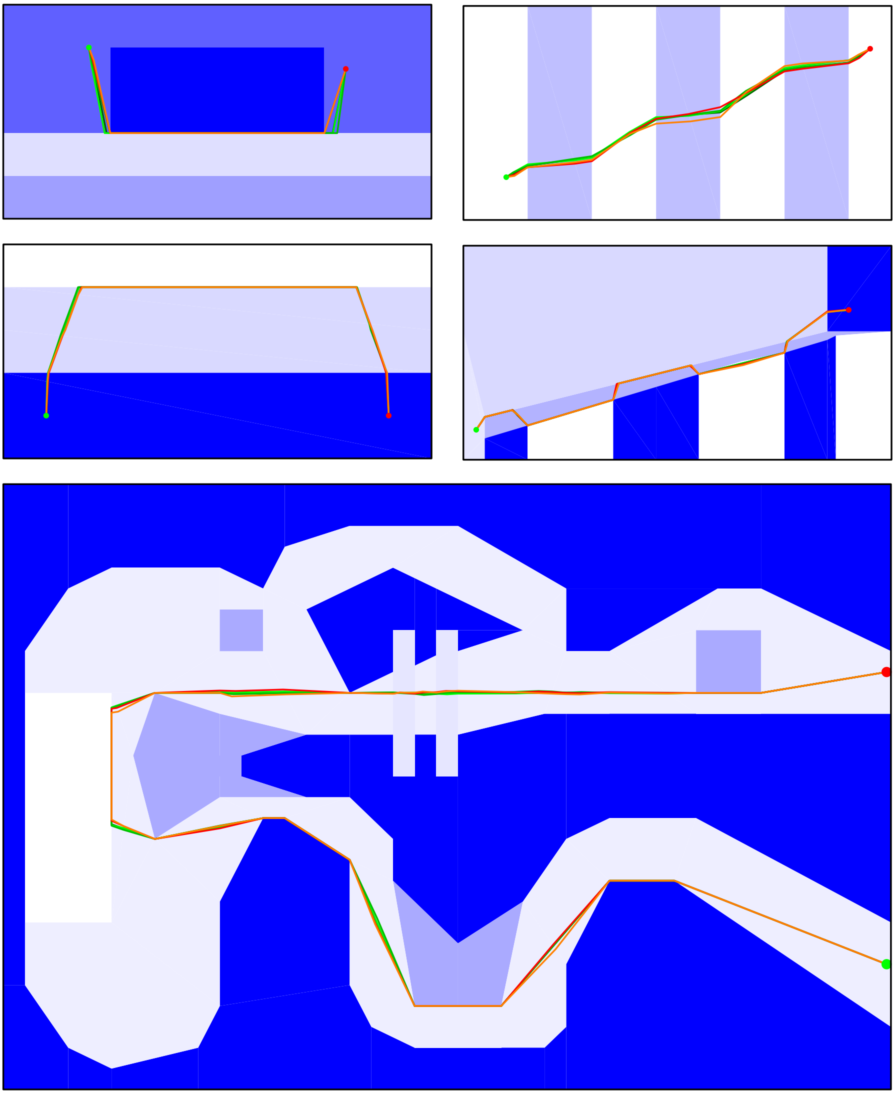


FIGURE 4.12: The paths computed with the SPM and the VBP method for different ϵ-error bounds: $\epsilon = 0.1$ (green), $\epsilon = 0.2$ (lightgreen), $\epsilon = 0.3$ (darkgreen), $\epsilon = 0.4$ (red), and $\epsilon = 0.5$ (orange). Note that both methods computed the same paths in all shown cases. In this representation of the scenarios, the higher the weight for a region the darker its shade of blue.

Scene	ϵ	Method	Constr. time (ms)	Query time (ms)	Nodes explored	Path costs
Puddle	0.1	SPM	49911.9	222.3	7441	231.5
		VBP	**18396.8**	**77.6**	**3561**	231.5
	0.2	SPM	2943.4	29.0	2856	231.5
		VBP	**1084.5**	**10.7**	**1367**	231.5
	0.3	SPM	617.7	8.7	1560	231.8
		VBP	**283.1**	**3.1**	**751**	231.8
	0.4	SPM	255.3	3.6	986	232.2
		VBP	**164.2**	**1.4**	**479**	232.2
	0.5	SPM	167.0	1.8	675	232.3
		VBP	**125.0**	**0.7**	**331**	232.3
Bars	0.1	SPM	6993.7	62.8	4908	273.6
		VBP	**5281.6**	**48.5**	**3957**	273.6
	0.2	SPM	450.3	8.4	1774	273.8
		VBP	**394.6**	**6.7**	**1468**	273.8
	0.3	SPM	**166.1**	2.4	909	273.8
		VBP	171.9	**1.9**	**776**	273.8
	0.4	SPM	**117.6**	0.9	542	274.0
		VBP	136.5	**0.8**	**475**	274.0
	0.5	SPM	**120.7**	0.5	370	274.2
		VBP	131.5	**0.4**	**329**	274.2
High-low	0.1	SPM	127198.0	471.1	7887	592.7
		VBP	**117328.3**	**448.8**	**7409**	592.7
	0.2	SPM	7895.9	60.9	3016	592.8
		VBP	**7245.3**	**59.9**	**2721**	592.8
	0.3	SPM	1409.0	17.7	1648	592.8
		VBP	**1335.7**	**16.6**	**1494**	592.8
	0.4	SPM	487.5	7.3	1039	592.8
		VBP	**475.6**	**6.9**	**946**	592.8
	0.5	SPM	**251.8**	3.5	723	592.9
		VBP	252.2	**3.4**	**662**	592.9
Zigzag	0.1	SPM	438767.4	2238.3	26245	343.6
		VBP	**303882.8**	**1789.0**	**21004**	343.6
	0.2	SPM	28167.3	256.5	10334	343.6
		VBP	**19252.0**	**203.3**	**8333**	343.6
	0.3	SPM	5089.8	72.8	5796	343.7
		VBP	**3547.5**	**57.8**	**4717**	343.7
	0.4	SPM	1591.8	30.0	3765	343.8
		VBP	**1192.1**	**24.2**	**3096**	343.8
	0.5	SPM	683.3	15.3	2645	343.8
		VBP	**565.3**	**12.4**	**2199**	343.8
Forest	0.1	SPM	163325.0	1141.1	47969	2461.2
		VBP	**16857.2**	**224.6**	**15413**	2461.2
	0.2	SPM	9638.0	127.8	17710	2461.4
		VBP	**1049.6**	**26.8**	**5700**	2461.4
	0.3	SPM	1794.7	34.3	9370	2461.9
		VBP	**351.9**	**7.8**	**3031**	2461.9
	0.4	SPM	600.6	13.4	5744	2462.5
		VBP	**245.0**	**3.1**	**1863**	2462.5
	0.5	SPM	300.4	6.4	3826	2464.2
		VBP	**225.4**	**1.4**	**1243**	2464.2

TABLE 4.1: Comparison of the Steiner-point method (SPM) by Aleksandrov et al. [2] and the VBP method on all five scenes with ϵ-error bounds ranging from 0.1 to 0.5.

In the *Zigzag*, *Puddle*, and particularly the *Forest* scene, the difference between the pruned graph and the initial graph is big. The time that can be saved to explore parts of the graph that are not relevant strongly dominates the additional time required for the initial A^* search. This yields an overall improvement in both construction times, query times, and the number of nodes explored during the search.

The following conclusions can be drawn from this first experiment. Compared to the original Steiner-point method, the VBP method needs additional time to perform the initial A^* search on the whole scene. As expected, VBP needs less time to construct a graph than the SPM when the number of pruned triangles of the original scene is large. Once the pruned graph is constructed for a particular query, VBP improves on the path-computation times over the SPM in correspondence to the number of triangles that have been pruned.

Furthermore, all paths computed with VBP in this first set of experiments are equal to the paths computed with the SPM. This is, however, not necessarily the case in general because the initial grid-path and an optimal path might not be region-homotopic as discussed in Section 3.3. We have conducted a second experiment to empirically determine how often this is the case.

4.3.3 | Empirical analysis of path differences

We executed the SPM and all three pruning methods for a large number of path queries on each of the first five scenes. We measured the overall number of paths that yielded different costs for the pruning heuristics and the SPM. The goal was to empirically determine in how many cases the paths computed on a pruned scene and on the entire scene are not region-homotopic. We randomly sampled each scene uniformly to generate different start and goal positions for each query. For the first four scenes, we picked 9 different x- and y-coordinates, yielding 81 start and 81 goal positions. For each combination of start and goal positions, we computed a path with both methods, yielding a total of 6561 different paths for each of the four scenes. For the larger *Forest* scene, we generated a total of 9000 different paths with both methods by uniformly sampling start and goal positions.

Table 4.2 shows the results of this experiment. On average, vertex-based pruning yielded the best results. It computed the same paths as the SPM in 97.3% of all cases. This further justifies our theoretical analysis that vertex-based pruning performs best when taking ϵ-approximate paths computed by the Steiner-point method on the entire scene as a ground truth. We can conclude that the same ϵ-error bound as for the Steiner-point method applies to VBP paths in most tested cases. In the few remaining cases, VBP paths are still ϵ-optimal with respect to an optimal path in the pruned scene.

Pruning heuristic	**Scene**	**Number of paths**	**Number of path differences**	**Success rate**
Triangle-based	Puddle	6561	18	99.7%
	Bars	6561	0	100%
	High-low	6561	128	98.0%
	Zigzag	6561	501	92.4%
	Forest	9000	1623	82.0%
	All averaged	35244	2270	93.6%
Edge-based	Puddle	6561	26	99.6%
	Bars	6561	0	100%
	High-low	6561	128	98.0%
	Zigzag	6561	399	93.9%
	Forest	9000	590	93.4%
	All averaged	35244	1143	96.8%
Vertex-based	Puddle	6561	23	99.6%
	Bars	6561	0	100%
	High-low	6561	132	97.9%
	Zigzag	6561	201	96.9%
	Forest	9000	587	93.4%
	All averaged	35244	**943**	**97.3%**

Table 4.2: Empirical comparison of the three pruning heuristics in all five scenarios. The success rate indicates how often a pruning heuristic yielded the same path as the original Steiner-point method by Aleksandrov et al. [2].

Overall, the results of our first two experiments are an indication that VBP can answer path-planning queries faster than the the original Steiner-point method when the underlying graph has already been constructed, and that VBP computes the same paths as the SPM in small scenes and for a comparably small grid resolution. However, VBP needs to construct a new pruned graph for each path-planning query, whereas the original Steiner-point method computes a graph only once as an offline step. Furthermore, the results do not indicate whether VBP also computes the same paths as the SPM when we use a larger scene and a correspondingly larger grid resolution. To address these two points and further validate our VBP method, we have therefore conducted a third experiment on the sixth scene named *abstractRegions16x16*.

4.3.4 | VBP vs. SPM in a large scene

In the third experiment, we tested whether the time needed to construct the pruned graph plus the time needed for searching the graph is lower than the query times of the original Steiner-point method alone. This property is expected to hold only in large scenes, and it is thus not reflected in our first experiment. At the same time, this property is key to achieve real-time performance in future gaming and simulation applications. To this end, we have tested both methods with ϵ-values of $0.3, 0.4$, and 0.5 for 65 pairs of query points in the *abstractRegions16x16* scene, which yielded a total of 195 paths. Both methods were run 10 times for each query point and each ϵ-value. The query points were determined by picking a set of 20 candidate points in a grid-like fashion such that possible paths between any two candidate points would vary in length and overall running direction. From the set of candidate points, we picked four diagonally-opposite pairs to test our expectation that VBP outperforms the SPM for these queries. For the remaining 61 query points, we randomly picked pairs from the set of candidate points. Since this large scene is 16 times as big as the first four scenes, we scaled the grid-cell width and height for the A^* search up to 16 units to ensure a comparable performance of our VBP method as in the first set of experiments.

Table 4.3 shows an excerpt of the averaged results. We show two queries for which VBP did not outperform the SPM and two other queries in which VBP did outperform the SPM. Overall, we can draw the following conclusions from this third experiment: First, the query times of the SPM for ϵ-values of 0.4 and 0.5 are still slightly lower than the construction plus query times for our VBP method. Second, the opposite is true when using an ϵ value of 0.3 and when the paths are sufficiently long, e.g. when they span almost the entire diagonal of the scene. In these cases, the sum of the construction and query times for VBP is smaller than the query time for the SPM alone. This turned out to be the case for the four queries we purposely picked, and for two more queries from the remaining set of query points, which were close to diagonally-opposite pairs. In all these cases, the increased number of Steiner points that are placed over the entire scene by the SPM yields query times that cannot compete with the overall time needed for running VBP. For the remaining query points, the SPM yielded lower query times than the overall time needed to run VBP. The key factor turned out to be the number of triangles that the SPM needs to traverse after the graph has been constructed.

Due to the design of both methods, we expect the shown improvement of VBP over the SPM to be greater when using even larger scenes, smaller ϵ-values, and query points that yield longer (not necessarily more expensive) paths. We leave further validation of these expectations for future work.

Contrary to our experiment in the first five scenes, VBP turned out to compute paths that all have higher costs than the ones computed with the SPM. While both sets of paths are proven to be ϵ-approximations of their theoretical optima [2], the optimal paths for the two methods are not the same due to the pruning step of VBP. This relates to our analysis in the previous chapter, in which we have shown that an optimal path and a grid-optimal path do not need to be region-homotopic. The differences in path costs we observed were small for the majority of queries, with some large outliers: On average, we observed an increase in path costs of 21.1% with a large standard deviation of 37.9 due to the fact that five out of 195 queries yielded an increase in path costs of more than 200%. A total of 159 out of all 195 queries yielded path differences below the mean of 21.1%, with a typical range being an increase between 1% and 10%.

Figure 4.13 shows an example query from $(10, 10)$ to $(700, 700)$ and the corresponding paths as computed by the SPM and by our VBP method. The SPM path has total costs of 2148.86, whereas the VBP has total costs of 2216.25, which corresponds to an increase of 3% in this particular example. Figure 4.14 shows the largest outlier that yielded costs of 6593.18 for VBP and costs of 1971.08 for the SPM, which corresponds to an increase of 235%. However, when visually comparing both examples, the path differences do not seem as extreme as the cost numbers indicate. These higher costs occurred in only five out of 195 queries, while the corresponding paths are still visually convincing. This gives rise to the conjecture that the high costs for the VBP path might result from numerical rounding errors. These errors might be due to the *edge-crawling* path segments that touch the high-cost region shown in blue and the fact that the shown example contains a comparably large number of such edge-crawling segments.

Query	ϵ	**Method**	**Constr. time (ms) [StDev]**	**Query time (ms) [StDev]**	**Nodes**	**Path costs**
$(10, 10)$	0.3	SPM	38004.1[91.7]	135.2[0.7]	34383	1016.6
$(10, 400)$		VBP	1324.0[20.3]	5.7[0.1]	1997	1120.2
	0.4	SPM	10406.5[40.2]	49.8[0.2]	20535	1016.9
		VBP	1274.9[3.9]	3.8[0.1]	1193	1119.6
	0.5	SPM	4174.14[11.7]	24.4[0.2]	13152	1032.4
		VBP	1262.8[4.8]	3.1[0.0]	777	1133.9
$(10, 10)$	0.3	SPM	38087[374.2]	460.6[3.6]	95648	1482.5
$(10, 700)$		VBP	1380.4[20.0]	15.9[0.1]	3455	1714.2
	0.4	SPM	10352[12.5]	161.4[1.7]	57262	1482.9
		VBP	1302.4[3.2]	13.5[0.1]	2069	1713.6
	0.5	SPM	4166.0[6.2]	72.4[1.4]	36661	1498.6
		VBP	1276.8[3.3]	11.2[0.1]	1353	1727.9
$(20, 700)$	0.3	SPM	38274[154.9]	1975.2[8.5]	298022	1560.1
$(700, 20)$		**VBP**	**1608.8 [16.9]**	**35.8 [1.1]**	**7299**	**1693.5**
	0.4	SPM	10442.5[16.6]	645.2[3.4]	177984	1561.7
		VBP	1372.1[3.3]	25.5[0.1]	4306	1695.5
	0.5	SPM	4201.0[6.4]	264.7[2.0]	113956	1569.4
		VBP	1313.6[2.7]	21.8[0.1]	2708	1703.2
$(10, 500)$	0.3	SPM	38103[186.4]	1968.6[1.9]	288428	1476.5
$(700, 10)$		**VBP**	**1795.1 [12.5]**	**42.8 [0.6]**	**8603**	**1566.17**
	0.4	SPM	10411.4[22.1]	639.3[2.3]	172778	1478.5
		VBP	1443.4[7.5]	30.8[1.3]	5182	1568.1
	0.5	SPM	4183.1[3.2]	263.1[2.2]	110703	1479.6
		VBP	1362.3[20.2]	24.2[0.3]	3347	1568.3

TABLE 4.3: Excerpt of the comparison between the SPM and VBP on the *abstractRegions16x16* scene with ϵ-error bounds ranging from 0.3 to 0.5. Lines are bold in which the sum of the construction time and the query time of VBP is smaller than the query time of the SPM alone.

ϵ	**# Queries**	**Path-cost increase (%) [StdDev]**
0.3	65	21.3[38.4]
0.4	65	21.2[38.3]
0.5	65	20.9[37.7]
all	**195**	**21.1% [37.9]**

Table 4.4: Overall results of the comparison between the SPM and VBP on the *abstractRegions16x16* scene.

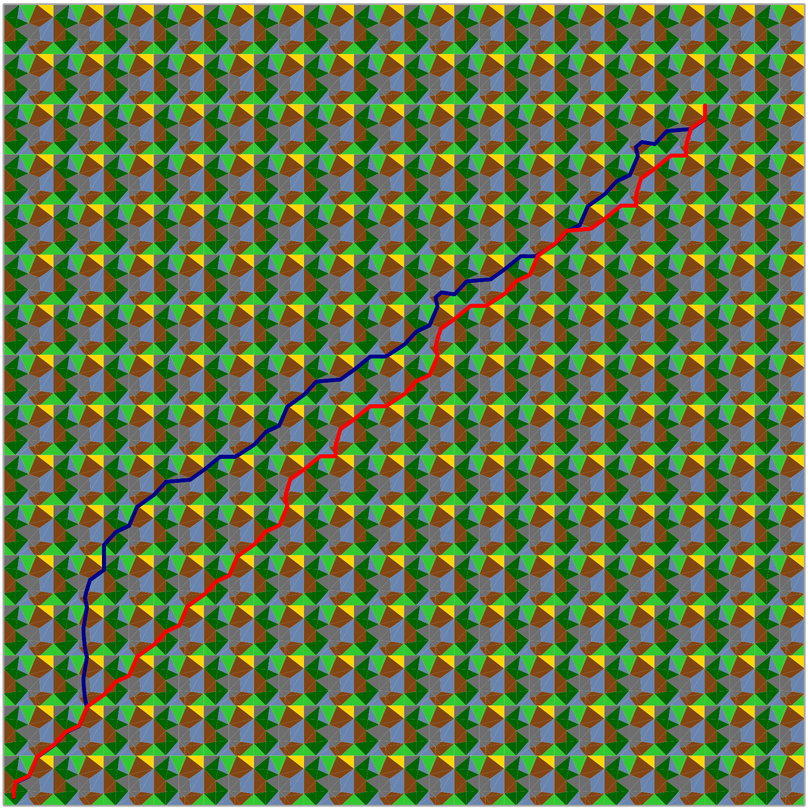


Figure 4.13: Two different paths that we computed in the *abstractRegions16x16* scene. The path computed with our VBP method (dark blue) has total costs of 2216.25, and the path computed with the SPM (red) has total costs of 2148.86.

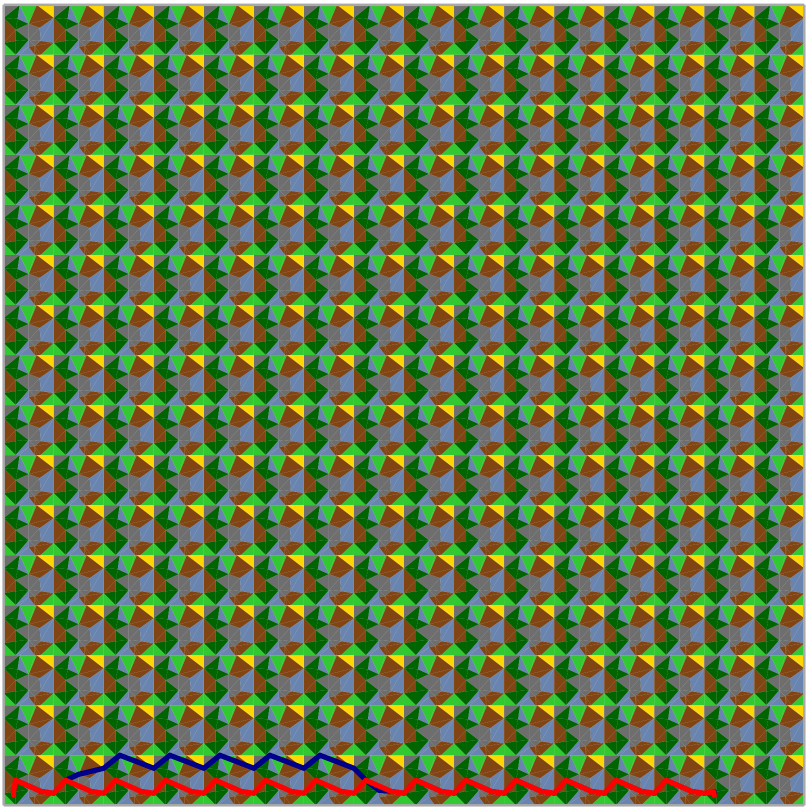
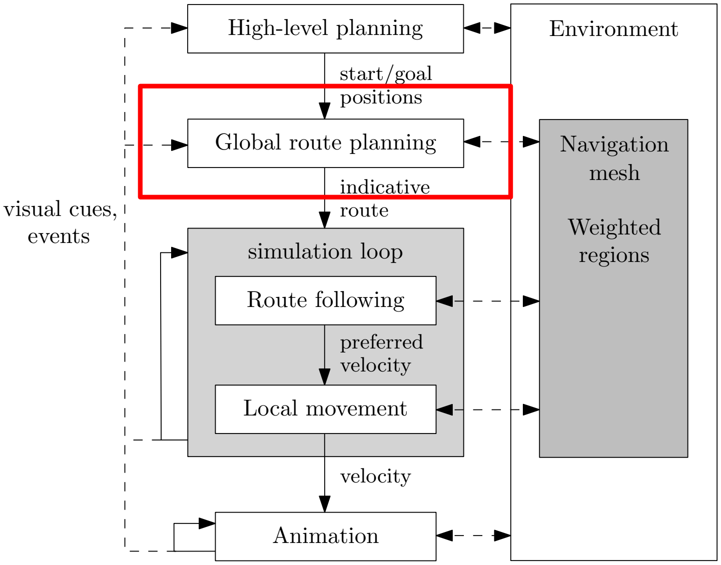


Figure 4.14: A path-cost outlier in the *abstractRegions16x16* scene. The path computed with our VBP method (dark blue) has total costs of 6593.18, and the path computed with the SPM (red) has total costs of 1971.08. We conjecture that the large increase in costs might be due to numerical rounding errors caused by the large number of edge-crawling segments at the boundary of the high-cost regions shown in darkgreen and blue.

Conclusion Part I



This chapter concludes the first part of this thesis: path planning in weighted regions. We have discussed the Weighted Region Problem (WPR) [87] under the aspect of virtual-agent navigation for advanced crowd behavior in future simulation and gaming applications.

In Chapter 3, we analyzed paths that are obtained from a grid representation of a weighted polygonal subdivison. We started with a simple observation for arbitrary grid resolutions and arbitrary environments. This observation states that there is no upper bound on the length of grid-optimal paths compared to the length of optimal *any-angle* paths between the same query points. The reason for this is that narrow passages in the environment can be blocked whenever the grid resolution is too coarse. This fact, which already holds in the classical path planning situation without weighted regions, gives rise to the question whether we can prove an upper bound on the path lengths when the grid resolution is chosen in such a way that all regions are aligned with the grid.

To tackle this question, we generalized the concept of *homotopic paths* in the context of the WPR by defining *region-homotopic paths*. We showed that aligning all regions with the grid does not guarantee a grid path and an optimal path to be region-homotopic. We believe, however, that this is a rare case in the sense that it only occurs when the weights are chosen carefully and are numerically close to each other. The intuitive impression is that – when all regions are aligned with the grid – there is only a small range in value differences in weights that allows a grid path and an optimal path to be not region-homotopic.

The main contribution of Chapter 3 is a path-cost analysis of grid paths in the context of the WRP when all regions are aligned with the grid. We have shown analytically that a grid path is never more expensive than $2\sqrt{2}$ times the cost of an optimal path.

In Chapter 4, we presented VBP to approximately solve the WRP for real-time simulation and gaming applications. VBP is a hybrid method that aims at combining the advantages of an efficient A^* search [42] on a grid representation with the ϵ-approximation property of the *Steiner-point method* (SPM) by Aleksandrov et al. [2]. We derived three pruning heuristics that use a rough A^* path as a guidance to indicate regions that are likely to be used by an optimal path. This is where our analysis of Chapter 3 comes into play: As we have shown, grid paths and optimal paths in weighted regions do not need to be region-homotopic, even when all regions are aligned with the grid. We showed empirically that in a set of small scenes with a high grid resolution, the number of paths that are not region-homotopic to paths computed with the SPM is small. This means that in most of these cases, our VBP method prunes the search space in such a way that it still contains all the regions that are intersected by an ϵ-optimal path computed with the SPM on the entire search space. For these cases, we can therefore guarantee the same ϵ-optimality as for the original method by Aleksandrov et al. [2]. For the remaining cases, we can still guarantee ϵ-optimality with respect to an optimal path in the pruned search space.

We have shown experimentally that VBP improves on the graph-construction times and the graph-query times when the initially computed grid path intersects only a small portion of the total number of triangles in a scene. In such scenarios, the additional time that is needed to prune the search space is small compared to the time we can save for computing a path in the pruned search space. In addition, we have shown that the time needed for constructing and querying a graph with VBP in a large scene can be smaller than the query times of the SPM alone. Our experiments give a first indication that this property holds when the number of pruned triangles is large, when we use small ϵ-values, and when the computed paths are sufficiently long. When these properties are met, the graph-search phase of the SPM has to explore such a large number of graph nodes that its query times cannot compete with the overall time needed to run VBP. In the context of real-time simulations and gaming applications, typical virtual environments contain even greater numbers of triangles than the large scene that we have tested. We believe that the improvement in running times becomes even greater with even larger scenes, in which more triangles can be pruned, and/or when using even smaller ϵ-values.

The improved performance comes at the price of higher path costs. The average increase that we observed in our experiments was 21.1%. However, in 159 out of all 195 cases that we have tested, we observed a smaller increase in path costs. A typical range was between 1% and 10%, whereas only a few outliers in our data caused the mean value to be higher. By visually inspecting these outliers, we concluded that the computed paths were still reasonably short. This gave rise to the conjecture that the increase in path costs was caused by numerical imprecision due to edge-crawling path segments on the boundary of high-cost regions. We leave further validation of this conjecture for future work.

VBP assumes a fixed set of weights for the regions that are given as an input. This, however, does not impose a hard constraint when we want to simulate various agent profiles in the context of games and simulations. By not storing fixed weight values but rather abstract region types in the environment, it is possible to have varying region preferences for different types of agents. To this end, we need to translate the region-type information to the actual weight values for a given agent profile before running the algorithm. As such, it is possible to model various application scenarios, e.g. to compute paths for pedestrians in a city environment that prefer to stay on a sidewalk, or paths for wild animals that prefer to stay hidden in dense woods. Note, however, that VBP does not take an agent's radius into account when it is represented as a disc. Similar to the original formulation of the WRP, we assume start and goal positions as points in the environment, and the computed paths have no guaranteed clearance from high-cost regions. Taking an agent's radius into account is handled by our novel path-following method as described in Chapter 7. As such, an open question for future work is how we can handle an agent's radius and keep clearance from high-cost regions when computing global paths as indicative routes.

Overall, the work presented in the first part of this thesis leaves room for interesting future investigations on the WRP in the context of grid-path analyses and real-time path-computation methods. One interesting challenge is to derive an abstract property that a WRP instance may or may not have to yield region-homotopic paths when using a grid approach. This seems a promising next step towards understanding the geometric principles behind grid approaches and the WRP in general, which in turn might lead to novel real-time approximation algorithms. Another challenge for future work is to further improve the upper bound on the path costs of grid-optimal paths. Regarding future real-time path computation methods, the VBP method should be seen as the first instance of a concept of hybrid methods. We designed it in such a way that all parts of the method can be changed independently. How we compute a rough guidance path does not depend on how we prune the search space, which again does not depend on how the final ϵ-approximate path in the pruned search space is computed. Research in all three areas can lead to improved variants of VBP. One interesting variant for future work could be to make VBP keep clearance from region boundaries when an agent is represented as a disk. In its current form, VBP does not take varying agent sizes into account and can therefore yield edge-crawling path segments. Traversing such a path as an indicative route when an agent has a spacial extent would make that agent partially intersect regions that potentially have undesired high costs.

In conclusion, we believe that the work presented in Part I of this thesis can help to further understand the underlying geometrical principles of the WRP. Furthermore, we believe that the VBP method can be seen as a first concept for developing future real-time approximation algorithms for the WRP and thus improve path planning in weighted regions in a variety of fields such as simulation, gaming and robotics applications.

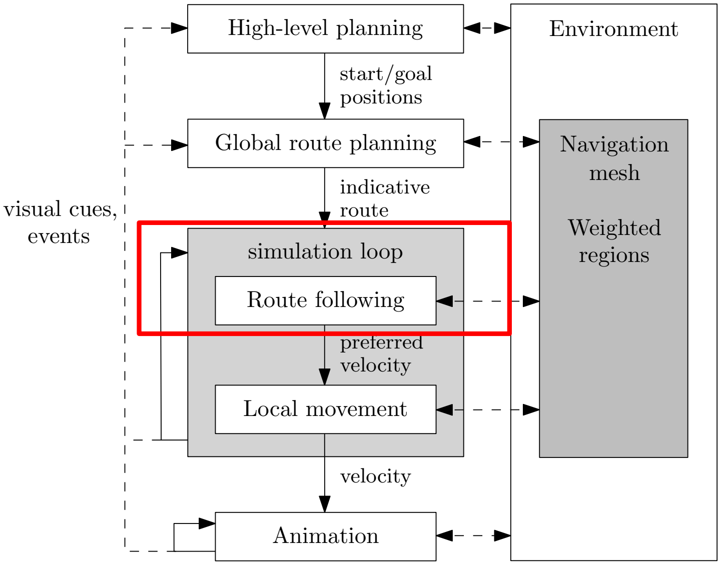
In the upcoming Part II of this thesis, we take this a step further. In the same way as the VBP method computes a first rough guidance path, which it then refines in a subsequent step, we will use the same principle one abstraction level higher: The paths computed with our VBP will again serve as a rough guidance path for autonomous virtual agents that traverse such paths in real time. The final trajectories traversed by such agents will be computed with novel methods that we will present in Part II of this thesis. Such trajectories can be then seen as refinements of the initial rough guidance path in the same way as the paths computed with VBP can be seen as refinements of the initial A^* grid paths to prune the search space.

Part II

Path Following in Weighted Regions

Chapter 6

THE MIRAN METHOD



As we have discussed in Part I of this thesis, the Weighted Region Problem (WRP) [87] is an interesting computational-geometry problem with high relevance in virtual-agent navigation and crowd-simulation applications. With the VBP method as described in Chapter 4, we can compute paths through weighted polygonal subdivisions in real-time that are proven to be ϵ-optimal in a pruned subset of a virtual environment. In this chapter, we will consider such paths as guidance paths or *indicative routes* as defined in the context of the *Indicative Route Method* (IRM) by Karamouzas et al. [63].

The main contribution of this chapter is a new path-following method for virtual agents, which is based on the IRM. This new method is called *Modified Indicative Routes and Navigation* (MIRAN). The goal is to use an indicative route to steer a virtual agent through an environment in real-time when the environment features weighted regions. Each agent has a set of individual region preferences, which makes our method applicable for a great variety of settings. City environments with traversable regions such as sidewalks, roads, and lawns can be modeled in the same way as weather-influenced environments such as swamps, dirt paths, deserts, and frozen lakes. Furthermore, psychological influences like unattractive environments (narrow passages or trashy areas) or attractive areas (artsy neighborhoods or areas with a panoramic view over a valley) can be modeled as well.

We start with discussing related work in Section 6.1. In Section 6.2, we present preliminaries, and in Section 6.3, we present the details of the MIRAN method. We validate and compare MIRAN with the IRM in Section 6.4.

This chapter is based on the following publication:

[52] N. Jaklin, A. Cook IV, and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds (CAVW)*, 24:285–295, 2013.

6.1 | RELATED WORK

To the best of our knowledge, MIRAN is the first path-following method (as we define it in the context of our five-level planning hierarchy; see Section 1.2) that handles region preferences for autonomous-agent navigation. As such, work that is related to our MIRAN method can be subdivided into two categories. The first category contains path-following methods in *homogeneous* environments that do not feature weighted regions. We now briefly discuss a selection of related work from this first category.

Harabor and Botea [41] introduced a path planner called *Hierarchical Annotated A^* (HAA^*)*. It uses a grid to guide a set of uniquely sized agents, where each agent occupies a $c \times c$ square in the grid. The value of c can be any non-negative integer. HAA^* keeps track of the nearest obstacle to each grid cell to guarantee that the computed paths have sufficient clearance from obstacles. Multiple terrain types are handled by associating each agent with a set of terrain types that this agent can traverse. In other words, for a fixed agent the problem is reduced to the classical path planning problem in homogeneous environments. Kang et al. [62] use a different approach to find paths for agents in homogeneous environments. They present an adaptive agent-navigation approach, which collects data and learns new paths from user-controlled agents. Lo et al. [82] show how agents can learn from raw vision input to navigate autonomously in homogeneous environments. They introduce a hierarchical state model and a novel regression algorithm to avoid the 'curse of dimensionality'.

The above-mentioned methods serve as complete agent-navigation models, and they do not necessarily aim at real-time performance. As such, contrary to the MIRAN method we introduce in this chapter, they are not designed in a modular way to be integrated in a broader real-time navigation framework.

This is different for the *Indicative Route Method* (IRM) by Karamouzas et al. [63], which can be seen as a predecessor of our new MIRAN method. Contrary to the above-mentioned methods, the IRM does aim at real-time performance, and it is designed in a modular way and implemented in the *Explicit Corridor Map* crowd-simulation framework [142]. The IRM is not designed to handle weighted regions. Furthermore, the path-following behavior of an agent that uses the IRM is dependent on the local geometry of the scene and the amount of free space that is locally available. Our MIRAN method adopts some general principles of the IRM, but it addresses and overcomes issues that are related to the latter two properties. We give more details on the similarities and differences between the IRM and MIRAN in the remainder of this chapter.

The second category of related work contains methods in heterogeneous environments with various regions, which often come from the robotics community. As such, these methods aim at the navigation of autonomous robots or vehicles. We now briefly discuss a selection of related work from this second category.

Yahja et al. [151] combine framed quadtrees [12] with a D^* search [123] for mobile-robot path planning in environments that are only partially known. The system has been tested in simulations and on an autonomous jeep. Guo et al. [38] present algorithms for mobile-robot path planning and motion control in environments that feature rough terrain. They introduce three modules, one for path searching, one for trajectory generation, and one for trajectory tracking. In each module, performance issues are addressed in the context of robot-safety considerations. Guo et al. [39] present a global path-planning method that is based on common sense and evolution knowledge. The authors show that the method can effectively solve the path-planning problem for robots in complex environments that feature various terrain types. Drews et al. [22] present a path-planning method that takes different terrain into account. The focus is on autonomous non-holonomic vehicles, i.e. agents that cannot instantly move into any arbitrary direction, but are constraint in their movements.

The methods from the second category treat regions mostly as terrain types for real-world applications. Our MIRAN method, by contrast, handles regions in a more general way. Furthermore, we do not aim at handling real-world navigation but virtual-world navigation, which has different requirements and goals. We aim at generating smooth and plausible trajectories rather than paths that are traversable by an actual robot that needs to obey to physical real-world constraints such as gravitation, balance, and stability. In the context of this work, the latter constraints are assumed to be reflected in the weight value for a given region, and they are thus treated from a higher-level perspective. When an application requires more detailed handling of such constraints, they should be addressed in the animation layer of the planning hierarchy. The animation layer operates on the actual $3D$ model of an agent, and methods from this layer are beyond the scope of this thesis.

6.2 | PRELIMINARIES

In this section, we provide preliminary definitions related to our MIRAN method. While the VBP method, which we have described in Chapter 4, tackles the WRP in its original abstract formulation, the MIRAN method aims at computing smooth trajectories that respect an agent's region preferences in the context of real-time virtual world navigation. The problem we tackle can be seen as a variant of the WRP. In this variant, we do not look for a cost-optimal path to a goal destination while taking weighted regions into account. Instead, we compute natural-looking trajectories that are smooth and follow an arbitrary indicative route. Such an indicative route, which could be computed with VBP, functions as a rough estimation of an agent's preferred path. MIRAN adopts the concept from the *Indicative Route Method* (IRM) by Karamouzas et al. [63] of using attraction points to make an agent proceed towards its goal position.

MIRAN is designed in a flexible and modular way that makes it independent of the way in which the traversable space in the environment is represented. The only requirement is that the underlying data structure allows efficient visibility checks between any two points in the environment, as well as obstacle-avoidance behavior. Similar to the situation in the previous chapters, an environment is given as a weighted polygonal subdivision. In the context of this chapter, the weight of a traversable region resembles a *region type*, which could occur in a simulation or gaming application. Examples of region types are terrain representations such as roads, sidewalks, carpeted floors, tile floors, grasslands, snowlands, deserts, and mud pits. A region type can also represent a psychological aspect like a dangerous area or a pleasant spot with a panoramic view. Other examples of abstract data are slope information and crowd-density information [141], which could also be used to weight the attractiveness of a traversable region. The union of these regions make up the free navigable space of the environment.

In addition to weighted regions, we assume obstacle polygons that are not traversable by any character at any time. Such static obstacles could be defined as special region types with an infinite weight. In practice, however, we prefer to treat these as separate object types because such regions should not be considered as part of the search space when looking for paths. We never want to allow an agent to pass through such an obstacle, which could happen if an obstacle was modeled as a region with an infinite weight: Whenever we send an agent to a goal position for which all possible paths intersect an obstacle, we rather want an agent to not move at all instead of walking through static obstacles due to a lack of lower-cost paths.

For our method, we assume that an agent is represented as a single point in the environment. Each agent has a unique set of region preferences that are given as positive numerical values. The less preferred a region is for an agent the higher its numerical value. For example, consider a family that strolls through a park. A child might walk through mud and puddles whereas the adults will avoid those spots;

see Figure 6.1. To model this, the adults have higher values for puddle regions than the child. As another example, a person who is in a hurry might be willing to run through muddy terrain to save time whereas a person in a nice suit is willing to take large detours. Such higher-level considerations can be modeled with MIRAN in the same way as geometrical terrain information by setting an agent's region preferences accordingly.

Following Karamouzas et al. [63], we adopt the concept of an *indicative route* that serves as a rough guidance path for an agent. This means that an agent can locally diverge from an indicative route to walk a smooth path or to avoid collisions with other agents. In theory, an indicative route can be any curve $\pi_{ind} : [0,1] \to \mathbb{R}^2$ through the navigable space of an environment. This curve passes through a sequence of traversable regions. In practice (see Section 6.4), we use indicative routes that consist of straight-line segments, which are either manually created by a user or computed by a path planning method in weighted regions such as our VBP method, which we presented in Chapter 4.

6.3 | METHOD DETAILS

In this section, we show how MIRAN works in detail. Given an indicative route π_{ind}, we assume that the initial position x_0 of an agent equals the starting point s of π_{ind}. Otherwise, we first compute a connection from x_0 to s to bridge the gap. Algorithm 3 gives an overview of the method.

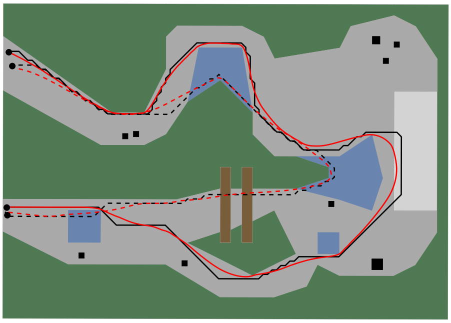


FIGURE 6.1: A path (gray) in a forest (green) with obstacle trees (black), puddles (blue), fallen trees (brown) and a spot with a panoramic view (light gray). Two agents (adult and child) follow automatically computed indicative routes (solid and dashed black). The smoothed paths (solid red for the adult, dashed red for the child) are computed with our MIRAN method. Both the indicative routes and the paths are based on the agents' terrain preferences. The adult avoids puddles and fallen trees, and is attracted to the spot with the panoramic view. The child prefers to walk through puddles, climbs over the trees and is not interested in the panoramic view.

Similar to the Indicative Route Method (IRM) by Karamouzas et al. [63], we define an attraction point in each simulation cycle that is located on π_{ind}. In the IRM, the attraction point is defined as the last point on π_{ind} that intersects the largest-clearance disk around the agent, and it is thus dependent on the local features of the given environment. As a consequence, an agent using the IRM will stick closer to its indicative route when the environment is cluttered and contains many narrow passages, whereas it moves more freely when there are large areas of free space. In particular, this property can make an agent skip arbitrarily large parts of an indicative route; see Section 6.4. Since an indicative route should serve as a rough *indication* of an agent's preferred trajectory, this lack of control in the IRM is clearly undesired.

For MIRAN, by contrast, we define a set of candidate attraction points from which the *best* candidate is picked in each simulation cycle. The best candidate is chosen based on a cost function that incorporates the agent's region preferences. It models the attractiveness of the underlying regions that the agent has to traverse when approaching a candidate attraction point, as well as the distance from an attraction point to the goal position along π_{ind}. We introduce two user-controlled parameters, the *shortcut parameter* σ and the *sampling distance* d, which together with the reference point determine the set $\mathcal{A}_i$ of candidate attraction points. Our method ensures that each candidate attraction point in $\mathcal{A}_i$ is visible from the agent's current position.

In Figure 6.2, we show an example situation and the corresponding attraction points as computed by the IRM (left) and MIRAN (right). Note that we assume only one region type in this example, which makes the agent pick the last visible candidate attraction point along the indicative route.

The rest of this chapter is organized as follows: In Section 6.3.2, we give details on how the set $\mathcal{A}_i$ of candidate attraction points is computed. In Section 6.3.3, we explain how to choose an attraction point from $\mathcal{A}_i$ using our cost-function. In

Algorithm 3 The MIRAN method

Input. Start s, goal g, indicative route π_{ind} from s to g
Output. Smooth region-dependent path from s to g

```
$i \leftarrow 0$
$x_0 \leftarrow s$
while $x_i \neq g$ do
    $r_i \leftarrow$ ComputeReferencePoint($x_i, \pi_{ind}$)
    $\mathcal{A}_i \leftarrow$ ComputeCandidateAttractionPoints($r_i, x_i, \pi_{ind}$)
    $\alpha_i \leftarrow$ PickBestCandidate($\mathcal{A}_i, x_i$)
    $x_{i+1} \leftarrow$ MoveAgentTowardsAttractionPoint($x_i, \alpha_i$)
    $i \leftarrow i + 1$
```

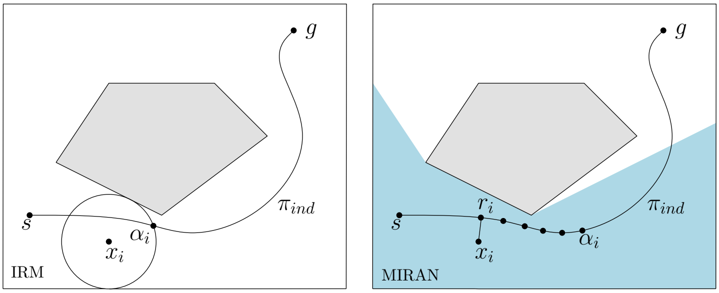


FIGURE 6.2: Comparison of attraction-point computation between the IRM [63] (left) and MIRAN (right). For MIRAN, the visibility polygon for the agent's position x_i is shown in light blue, and all candidate attraction points are inside the visibility polygon. The example shows a scene with only one region type, which makes the MIRAN agent pick the last visible candidate attraction point α_i along the indicative route. The attraction point does not depend on the locally available amount of free space when using MIRAN.

Section 6.3.4, we discuss how we move the agent towards its attraction point in a subsequent step. In Section 6.3.5, we prove the correctness of our MIRAN method, and we conduct experiments in Section 6.4.

6.3.1 | COMPUTING A REFERENCE POINT

We will now discuss how to compute a reference point r_i in each step i of the method. Let x_i be the agent's current position. We define r_i as the *first closest point* from x_i to the part of π_{ind} that lies between the previous reference point r_{i-1} and the previous attraction point α_{i-1} for $i \geq 1$; see Figure 6.3.

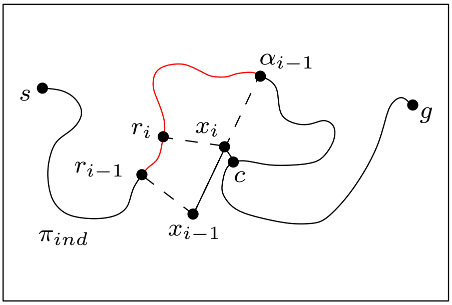


FIGURE 6.3: Only the subpath of π_{ind} between r_{i-1} and α_{i-1} (shown in red) is taken into consideration for the computation of reference point r_i. Choosing the closest point c as the reference point would lead to an undesired shortcut.

For the initial step $i = 0$, we have $r_0 = x_0$ because we assumed x_0 to be the starting point s of π_{ind}. Whenever the agent is located on the indicative route, the reference point r_i equals the current position x_i. We restrict the reference point to the given sub-path of π_{ind} because otherwise we might refer to a point that lies too far ahead along the route, leading to undesired shortcuts. In the example shown in Figure 6.3, picking the closest point c as the next reference point r_i would lead to skipping the whole part of the indicative route between α_{i-1} and c, which can be arbitrarily large in general.

With the above definition of the reference point, we are now able to compute the set $\mathcal{A}_i$ of candidate attraction points. We will show next how this is done in detail.

6.3.2 | Computing the candidate attraction points

As sketched before, we introduce two parameters that can be adjusted by the user to compute the set $\mathcal{A}_i$ of candidate attraction points:

- The *shortcut parameter* σ defines the maximum allowed curve length distance from the reference point to the farthest candidate attraction point.
- The *sampling distance* d defines the maximum curve length distance between each candidate attraction point.

The shortcut parameter σ is used to control the degree of smoothing we want to allow and to prevent the agent from taking undesired shortcuts. It defines the maximum curve length distance the agent is allowed to skip when following the route. Since all candidate attraction points in $\mathcal{A}_i$ are not farther away from r_i than σ (with respect to the curve length of π_{ind}), we ensure that we can pick any of them without generating undesired shortcuts. By σ_i, we denote the point on π_{ind} that has curve length distance σ from r_i. Let $\pi_{ind}(r_i, \sigma_i)$ be the sub-path of π_{ind} from r_i to σ_i. Our candidate attraction points are always located on $\pi_{ind}(r_i, \sigma_i)$.

Now, the first step in the computation of $\mathcal{A}_i$ is to determine all parts of $\pi_{ind}(r_i, \sigma_i)$ that are *visible* from the current position x_i with respect to all static obstacle polygons. In this context, we assume obstacle polygons to obstruct both passability and visibility. Areas such as rivers in a virtual world, which should not be crossed but still allow visibility of what lies beyond, are not treated as static obstacles but rather as weighted regions with a correspondingly high cost. Formally, we compute the *visibility polygon* $\mathcal{P}_i$ for x_i in a polygon that is the union of all traversable space (with the obstacles being holes in that polygon), or – equivalently – in a scene with non-intersecting straight-line segments (the boundary edges of the obstacles). We let the intersection $\mathcal{V}_i := \mathcal{P}_i \bigcap \pi_{ind}(r_i, \sigma_i)$ be the set of all points on $\pi_{ind}(r_i, \sigma_i)$ that are currently visible; see Figure 6.4.

Computing $\mathcal{P}_i$ is a well-studied computational-geometry problem. It can be computed in $O(n \log n)$ time with n being the number of obstacle vertices [36]. Since MIRAN needs to compute this visibility information in each simulation cycle, computing the actual visibility polygon yields a bottleneck for real-time crowd simulation applications with a high number of agents in large environments. In practice, however, it is sufficient to sample the indicative route and perform a simpler visibility check for each sampled candidate attraction point. For further details on how we perform these visibility checks within the *Explicit Corridor Map* framework [56], see Chapter 11.

The set $\mathcal{V}_i$ yields a division of the indicative route into a set of real intervals $V_j = [a_j, b_j] \subset \mathbb{R}$, such that a point $\pi_{ind}(t)$ is visible from x_i for all $t \in V_j$. We let the two end points $\pi_{ind}(a_j)$ and $\pi_{ind}(b_j)$ of each interval be candidate attraction points. The only exception is that we do not want the reference point r_i to be a candidate attraction point. Since r_i equals $\pi_{ind}(a_1)$ whenever the reference point is visible, the agent could possibly be attracted to its current position. While this is not a problem when there is at least one more candidate attraction point that yields lower costs (see Section 6.3.3), in theory it can happen that the reference point is the best choice whenever all other candidates are in high-cost regions. Picking the reference point as the attraction point would make the agent come to a stop without it having reached the goal. Thus, we ignore the reference point r_i and start assigning the candidate attraction points with the point $\pi_{ind}(b_1)$.

We continue to add more values to our set A_i by sampling each visible interval of the indicative route, using the sampling distance d. The closer to 0 the sampling distance the more candidate attraction points we generate and the more is our set $\mathcal{A}_i$ an approximation of a continuous set. A smaller sampling distance therefore generates higher accuracy while increasing computation time. If d is set too large, the resulting inaccuracy may lead to undesired output paths. In practice, however, feasible values of d can be easily set if the size of the environment and the curve length of the indicative route are known (see Section 6.4 for examples). Once set to a feasible value, smaller values affect the overall output paths only insignificantly. In areas near static obstacles with no change of the underlying region, smaller values

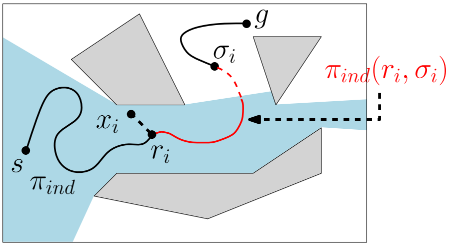


FIGURE 6.4: Visible parts of the environment (light blue) and the indicative route (red) from the current position x_i.

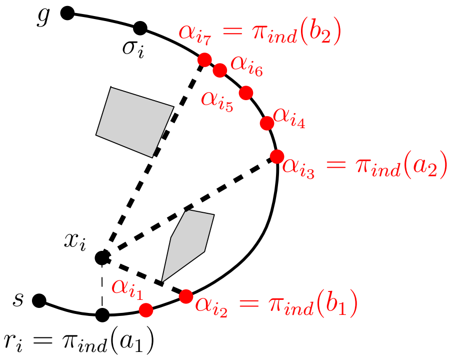


FIGURE 6.5: Example of candidate attraction points computed by our method.

of d do not change the output paths at all because the last visible point along the route will always be picked as an attraction point (see Section 6.3.3 for details).

Sampling each visible interval is done as follows. If for any real interval V_j the curve length distance between $\pi_{ind}(a_j)$ and $\pi_{ind}(b_j)$ is greater than the sampling distance d, we add a candidate attraction point between those two points with curve length distance d from $\pi_{ind}(a_j)$. We iterate this process until the maximum distance between any two subsequent candidate attraction points is d. Note that the curve length distance between b_j and the previous sampled candidate attraction point can be smaller than d. We then let $\mathcal{A}_i = \{\alpha_{i_1}, \alpha_{i_2}, ...\}$ be the final set of candidate attraction points, ordered by their positions along the indicative route. See Figure 6.5 for an example of the resulting set $\mathcal{A}_i$.

6.3.3 | CHOOSING THE ATTRACTION POINT

Now that we have computed the set $\mathcal{A}_i$ of candidate attraction points, we pick the best candidate with respect to a weight function ω as our final attraction point α_i for the current step i of the algorithm.

Let k be the total number of candidate attraction points in the current step i. We consider the straight line segments $l(\alpha_{i_j}, x_i)$ between α_{i_j} and x_i, with $1 \leq j \leq k$. We compute a weight $\omega(l(\alpha_{i_j}, x_i))$ for each such line segment and choose α_i to be the final attraction point for which the corresponding line segment has minimum weight. The Euclidean length of the line segments, the different region types that intersect the line, as well as the agent's preference values influence the weight function. Furthermore, we define the weight function such that candidate attraction points that are farther away from the current position reduce the weight of the line segments. This ensures that the agent will be rewarded for picking an attraction point that is farther away.

For each line segment $l(\alpha_{i_j}, x_i)$, let $\mathcal{R}_{i_j}$ be the set of all region types that $l(\alpha_{i_j}, x_i)$ intersects. Let d_{i_j} be the curve length distance along the indicative route from the reference point r_i to the candidate attraction point α_{i_j}. For each region type $R \in \mathcal{R}_{i_j}$, we let $w(R) > 0$ be the agent's corresponding region preference value. By $l_{i_j}^R$, we denote the amount of region R on $l(\alpha_{i_j}, x_i)$ by summing up the length of all parts of $l(\alpha_{i_j}, x_i)$ that cross region type R.

We define the weight $\omega(l(\alpha_{i_j}, x_i)) := \sum_{R \in \mathcal{R}_{i_j}} w(R) \cdot l_{i_j}^R / d_{i_j}$. The fraction $l_{i_j}^R / d_{i_j}$ describes the relation between the Euclidean length of the line segment with underlying region type R and the curve length distance d_{i_j}. It ensures that we reward picking an attraction point that is farther away along the route.

After computing the weights for each one of the k line segments, we pick the final attraction point $\alpha_i := \alpha_{i,m}$ with $m = \operatorname*{argmin}_{1 \leq j \leq k} \omega(l(\alpha_{i_j}, x_i))$. If there are several candidate points with minimum weight for the corresponding line segments, we pick the one with greatest curve length distance from the reference point along the indicative route.

6.3.4 | MOVING THE AGENT

Once the final attraction point α_i is computed for the current step i of the algorithm, the agent moves towards that point. How this is done depends on the overall framework in which MIRAN is being used.

If MIRAN together with a path planning method such as VBP is used as a stand-alone simulation method, one can simply apply the computed velocity as the agent's new velocity, or or one can use a local force-based steering method as described in the IRM [63] to move the agent towards its attraction point. However, other force-based approaches can be used, as well. Note that an agent might be pushed behind an obstacle due to forces induced by other moving agents. This might lead to an agent not being able to see any parts of its indicative route. In this case, our method terminates and we continue with computing an indicative route from the agent's new position and using it as an input for MIRAN.

We use the *Explicit Corridor Map* (ECM) framework as described in Section 1.2 [31]. This framework provides a planning pipeline, in which MIRAN is embedded as a path-following method. Collision-avoidance with other agents and obstacles with a variable user-controlled amount of clearance are performed as separate subsequent steps. As mentioned before, any other navigation mesh or grid structure that allows efficient visibility checks and obstacle avoidance can be used with MIRAN.

6.3.5 | PROOF OF CORRECTNESS

Now we show that our method ensures that the agent will always find a path to the goal position g – provided the goal can be reached and the agent is not pushed away by other factors (e.g. other moving agents) such that the indicative route becomes invisible.

To prove the correctness of MIRAN, we assume that we have a finite number of polygons that are not infinitesimally small. Note that this does not imply any restrictions for practical applications. In addition, we assume that the agent is moving towards each attraction point directly. While in practice we use Euler Integration as an integration scheme to compute paths that are proven to be C^1-continuous [33], we do not consider this step as part of the MIRAN method itself, but as an optional step in the planning sequence of the *Explicit Corridor Map* framework, in which we have implemented the method. With such an integration scheme, the agent might be pushed behind an obstacle due to a too large step size. In such a case, we can compute a new indicative route from the agent's new position and run MIRAN for that new route. In practice, however, this is unlikely and can be easily avoided by adjusting the step size accordingly.

First, we prove that there is at least one candidate attraction point we can choose from in each step of the algorithm.

Lemma 1. Let i be the current step of the algorithm. It holds that $\mathcal{A}_i \neq \emptyset$.

Proof. We prove this by induction on i. For $i = 0$, we have $x_0 = r_0 = s$. The agent's initial position is the starting point s of the indicative route. Due to the definition of the *visibility intervals* $V_j = [a_j, b_j]$ in Section 6.3.2, it immediately follows that $\pi_{ind}(a_1) = x_0$. Since we assume obstacles as polygons, which by definition do not have curved sides, the agent's position x_0 cannot be the only visible point on the route up to $\pi_{ind}(b_1)$. We conclude that $a_1 \neq b_1$ and also $\pi_{ind}(a_1) \neq \pi_{ind}(b_1)$ (note that the latter does not necessarily follow from $a_1 \neq b_1$ in general, as the route can have self-intersections). By definition of the set $\mathcal{A}_i$, it follows that $\pi_{ind}(b_1) \in \mathcal{A}_0$.

Let $i > 0$. By the induction assumption, we have $\mathcal{A}_{i-1} \neq \emptyset$. Therefore, an attraction point α_{i-1} has been chosen in step $i - 1$ and the agent moved from position x_{i-1} to position x_i, with x_{i-1}, x_i and α_{i-1} being collinear. It immediately follows that the point α_{i-1} is still visible in step i. So there must be an index j such that $\alpha_{i-1} \in V_j = [a_j, b_j]$. By definition of $\mathcal{A}_i$, the point $\pi_{ind}(b_j)$ is always a valid candidate attraction point. □

Next, we show that the sequence of reference points moves forward along the indicative route.

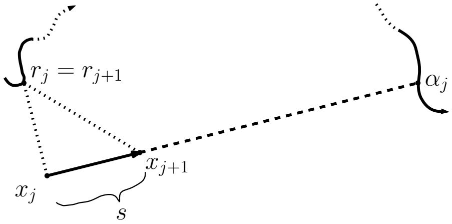


FIGURE 6.6: The distance between x_j and α_j differs from the distance between x_{j+1} and α_j by the amount of the strength c_s of the steering force.

Lemma 2. Let i be the current step of the algorithm. It holds that there is a future step $j > i$ in which the reference point r_j is ahead of r_i along the indicative route.

Proof. Assume by way of contradiction that the opposite holds. Since we define the reference point to be a point between the last reference point and the last attraction point, the opposite assumption would mean that there is a reference point that does not change for all future steps. So we assume that there is a step i in which the reference point r_i equals r_j for all $j > i$. To simplify the notation, we skip the index and denote the fixed reference point as r. It immediately follows that the agent does not reach the goal position g. Otherwise there would be a step $j > i$ in which x_j is closer to g than to r, thus making g the new reference point in step $j + 1$.

Because the agent does not reach g but moves forward in each step due to Lemma 1, it follows that we have an infinite sequence of agent positions $x_i, x_{i+1}, x_{i+2}, \ldots$. The Euclidean distance between each x_j and x_{j+1} in this sequence always equals the move distance of an agent for one simulation step, which is determined by its preferred speed s (see Figure 6.6 and Section 6.3.4). When the agent moves from x_j to x_{j+1}, there is a corresponding attraction point α_j co-linear with x_j and x_{j+1} that has been picked as the best point among all candidate attraction points in $\mathcal{A}_j$. This means that the weight $\omega(l(\alpha_j, x_j)) = \sum_{R\in\mathcal{R}_j} w(R) \cdot {l_j}^R/d_j$ is minimal among all candidate line segments in step j.

Now, in the following step $j + 1$, the next candidate attraction point is computed. The former attraction point α_j has the same curve length distance $d_j = d_{j+1}$ from the reference point r along the indicative route as in step j because r stays the same point due to our assumption. So the weight $\omega(l(\alpha_j, x_{j+1}))$ differs from $\omega(l(\alpha_j, x_j))$ only in the Euclidean distance between the corresponding points and the regions that are intersected by the line segment. Since the Euclidean distance is smaller (it has been reduced by $s > 0$; see Figure 6.6), the weight for α_j in step $j + 1$ is smaller than the weight for α_j in step j. It follows that the attraction point picked in step $j + 1$ must have a smaller weight than the one picked in step j. Therefore, following the sequence $x_i, x_{i+1}, x_{i+2}, \ldots$ of agent positions, the weight for picking the corresponding attraction points becomes smaller in each step by an absolute amount.

However, there is a lower bound for the weight. Let $d_{min} := \min_{i \leq j} ||x_j - r||$ be the minimal distance between the fixed reference point r and all agent positions $x_i, x_{i+1}, x_{i+2}, \ldots$. Then it holds that $||l(\alpha_j, x_j)|| \geq d_{min}$. Otherwise the corresponding attraction point α_j would be closer to x_j than r, thus becoming the new reference point in step $j + 1$. This contradicts our assumption that r stays the same point for all future steps. Furthermore, the curve length distance d_j between the reference point and the attraction point is never greater than the shortcut parameter σ, i.e. $d_j \leq \sigma$. If we let $w_{min} := \min_{R \in \mathcal{R}} w(R)$ be the agent's minimum preference value for all region types, the following lower bound for the weight $\omega(l(\alpha_j, x_j))$ applies:

$$\begin{aligned}\omega(l(\alpha_j, x_j)) &= \frac{1}{d_j} \sum_{R \in \mathcal{R}_j} w(T) \cdot {l_j}^R \\ &\geq \frac{w_{min}}{\sigma} \sum_{R \in \mathcal{R}_j} {l_j}^R \\ &= \frac{w_{min}}{\sigma} \cdot ||l(\alpha_j, x_j)|| \geq \frac{w_{min}}{\sigma} \cdot d_{min}.\end{aligned}$$

Now we know that the weights become smaller in each step, we have an infinite number of such steps, and there is a lower bound for the weight. It follows that the weights must asymptotically approximate the lower bound. This corresponds, however, to an infinite number of asymptotically small portions of region polygons that the agent crosses. Since we assume that there are a finite number of polygons that are not infinitesimally small, we get a contradiction to our assumption, which proves the lemma. ☐

Lemmata 1 and 2 ensure that our method makes the agent move forward in each step of the algorithm. Now we prove that the agent will always reach the goal position g.

Theorem 2. There is an index $i \in \mathbb{N}$ such that $x_i = g$.

Proof. By Lemma 2, it holds that the curve length distance along π_{ind} from the reference point to g becomes smaller over time. Assume by way of contradiction that the agent does not reach its goal, i.e. for all steps i the agent's position x_i does not equal g. Since the reference point gets closer to g over time, it follows that the sequence $(r_i)_{i \in \mathbb{N}}$ of reference points has a limit $l \in \pi_{ind}$. This limit l cannot be reached. Otherwise, if there was a step j such that $x_j = l$, by Lemma 1 there would be at least one candidate attraction point we could choose from, thus making the agent go beyond l, a contradiction.

Since l is a limit point on the indicative route and the agent moves forward in each step, it follows that there is a step i in the algorithm where the curve length distance along π_{ind} between r_i and l is smaller than the sampling distance d. By Lemma 1, we know that there is at least one candidate attraction point α_i to choose from. By the definition of the set $\mathcal{A}_i$ and because of the sampling distance d, this point α_i either lies beyond l or equals the reference point r_i. If the latter case holds and $r_i = \alpha_i$, the agent is attracted to its reference point until it reaches that point or a different attraction point beyond l will be picked. In any case, an attraction point α_j beyond l will finally be picked. The method makes the agent move towards α_j and beyond l, yielding a contradiction. ☐

6.4 | EXPERIMENTS

We have validated MIRAN and compared it against the Indicative Route Method (IRM) [63], in a set of experiments. The goals of these experiments are threefold: First, we show that – similarly to the IRM – MIRAN creates paths that are smooth and visually convincing for simulations and gaming applications, with the novel addition of handling an agent's region preferences while following an indicative route. This validation is done via visual inspection of a set of typical paths that were computed using MIRAN and the corresponding paths that were computed using the IRM. Second, we show that MIRAN allows control over how closely an indicative route is being followed and what the impact of the two user parameters is in this context. This is again done via visual inspection of a set of paths with varying parameter settings. Third, we show that MIRAN is real-time applicable and that the average computation time that is needed for one simulation step is small, even when we use parameter settings that induce a large number of candidate attraction points per simulation step. All the experiments were performed on a PC running Windows 7, with a 3.2 GHz AMD Phenom$^{\text{TM}}$ II $X2$ CPU and 4 GB memory. We used one CPU core for the computations.

In Section 6.4.1, we present the scenes that we have used for the experiments. In Section 6.4.2, we show example paths as computed by MIRAN and by the IRM in scenes with weighted regions. In Section 6.4.3, we show the impact of the two user parameters on the path-following behavior of MIRAN. In Section 6.4.4, we measure the time that is needed to compute one step of the simulation using MIRAN for different parameter settings.

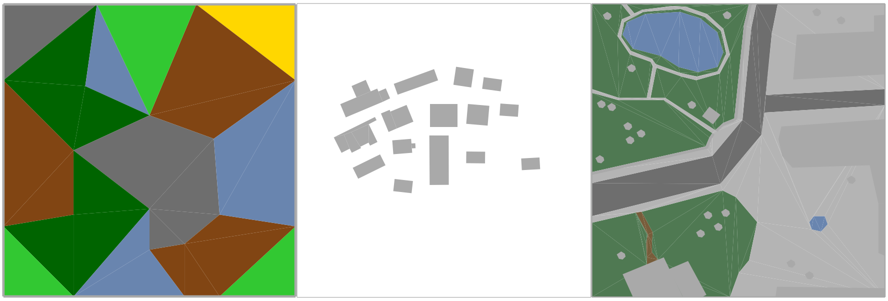


FIGURE 6.7: The three scenes we have used in our experiments: *abstractRegions* (left), *military* (center), and *suburb* (right).

6.4.1 | THE TESTED SCENES

We tested the method on three scenes with varying types of indicative routes and parameter settings. The first scene we used is called *abstractRegions*, and it spans an area of 50×50 units; see Figure 6.7 (left). It contains no obstacle polygons (except the four line segments that form its bounding box), but a set of weighted regions with arbitrary weights. We used the same weights as in Chapter 4, where we have used this scene to create a 16×16 tiled large environment: gray 4, darkgreen 30, lightgreen 3, blue 30, yellow 6, and brown 2.

The second scene we used is called *military*, and it represents a $2D$ footprint of the McKenna MOUT training site at Fort Benning, Georgia, USA; see Figure 6.7 (center). This scene spans an area of 200×200 units. It contains no weighted regions – or in other words, one region type (white) with a default weight of 1 – and a set of 23 convex obstacle polygons that represent a total of 15 buildings.

The third scene is called *suburb*, and it contains both obstacle polygons (shown in gray) and weighted regions; see Figure 6.7 (right). It spans an area of 100×100 units. We used the following weights for this scene: light gray (sidewalk) 1, dark gray (road) 4, green (garden/lawn) 30, blue (water) 40, and brown (path to the house) 1.

6.4.2 | MIRAN PATHS VS. IRM PATHS

The main contribution of MIRAN is that it handles an agent's individual region preferences while computing paths that are as smooth and visually plausible as paths computed with the IRM [63]. To illustrate this, we have computed a set of example paths with MIRAN and with the IRM for the same query points. For all path computations, we used a shortcut parameter σ of 5 and a sampling distance d of 1.

We have computed paths in the *suburb* and *abstractRegions* scenes with varying indicative routes. Figures 6.8 and 6.9 show a total of 24 typical example paths. In each of the two scenes, we computed twelve paths, of which six were computed using MIRAN and six were computed using the IRM. Each set of six paths consists of three paths based on indicative routes that were computed using A^* on a grid, and three paths based on manually drawn indicative routes that respect clearance from region boundaries.

The following conclusions can be drawn from this visual comparison: MIRAN indeed computes smooth paths that are visually plausible with respect to the underlying region types. The IRM, by contrast, is not designed to account for region preferences and treats all region types as non-weighted traversable space. This does not mean that MIRAN and IRM paths always differ to a great extent. In some cases, the paths are still close to each other; see Figure 6.8 (top-left) for an example. The fact that both paths are comparably similar in this example is a result of the obstacle placements in this environment. The IRM computes an attraction point on the indicative route based on the largest-clearance disk around the agent's position, retracted to the medial axis of the environment. The placement of obstacles (walls and trees) in the *suburb* scene lets the agent pick attraction points that happen to be close to the attraction points as computed by MIRAN. The same attraction-point computation principle of the IRM causes the two paths to differ to a great extent in other examples; see Figure 6.8 (top-right). Here, the amount of free space around the agent's retracted position when the agent approaches the lake region makes it follow a straight line towards the goal.

6.4.3 | THE IMPACT OF σ AND d ON PATH-FOLLOWING BEHAVIOR

Another contribution of MIRAN is that it allows the user to control how closely an indicative route should be followed, whereas the IRM depends on the environment geometry in this regard. In this second experiment, we tested the impact of the shortcut parameter σ and the sampling distance d on an agent's path-following behavior.

We used the *military* scene with an indicative route that was computed using the underlying *Explicit Corridor Map* navigation mesh: The indicative route was set to be the shortest route to the goal that keeps clearance from obstacles with respect to the agent's radius. Figure 6.10 shows the resulting paths as computed with MIRAN for a fixed shortcut parameter σ of 5 and sampling distances d varying between 0.1 to 4.0.

By visually inspecting the computed paths, we can confirm the following expected property: In scenes with no weighted regions, the sampling distance has no impact on path-following behavior, as long as at least one candidate attraction point is visible in each simulation step, i.e. when d is not set to a larger value than σ.

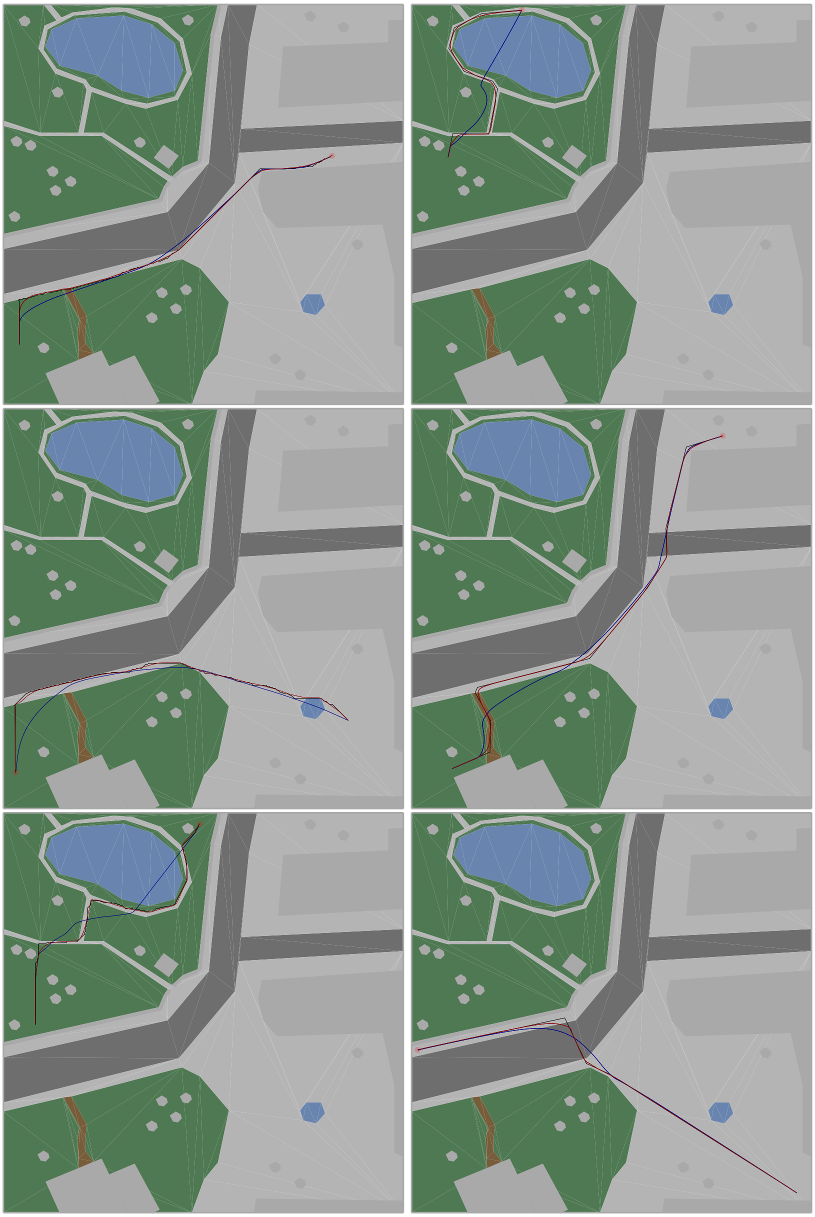


FIGURE 6.8: Visual comparison of MIRAN with the IRM in the *suburb* scene. *Left column:* Paths based on indicative routes that were computed using A^* on a grid. *Right column:* Paths based on manually drawn indicative routes that respect clearance from region boundaries. All indicative routes are shown in black, all MIRAN paths are shown in red, and all IRM paths are shown in blue.

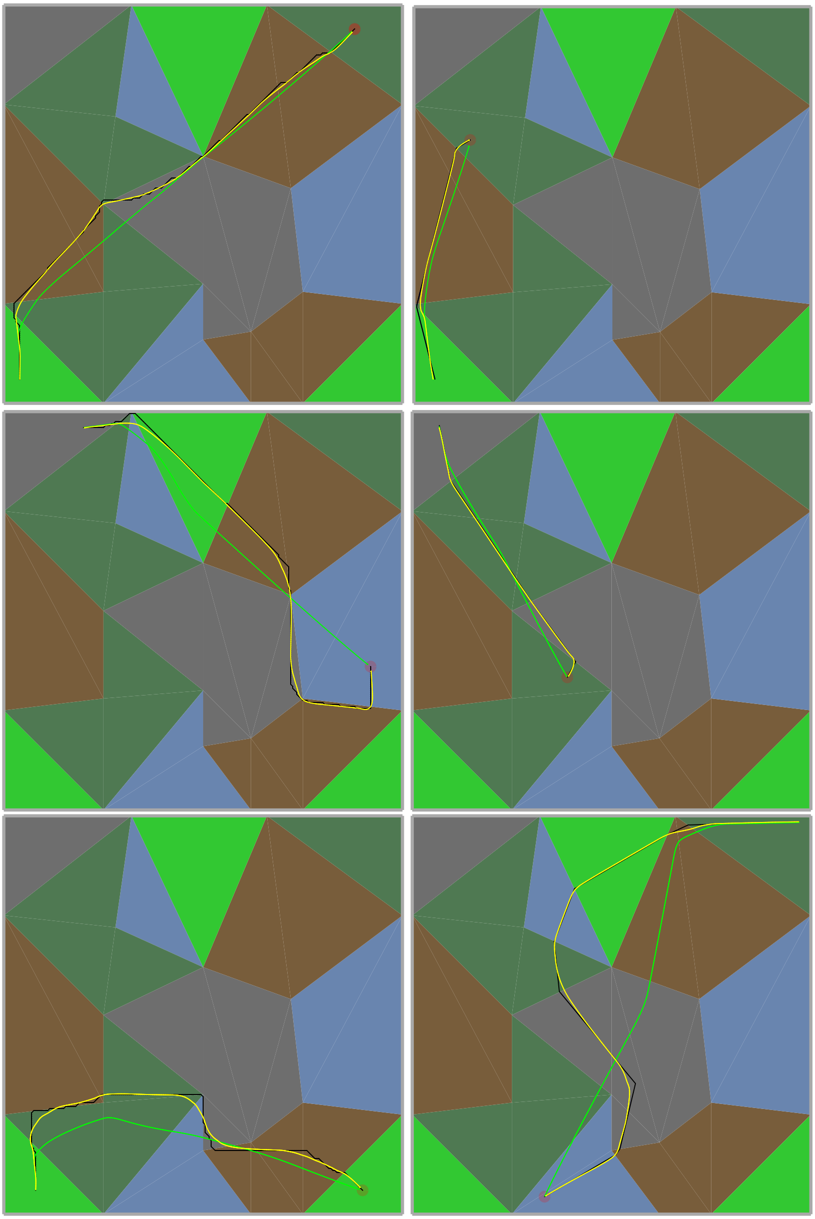


FIGURE 6.9: Visual comparison of MIRAN with the IRM in the *abstractRegions* scene. *Left column:* Paths based on indicative routes that were computed using A^* on a grid. *Right column:* Paths based on manually drawn indicative routes that respect clearance from region boundaries. All indicative routes are shown in black, all MIRAN paths are shown in yellow, and all IRM paths are shown in green.

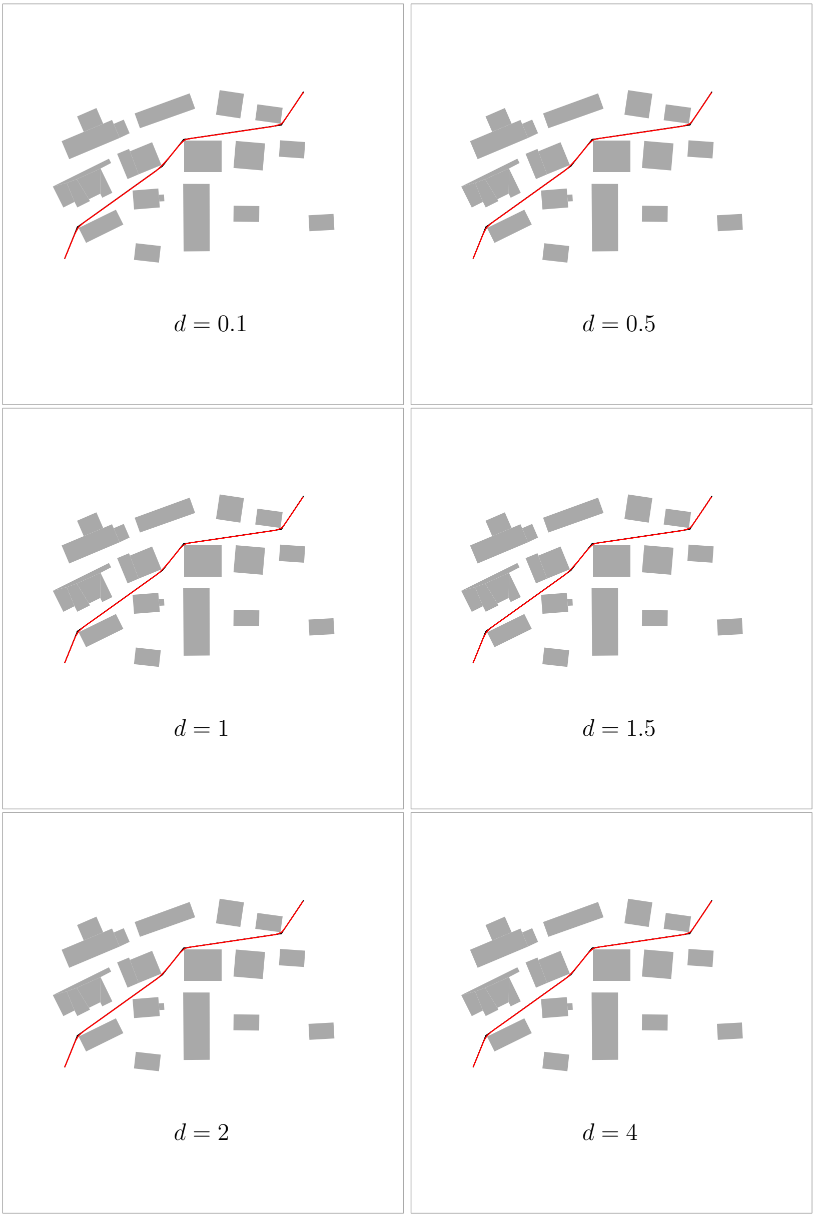
The reason for this is that all candidate attraction points make the agent cross the same (default) region type, and thus, only their location on the indicative route determines their costs. Since we reward the agent for picking a point that is further ahead, the agent will always pick the furthest visible point. This point is either directly determined by σ or by obstacle corners that occlude parts of the indicative route, but never by the sampling distance d. Since we defined the endpoints of the visibility intervals along the indicative route as candidate attraction points that are independent of d, we can always set d equal to σ when using MIRAN in scenes with no weighted regions.

Contrary to scenes with non-weighted regions, we expect d to have an impact in scenes with weighted regions. The reason is that a varying number of candidate attraction points induces a varying number of regions that an agent could possibly cross. In other words, a particular sampling distance could enable a traversal that would not be available when we used a larger sampling distance. To test whether our expectation holds in practice, we have computed a set of paths in the *suburb* scene with the same parameter settings as in the previous experiment: σ was set to 5, and d varies between 0.1 and 4.0. Figure 6.11 shows the resulting paths.

While the overall paths look similar, we can indeed spot subtle differences in the path-following behavior that are caused by changing the sampling distance. The differences are mainly noticeable around bending points of the indicative route, while the paths are similar when the agent follows a rather long straight-line segment of the indicative route. For instance, near the end of the indicative route, where the agent moves from the sidewalk region to the road region, the agent stays on the sidewalk for a longer period of time when d is small. The reason is that smaller values for d induce larger numbers of candidate attraction points, and the agent has the option to leave the lower-cost region at a later point in time than with larger values for d. Still, the overall impact of d in path-following behavior is comparably small, and we can conclude that d can be seen as parameter that is related to performance rather than path quality.

To test the impact of σ on path-following behavior, we used the same scenes and indicative routes as in the first example, but this time, we used a fixed sampling distance of 1.0 and shortcut parameters σ varying between 5 and 30. Figures 6.12 and 6.13 show the resulting paths. From these results, we can conclude the following:

In general, when using large values for σ, an agent could take shortcuts and skip large portions of an indicative route (also see Figure 6.14 (bottom)). However, this does not happen in the examples we show in Figure 6.12. Due to the geometry of the scene and the fact that we use a shortest path with clearance as an indicative route, the only bending points of the indicative route are near obstacle corners. As such, different values for σ result in the same paths because the agent is not able to take any shortcuts without running into obstacles. This is different in the *suburb* scene. Here, the indicative route has bending points that are not induced by obstacles, but by the underlying weighted regions in the environment. With larger



$d = 0.1$

$d = 0.5$

$d = 1$

$d = 1.5$

$d = 2$

$d = 4$

FIGURE 6.10: Six example paths in the *military* scene for a fixed shortcut parameter $\sigma = 5$ and varying sampling distances d. The indicative route is shown in black, and the resulting MIRAN paths are shown in red.

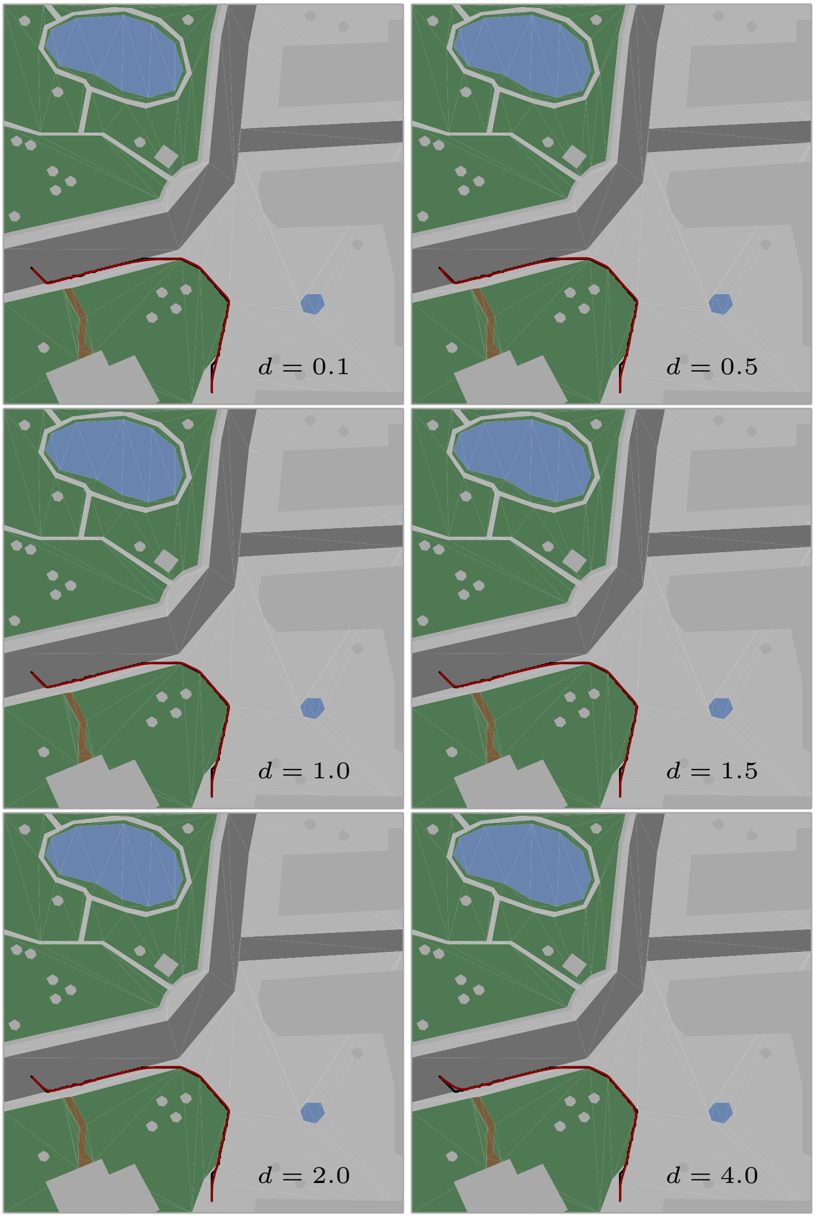


FIGURE 6.11: Six example paths in the *suburb* scene for a fixed shortcut parameter $\sigma = 5$ and varying sampling distances d. The indicative route is shown in black, and the resulting MIRAN paths are shown in dark red.

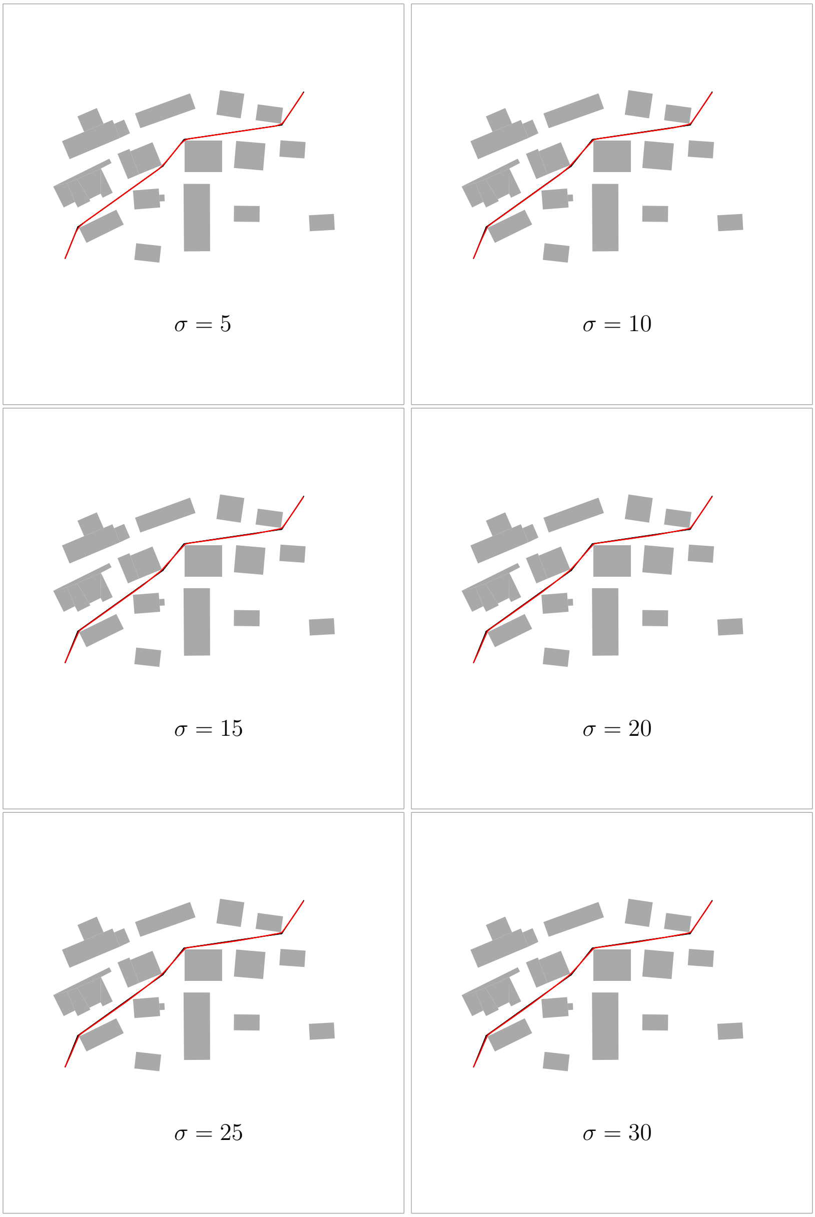
values of σ, the agent is therefore permitted to take shortcuts. As an example, this is noticeable with the lawn/garden region, which the agent starts to cross sooner with larger values for σ. The reason is that the reward factor for picking an attraction point that is further ahead dominates the higher costs for crossing the lawn, and this happens in correspondence with the value that is set for σ. Overall, this experiment shows that σ can have a great impact on the path-following behavior in the way that we expected it. However, this is not necessarily the case when the geometry in a scene does not allow any shortcuts. The latter happens when we use a shortest path with clearance as an indicative route, for which all bending points are induced by obstacles and not by traversable weighted regions.

Two particular problems with the MIRAN parameters arise when we either set the shortcut parameter to a very large value, or when we use indicative routes that contain self-intersections. Figure 6.14 shows two examples that illustrate these problems. In the top example, we used a manually drawn indicative route that circumnavigates the lake region in the *suburb* scene. We used a comparably large shortcut parameter of 250. As can be seen, the agent starts crossing the lake because it is attracted to the goal position too soon. The reason for this is that we reward an agent for picking a candidate attraction point that is further ahead along the indicative route. Since we use a very large shortcut parameter, the last visible candidate attraction point along the indicative route yields such a large reward in the MIRAN weight function that the position of the candidate attraction point alone dominates the punishment for crossing the high-cost lake region. For future work, it would thus be interesting to decouple the range of region-weight values from the curve-length distances of the indicative routes.

Figure 6.14 (bottom) shows an example for the second problem. We use the *military* scene and an indicative route that was manually drawn and contains self-intersections. While self-intersections are not a problem per se for the MIRAN method, they can yield undesired behavior when we use a somewhat unlucky shortcut parameter setting: When the shortcut parameter happens to be large enough such that it covers the curve-length distance of the loop that is induced by the self-intersection, the agent might be attracted to a point that makes it move against the desired walking direction that is induced by the indicative route. This behavior can be observed for the path shown in blue with a shortcut parameter of 300. For future work, we could use the intended walking direction as an additional factor to decide whether a candidate attraction point is feasible or not.

6.4.4 | PERFORMANCE

In this final experiment, we measured the performance of MIRAN. The goal was to validate that MIRAN still runs at interactive rates, even when the number of candidate attraction points per simulation step is large.



$\sigma = 5$

$\sigma = 10$

$\sigma = 15$

$\sigma = 20$

$\sigma = 25$

$\sigma = 30$

FIGURE 6.12: Six example paths in the *military* scene for a fixed sampling distance $d = 1.0$ and varying shortcut parameters σ. The indicative route is shown in black, and the resulting MIRAN paths are shown in red.

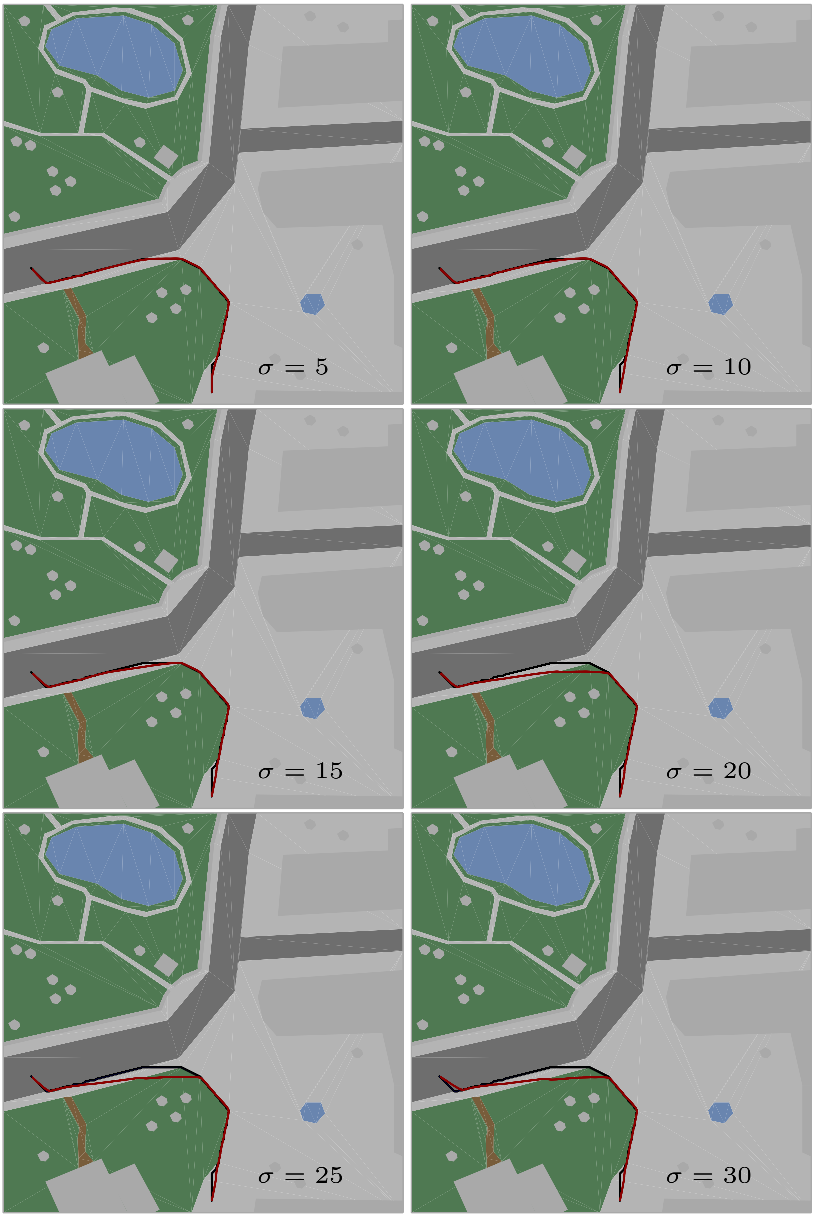


FIGURE 6.13: Six example paths in the *suburb* scene for a fixed sampling distance $d = 1.0$ and varying shortcut parameters σ. The indicative route is shown in black, and the resulting MIRAN paths are shown in dark red.

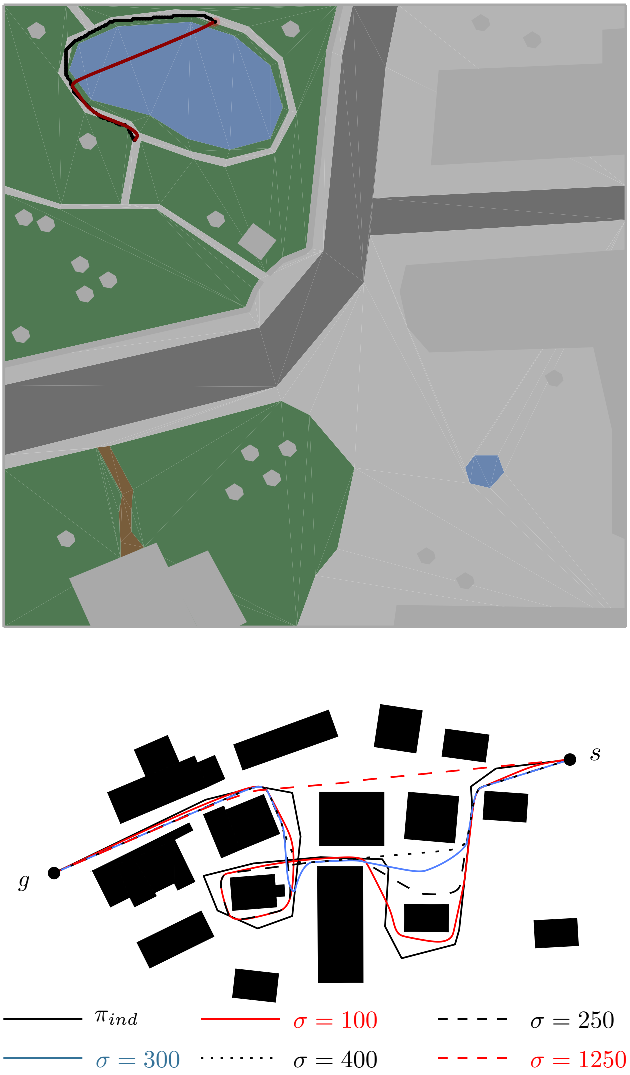


FIGURE 6.14: Problems with large values for σ. *Top:* An agent crosses large parts of the high-cost lake region due to the reward factor in the weight function. *Bottom:*: An 'unlucky' value for σ (here: 300) can make an agent move against the walking direction that is induced by the indicative route. The sampling distance d used for the shown paths is 20.

abstractRegions16x16	$d = 0.2$	$d = 0.4$	$d = 0.6$	$d = 0.8$	$d = 1.0$
$\sigma = 5$	0.28[0.08]	0.21[0.06]	0.20[0.09]	0.17[0.05]	0.16[0.05]
$\sigma = 10$	0.51[0.16]	0.31[0.10]	0.27[0.08]	0.22[0.06]	0.20[0.06]
$\sigma = 15$	0.77[0.23]	0.45[0.12]	0.36[0.10]	0.29[0.07]	0.25[0.07]
$\sigma = 20$	1.08[0.31]	0.61[0.18]	0.45[0.11]	0.36[0.09]	0.31[0.08]
$\sigma = 25$	1.44[0.38]	0.77[0.20]	0.58[0.17]	0.46[0.12]	0.38[0.09]
$\sigma = 30$	1.89[0.68]	0.99[0.27]	0.71[0.32]	0.56[0.15]	0.46[0.12]
suburb	$d = 0.2$	$d = 0.4$	$d = 0.6$	$d = 0.8$	$d = 1.0$
$\sigma = 5$	0.25[0.08]	0.18[0.06]	0.16[0.06]	0.14[0.05]	0.13[0.05]
$\sigma = 10$	0.46[0.19]	0.27[0.10]	0.22[0.08]	0.19[0.07]	0.17[0.06]
$\sigma = 15$	0.70[0.29]	0.41[0.16]	0.30[0.12]	0.25[0.09]	0.22[0.08]
$\sigma = 20$	1.00[0.44]	0.55[0.24]	0.43[0.19]	0.34[0.15]	0.28[0.11]
$\sigma = 25$	1.33[0.62]	0.72[0.33]	0.51[0.22]	0.43[0.19]	0.35[0.16]
$\sigma = 30$	1.66[0.77]	0.91[0.41]	0.60[0.27]	0.51[0.25]	0.42[0.18]

Table 6.1: Average time (ms) needed to compute one simulation step when using MIRAN in the *abstractRegions16x16* and *suburb* scenes. The total number of simulation steps was around 460 (*abstractRegions16x16*) and 950 (*suburb*). The standard deviations are shown in square brackets.

We measured the average time that was needed to compute one step of the simulation. To this end, we used MIRAN in two scenes with indicative routes that are sufficiently long to induce a large number of simulation steps – and thus, a sufficiently large number of samples – before the agent reaches the goal position. We used the *abstractRegions16x16* scene, which features no obstacles (and thus, the visibility checks are expected to be fast), and the *suburb* scene, which features both weighted regions and obstacles. The total number of simulation steps was around 460 and 950 in the *abstractRegions16x16* and *suburb* scenes, respectively.

Table 6.1 shows the computation times per simulation step, and Figure 6.15 visualizes the results for the *abstractRegions16x16* scene and *suburb* scenes. From these results, we can conclude that MIRAN does indeed run at interactive rates, with computation times per simulation step below 2 ms, even for parameter settings that induce a large number of candidate attraction points. These observations hold for both scenes with and without obstacle polygons, which induce additional visibility checks and a wider spread of the measured computation times from their mean value.

6.5 | LIMITATIONS

The MIRAN method as described in this chapter improves on the IRM in several aspects. Still, the method has its limitations.

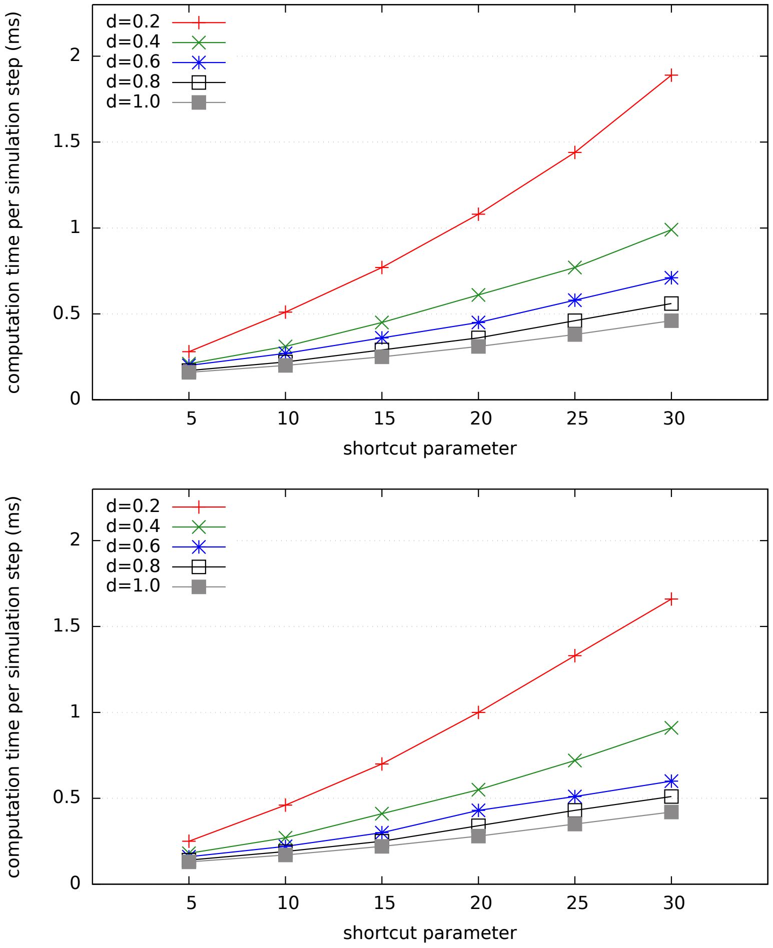


FIGURE 6.15: Average time (ms) needed to compute one simulation step when using MIRAN in the *abstractRegions16x16* (top) and *suburb* (bottom) scenes.

First, it is defined for a virtual agent that is represented as a point. Since an agent is represented as a disc or similar shape that has an actual width in practical applications, keeping clearance from static obstacles needs to be handled separately when using MIRAN in the form as described in this chapter. How this is done depends on the underlying data structure for representing traversable space. In the *Explicit Corridor Map* framework, for instance, we can perform visibility checks that account for an agent's radius to compute feasible candidate attraction points; see Chapter 11.

Second, MIRAN incorporates region preferences to generate smooth, region-based trajectories, but when handling collisions as a subsequent step in the navigation-planning hierarchy, regions are not taken into account. If, for instance, two agents that are using MIRAN approach each other and make an avoidance maneuver, the agents might still traverse high-cost terrain or otherwise undesired region types during that maneuver. As future work, this could be handled by using a method such as Moussaïd et al. [88] that is based on candidate directions for avoiding collisions with other agents. These could be weighted with the underlying regions that need to be crossed when picking a particular direction, such that an agent will always choose a lower-cost region for its avoidance maneuver when all other invovled factors (such as deviation from the goal direction) are equally feasible.

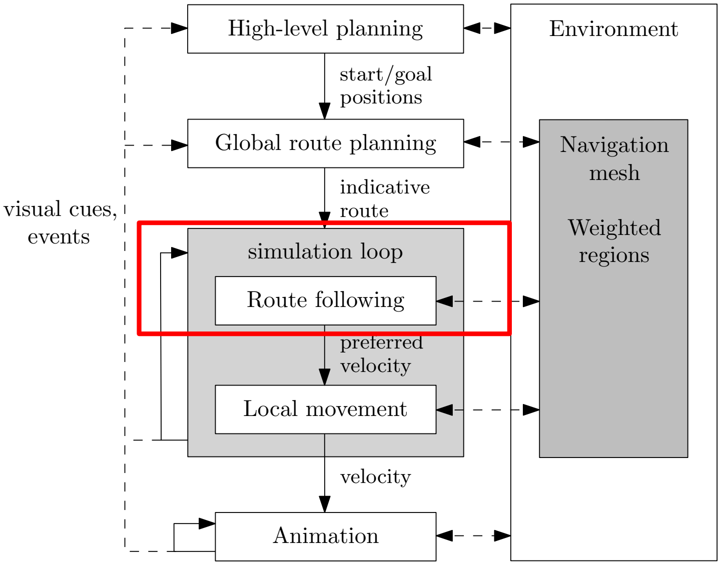
Third, MIRAN is designed in a way such that the actual curve length of an indicative route influences the costs for particular candidate attraction points. While this is not a problem per se, it imposes certain constraints on the value ranges of region preferences that should be used to model a scenario. If, for instance, MIRAN is used in a very large environment with an agent following an extremely long indicative route and a large shortcut parameter σ, picking small values for region preferences will have almost no effect on the agent's trajectory because the curve-length parameter in the weight function overrules the particular region preference values. As such, users of MIRAN need to set region preference values in accordance with the overall size of the environment to generate feasible results. For future work, it would therefore be interesting to decouple the range of feasible weight values from the size of the environment and the lengths of indicative routes. In this context, another promising next step would be to handle dynamically changing weights and direction-dependent weights. The latter could, for instance, be used to model slopes and steepness information, where a region resembles a hill or mountain that has a large weight when an agent tries to ascend it, whereas the weight is low when the agent tries to descent it.

Lastly, MIRAN is a sampling-based method that allows the user to control the sampling accuracy via the sampling distance d. It would be interesting to modify the computation of attraction points by not sampling the indicative route, but using the route as a continuous set and computing the points that minimize the underlying weight function accordingly. One approach could be to use a radial sweep with its center point being the agent's position and having the region-triangle vertices as event points. This is, however, an open problem because we do not impose any constraints on the environment. This means that the given polygonal subdivision can have arbitrary many different shapes and triangle vertices, and the indicative route can be any curve in the plane, which makes it difficult to categorize and sort all possible event points in an ordered fashion.

This concludes the chapter on our MIRAN method. In the next chapter, we will discuss extensions to the MIRAN method. In particular, we will discuss how to modify MIRAN such that it handles agents that are represented as a disc. To this end, we introduce a new weight function, and we explain how to modify the computation of candidate attraction points accordingly.

CHAPTER 7

MIRAN FOR DISC-BASED AGENTS



While the MIRAN method as described in the previous chapter enables the simulation of region-based path following at interactive rates, it has its limitations because an agent is assumed to be represented as a point. In the context of virtual agents in simulations or gaming applications, this might cause undesired behavior. In such applications, an agent's representation for path planning and collision avoidance has a spacial extension, e.g. the shape of a disc. In this context, we define visibility in a way such that the entire disc needs to be able to move in a collision-free straight-line fashion towards a candidate attraction point. As such, a candidate attraction point that is visible for a point-based agent might not be visible for a disc-based agent. Furthermore, a disc-based agent should, if possible, not partially intersect a high-cost region. This might happen when we use our point-based MIRAN method for a disc-based agent; see Figure 7.1.

In this chapter, we discuss modifications to the MIRAN method for agents that are represented as a disc. Algorithm 4 shows the pseudo-code of the modified MIRAN

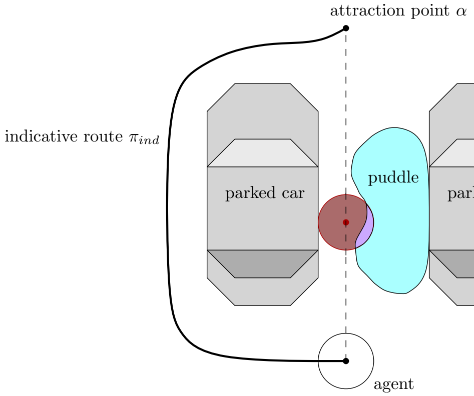


FIGURE 7.1: Example situation with two parked cars, a puddle between them, and a given indicative route π_{ind} that circumnavigates the cars. MIRAN for a point-based agent could make the agent pick attraction point α and thus traverse the dashed line. This yields an undesired intersection with the puddle due to the spacial extent of the agent.

method. The differences to the point-based variant of the method are the computation of candidate attraction points and the way in which a best candidate is determined in each simulation cycle. Both of these new sub-methods take the agent's radius R into account.

The remainder of this chapter is organized as follows: In Section 7.1, we describe how to modify the computation of candidate attraction points accordingly. In Section 7.2, we introduce a new weight function to determine the costs of a *capsule* shape over arbitrary regions, induced by a sliding disc. This new weight function replaces the weight function in MIRAN for a line segment from an agent's position to a candidate attraction point as described in the previous chapter. Finally, in Section 7.3, we show some example paths that were computed with the described modifications. We also discuss future work such as how to use the new weight function in collision-avoidance methods to account for region preferences during a collision-avoidance maneuver.

7.1 | CANDIDATE ATTRACTION POINTS FOR DISC-BASED AGENTS

The computation of candidate attraction points for disc-based agents follows the same general principle that we have used for point-based agents in the previous chapter: Given an indicative route π_{ind}, we first determine the endpoints of the sub-routes of π_{ind} that are *visible* from the agent's current position x_i. We refer to these sub-routes as *visibility intervals*. In this context, we say that two points

Algorithm 4 The MIRAN method for discs

Input. Start s, goal g, indicative route π_{ind} from s to g, **agent radius** R
Output. Smooth region-dependent path from s to g

```
$i \leftarrow 0$
$x_0 \leftarrow s$
while $x_i \neq g$ do
    $r_i \leftarrow$ ComputeReferencePoint($x_i, \pi_{ind}$)
    $\mathcal{A}_i \leftarrow$ ComputeCandidateAttractionPointsForDisc($r_i, x_i, \pi_{ind}$, $R$)
    $\alpha_i \leftarrow$ PickBestCandidateForDisc($\mathcal{A}_i, x_i$, $R$)
    $x_{i+1} \leftarrow$ MoveAgentTowardsAttractionPoint($x_i, \alpha_i$)
    $i \leftarrow i + 1$
```

are mutually *visible* when the straight-line segment between the points, which is inflated by the radius of the disc that represents the agent, does not intersect any static obstacles. We then add the endpoints of the visibility intervals as candidate attraction points. Subsequently, we sample the visibility intervals and add the sampled points as additional candidate attraction points. Figure 7.2 shows an example of a disc-based agent and its candidate attraction points on π_{ind}.

The sampling step does not require any changes from the point-based version of MIRAN to account for disc-based agents. If we ensure that the endpoints of the visibility intervals are computed correctly, it trivially follows that any sampled point within such a visibility interval is visible for a disc-based agent. As such, the computation of the endpoints is the only step that requires some changes to account for discs.

In theory, it is sufficient to compute the visibility endpoints for a disc-based agent by computing the visible parts of the scene for a point-based agent amidst obstacles that are inflated by the radius that represents the agent; see Figure 7.3. The inflation of obstacle polygons can be performed by computing the Minkowski sum of each polygon with the disc that represents the agent [77], and by handling possible intersections between the inflated polygons [145]. This can be done in $O(n \log^2 n)$ time

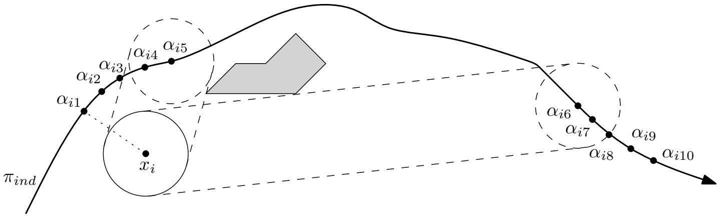


Figure 7.2: Example of candidate attraction points α_{ij} for a disc-based agent at position x_i with indicative route π_{ind} and reference point r.

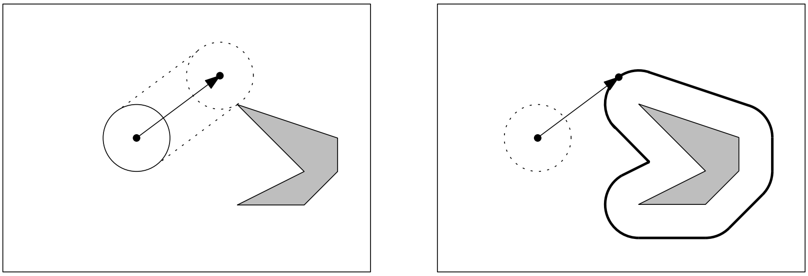


FIGURE 7.3: Left: Visibility of a candidate attraction point for a disc-based agent amidst polygonal obstacles. Right: The same scenario with obstacles that are inflated by the radius of the disc that represents the agent, which allows simple straight-line visibility checks.

using a divide-and-conquer approach [67] or in $O(n \log n)$ using an randomized incremental approach [20]. Similar to the inflated obstacle polygons, the visible parts in the inflated scene consist of straight-line segments and circular arcs. Similarly to the generalized visibility graph [74] as discussed in Chapter 2, we can define the circular arcs as *generalized vertices*, and *visibility* between such an arc a and a point p as the question whether there is a straight-line from p that is tangent to a. We can then compute the visible parts in the inflated scene with m generalized vertices in $O(m \log m)$ time by using the same methods as for the visibility polygon for a point amidst non-inflated obstacles. Note that in general, m is different from n due to the fact that inflated obstacles might overlap and induce new (generalized) vertices. However, it is known [67] that when the obstacles are disjoint, then $m = O(n)$ and therefore $O(m \log m) = O(n \log n)$.

In practice, similarly to the point-based version of MIRAN as described in the previous chapter, we use a simpler approach to compute the visibility endpoints: We start with sampling the route and immediately check the visibility of each sampled point. Here, we assume that the sampling is dense enough such that any two consecutive sample points do not skip an entire obstacle. As soon as we detect the change from a visible point to an invisible point, we perform a binary search on the route between those two points to determine the actual visibility-interval endpoint. This is an efficient approach in practice because a visibility check can be performed efficiently within the *Explicit Corridor Map* framework [56]. For further details on how we perform such visibility checks within our framework, see Chapter 11.

The above-mentioned modifications are sufficient to compute candidate attraction points for disc-based agents. Figure 7.4 shows an example that was computed within our framework based on the adjustments as described in this section. What remains to discuss is a modified weight function for enabling disc-based agents to use MIRAN.

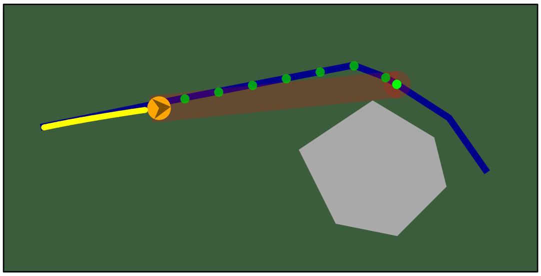


FIGURE 7.4: Example of candidate attraction points (shown in green) for a disc-based agent, computed within the *Explicit Corridor Map* framework [56]. The indicative route is shown in blue, and the final trajectory of the agent is shown in yellow.

7.2 | WEIGHT FUNCTION FOR DISC-BASED AGENTS

With the modified method to compute candidate attraction points as described in the previous section, we can now guarantee collision-free movement towards such a point for a disc-based agent when using MIRAN. What is left to define is a new function that assigns weights to such a sliding-disc traversal. The new weight function should account for the regions that a disc-based agent needs to cross when moving from its current position to a candidate attraction point. The part of the traversable space that is intersected by such a move has the shape of a *capsule*. Alternatively, in the context of generalizing MIRAN from point-based agents to disc-based agents, such a capsule can also be seen as an inflated line segment. Figure 7.5 shows an example of a capsule induced by a disc-based agent moving from its current position x_i to an attraction point α_i. We now describe the general concept behind the modified weight function for a capsule.

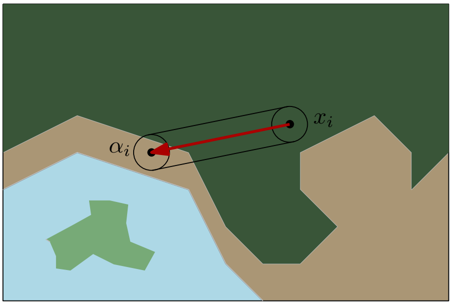


FIGURE 7.5: Example of a traversal amidst weighted regions for a disc-based agent from its current position x_i to an attraction point α_i, inducing a *capsule* shape.

7.2.1 | GENERAL CONCEPT

One option to define the costs for such a capsule movement is to compute the area of each region that is intersected by the capsule. We can then sum up the corresponding weights according to the area ratios against the total area of the capsule. This, however, might cause an agent to slightly intersect a high-cost region when other intersected regions have sufficiently low costs to make the overall movement a feasible option. In the context of virtual-agent navigation in simulation and gaming applications, this is not desired behavior. Particular high-cost regions that represent undesired or even dangerous areas should not be modeled as a hard obstacle because an autonomous agent should still be able traverse them, but it should avoid it whenever possible. Examples are water puddles (Figure 7.1) or lava pits in a game. In such an example, an agent should never voluntarily touch the lava, unless external forces push it.

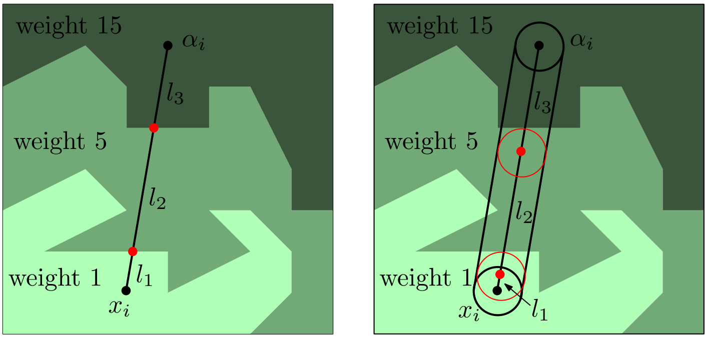
What we want instead is a high-cost region such as lava to overrule all lower-cost regions when determining the overall costs of a capsule-shaped movement. As such, we use the following general principle to compute the weights for a capsule: Starting from an agent's current position x_i and considering the movement as a sliding disc over time, we compute the points in time when the sliding disc starts and ends intersecting a particular region. Whenever the disc starts intersecting a particular region, we check whether the newly intersected region should dominate the overall costs, i.e. whether the costs for the newly intersected region is highest among all region costs that are currently being intersected by the disc. Similarly, whenever the disc ceases to intersect a particular region, we check whether that region was the dominating one and update the costs accordingly. Overall, what we compute with this heuristic is a set of *interval points* on the straight-line segment between the agent's position x_i and its (candidate) attraction point α_i. Any two consecutive interval points on that line segment determine an interval in which a particular region dominates all other currently intersected regions. Similarly to the point-based version of MIRAN as described in the previous chapter, we define the overall weight for a sliding-disc traversal as the weighted Euclidean length of these intervals (and a reward factor for picking points that are further along the indicative route, which we omit here for ease of explanation). Figure 7.6 shows an example of a traversal, the corresponding interval points, and the overall weight when using our new disc-based version of MIRAN compared against the point-based version of MIRAN. In the next section, we describe how to compute the interval points.

7.2.2 | COMPUTATION OF INTERVAL POINTS

We now discuss how to compute the interval points on the straight-line segment between an agent's position x_i and a (candidate) attraction point α_i. Throughout the rest of this chapter, we denote the straight-line segment by l.

The first step for computing the interval points is to determine all regions that are intersected by the entire capsule shape. For this step, we assume the region polygons to be triangulated and determine a corresponding list of intersected region triangles. How this is done in detail depends on the underlying navigation mesh that is being used. A fast and easy way is to use a grid for efficient point-location queries to determine the triangle that contains the agent's position x_i. Using a structure such as a doubly-connected edge list (DCEL) [90] allows traversing adjacent triangles to check for intersection with the capsule. The capsule consists of three parts, for each of which we check for intersections: the disc centered in x_i, the disc centered in α_i, and the rectangular *box* that forms the connection between the two discs; see Figure 7.7.

$$\omega_i = 1 * ||l_1|| + 5 * ||l_2|| + 15 * ||l_3||$$



point-based MIRAN: $\omega_i = 43.5$ disc-based MIRAN: $\omega_i = 75.9$

FIGURE 7.6: Computation of *interval points* (shown in red) on the straight-line segment between x_i and α_i for point-based MIRAN (left) and disc-based MIRAN (right).

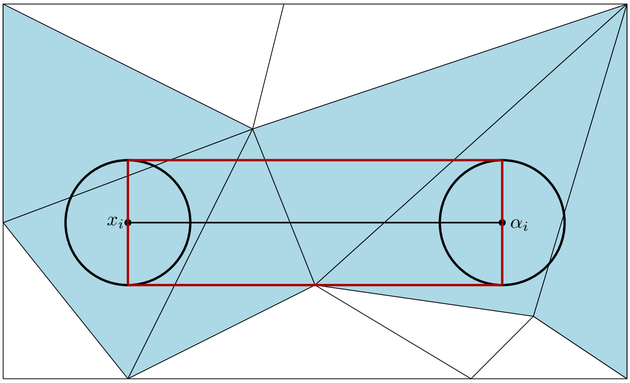


FIGURE 7.7: Example of region triangles that are intersected by a capsule (shown in light blue). The capsule itself consists of the disc centered in x_i, the disc centered in α_i, and a *box* (shown in red) that connects the two discs.

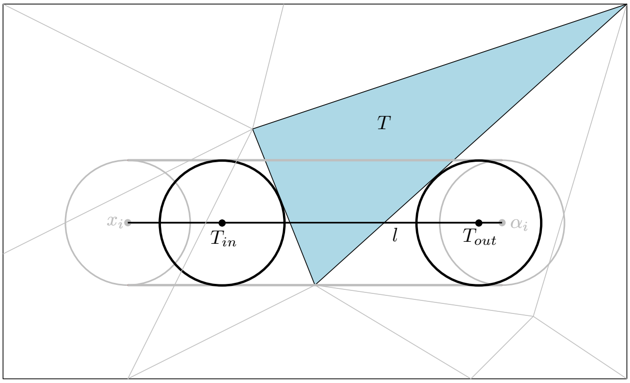


FIGURE 7.8: An intersected triangle T and the corresponding in-point T_{in} and out-point T_{out} on the straight-line segment l.

Once the list of intersected triangles has been computed, we compute for each triangle T in that list an *in-point* T_{in} and an *out-point* T_{out} on l. By orienting l from x_i to α_i and considering the agent's disc sliding along l, we define the in-point T_{in} as the point on l at which the agent's disc centered in T_{in} starts intersecting triangle t. Similarly, we define the out-point T_{out} as the point on l at which the agent's disc centered in T_{out} ceases intersecting triangle t; see Figure 7.8. Note that these points can also lie on the extension of l beyond x_i or α_i. To detect this, we parameterize l as $l(t) = (1 - t)\, x_i + t\, \alpha_i$ for $t \in [0, 1]$. In-points or out-points beyond l then yield $t < 0$ or $t > 1$. Whenever this happens, we let $T_{in} = x_i$ or $T_{out} = \alpha_i$, respectively. The reason is that the disc then either already intersects triangle T at the start of the traversal, so the intersection interval on l starts in x_i, or the disc still intersects triangle T at the end of the traversal, so the intersection interval on l ends in α_i.

For ease of explanation, we define a coordinate system with x_i being the origin, and we let l determine the positive direction of the x-axis of that coordinate system. We can do the computations for each of the three edges per intersected triangle and pick the minimum and maximum values in x-dimension of the computed points to determine the in-point and the out-point for each triangle. Alternatively, to save some of the computations, we can compute a list of intersected triangle edges in the initial step instead of intersected triangles, using a DCEL representation of the environment.

Having computed the interval points of all intersected triangles, we can now go through these points in increasing order of x-coordinates on l and check for each found in-point whether the corresponding triangle has a weight that dominates the previously found dominating weight value. We then update the currently dominating weight accordingly. Similarly, we update the currently dominating weight whenever we find an out-point on l. While doing this, we discard all interval points that do not correspond to the dominating weight and only store the ones that are needed for the final computation of the overall weight of the traversal from x_i to α_i. This can be done by going through the sequence of interval points once. By l_i, $1 \leq i \leq k$, we denote the Euclidean length of the $i-th$ interval of the sequence of k final interval points on l. By $R(l_i)$, we denote the corresponding region costs for l_i, and by d_i we denote the curve-length distance on the indicative route from the agent's reference point to the candidate attraction point α_i. The overall weight ω_i of the traversal is then computed similar to the point-based version of MIRAN as described in the previous chapter:

$$\omega(l(x_i, \alpha_i)) = \sum_{1 \leq i \leq k} R(l_i) \cdot l_i / d_i \tag{7.1}$$

This concludes the description of the disc-based modifications of our MIRAN method. In the following section, we discuss results and possible future extensions.

7.3 | Results and future extensions

With the modified computation of candidate attraction points and the modified weight function as described in the previous sections, we are now able to use MIRAN for disc-based agents without having to deal with collisions between an agent and static obstacles due to an agent's spacial extension. Furthermore, an agent is now able to follow a given indicative route based on its region preferences that account for its whole spacial extension rather than its center point alone. In other words, an agent now tries to avoid intersections with high-cost regions, even when such intersections occur only partially and do not affect an agent's entire disc.

Figure 7.9 shows example paths that were computed with our modified MIRAN method within the *Explicit Corridor Map* (ECM) framework [56]. All paths were computed by simulating the agents' traversal in real-time on a PC running Windows 7 with a 3.20 Ghz AMD Phenom$^{\text{TM}}$II $X2$ $B57$ Processor, 4 GB Ram, and an NVIDIA GeForce GTX 650 graphics card.

In addition, we have created the car-puddle example as shown in theory in the beginning of this chapter (Figure 7.1), and compared the point-based version of MIRAN to the disc-based modification of MIRAN. We have used the same indicative route for both methods, and the same overall parameter settings: The sampling distance d was set to 0.2, and the shortcut parameter σ was set to 50. The overall scene is 20 x 20 meters, and it features one puddle-region with weight 30. The cars are obstacles and thus not part of the traversable space, and the remaining region is a default terrain with weight 1. The scene and the resulting paths can be seen in Figure 7.10. Note that for both types of paths, we used the adjustments for computing the endpoints of the visibility intervals that we described in this chapter. The reason for this is that agents in the ECM framework are represented as discs,

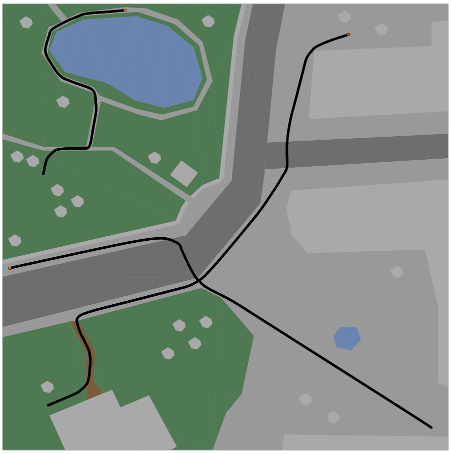


Figure 7.9: Example paths computed with our disc-based modification of MIRAN within the *Explicit Corridor Map* framework [56]. Goal positions at the end of each path are shown in red.

and thus, the point-based computation of the visibility intervals as described in Chapter 6 would yield undesired collisions with obstacles corners. As such, the differences between the paths are due to the differences in the weight functions only.

These preliminary experiments show that our modifications for disc-based agents indeed generate the desired differences in an agent's navigational behavior. In the car-puddle scene, the agent avoids the puddle and circumnavigates the cars when using the disc-based weight function. Furthermore, the extensions to MIRAN that we describe in this chapter do not affect the real-time applicability of our method, and the paths are still smooth and visually convincing for autonomous-agent navigation in simulation or gaming applications.

For more future extensions, it would be interesting to try the modified weight function in a collision-avoidance method that uses sampled candidate directions. The collision-avoidance method by Moussaïd et al. [88] is an example of such a method. Adjusting the computation of candidate directions to avoid collisions with other agents might enable region-based collision-avoidance. This feature is currently missing in the work presented in this thesis. While agents are now able to traverse a virtual environment based on region preferences, agents might still temporarily intersect high-cost regions when avoiding collisions with other agents.

This concludes our chapter on the disc-based extensions of MIRAN. In the following chapter, we will recapitulate our region-based path-following contributions and give an outlook on open problems and future work in this area.

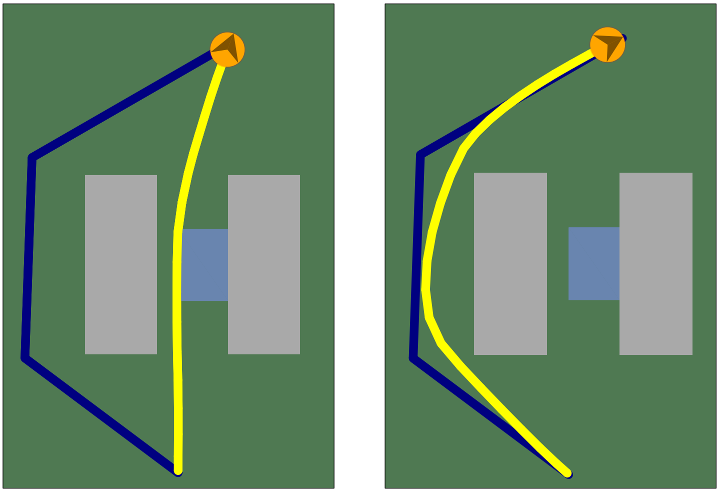
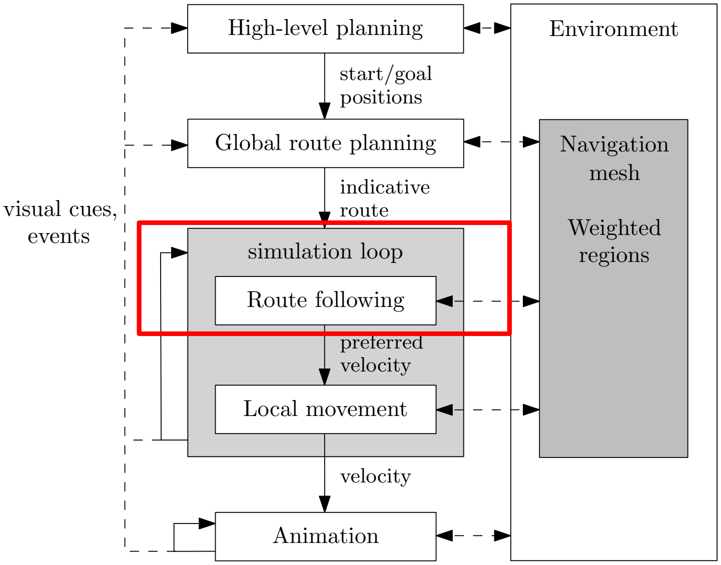


FIGURE 7.10: Comparison of point-based MIRAN (left) with disc-based MIRAN (right) in the car-puddle example within the *Explicit Corridor Map* framework [56]. The indicative route is shown in blue, and the agents' final trajectories are shown in yellow.

Chapter 8

Conclusion Part II



This chapter concludes the second part of this thesis: path following in weighted regions. We have taken the discussions on path planning in weighted regions in Part I to the next level of our five-level planning hierarchy (see Chapter 1). Here, a global path, computed within the route-planning level of the hierarchy, is taken as an input to make an agent traverse it while considering an agent's individual region preferences.

In Chapter 6, we have introduced a novel path-following method named *Modified Indicative Routes and Navigation* (MIRAN) for point-based agents. MIRAN can be seen as the successor of the *Indicative Route Method* (IRM) by Karamouzas et al. [63]. It adopts the concept of computing an attraction point on a given global path (or *indicative route*) in each step of the simulation and making an agent approach its attraction point based on steering forces. The novel contributions of MIRAN over the IRM are twofold: First, MIRAN takes an agent's region preferences into account. In each simulation step, an agent picks its attraction point from a set of candidate points, based on the region types it would have to cross, plus the location of a candidate point on the indicative route. Second, MIRAN allows to control the path-following behavior of an agent. To be precise, MIRAN introduces a *shortcut parameter* that determines how closely an agent should follow its indicative route, or – in other words – how much of its indicative route an agent is allowed to skip while following it. In the IRM, there is no control over this type of behavior because the location of an attraction point is dependent on the local geometry of the environment around an agent.

In Chapter 7, we have discussed extensions to the MIRAN method that enable the simulation of agents that are represented as disks. We have discussed how to adjust the computation of candidate attraction points accordingly. Furthermore, we have introduced a new weight function that replaces the weight function for point-based MIRAN. The new weight function takes the entire spacial extent of a disk-based agent into account. It computes the costs for a particular candidate attraction point α based on a *capsule* shape that is induced by the movement of the agent from its current position towards α. The region types that are intersected by the capsule determine the overall costs. Here, high-cost regions dominate lower-cost regions in the sense that they override the overall costs whenever the disk intersects them, even when the intersection is only a small portion of the disk. This heuristic yields navigational behavior that is desired for typical simulations and gaming applications, in which particular region types (e.g. lava pits in a game) should never be purposely crossed by an agent, while still allowing an agent to be pushed into them, so that treating such regions as hard obstacles is not a feasible solution.

The path-following methods that we have discussed in Part II assume an indicative route as an input. We theoretically defined such an indicative route as an arbitrary curve in the plane, which is usually a connected sequence of straight-line segments in practice. In the context of this thesis, decoupling the computation of an indicative route from subsequent path-following behavior is one of the key aspects of the five-level planning hierarchy that we have discussed in Chapter 1. However, using a fixed indicative route in such a hierarchy still leaves room for improvements. There are open problems that arise from the fact that we use a fixed indicative route that cannot change during the simulation.

In many applications, an indicative route is a shortest path with clearance from obstacles for disk-based agents. This causes problems when all agents in a simulation use such an indicative route. The areas around obstacle corners usually get congested quickly, which might lead to unrealistic deadlock situations. In real life, one would expect a crowd to make use of all traversable space around such a congested corner. Even though our MIRAN method allows an agent to skip particular parts of an indicative route, all attraction points are still located on the corresponding indicative route. This makes an agent stick closely to such a route and does not allow them to make use of the free space around a congested corner. The main reason is that – both with the IRM and with MIRAN – candidate attraction points that are further ahead are not visible until an agent succeeds in navigating around an obstacle corner. A possible approach for future work could be to generalize the concept of an indicative route to an *indicative surface*, so that candidate attraction points can be located in a two-dimensional area instead of a one-dimensional path. Then a path-following method such as MIRAN could be extended to not only account for region preferences but also for the current local crowd density around a candidate attraction point when picking the currently best candidate.

Another problem is that a fixed indicative route, together with a force-based steering approach, can always lead to an agent being pushed behind an obstacle such that the indicative route is not visible any more. Again, an indicative surface could overcome this issue. Alternatively, allowing local and dynamic changes of parts of an indicative route, together with re-planning strategies that do not re-plan an entire indicative route but only relevant parts of it, could be another interesting approach for future work.

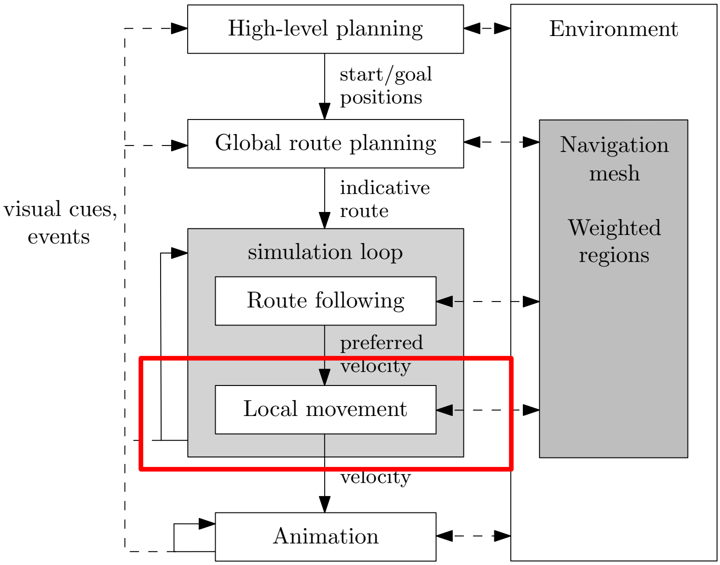
In the upcoming Part III of this thesis, we take our discussions from single-agent behavior to crowd behavior. We will discuss how we can improve coordination among agents when crowd density is high. In this context, we also tackle the aforementioned problem of unrealistic deadlock situations due to a lack of coordination. Furthermore, we will discuss social-group behavior from a navigational point of view by introducing a novel method that generates emergent social formations and group coherence.

Part III

Crowd Simulation

CHAPTER 9

THE STREAM MODEL: COORDINATING DENSE CROWDS



The novel algorithms – VBP and MIRAN – that we have presented in Parts I and II of this thesis, respectively, aim at steering virtual agents through an environment with weighted regions. They can be combined with collision-avoidance methods to form a crowd-simulation framework for large numbers of agents, in which each agent uses VBP and MIRAN independent of other agents. In other words, both VBP and MIRAN are designed as single-agent methods. Such methods usually do not consider the coordination of multiple agents and large virtual crowds. This lack of coordination can lead to unrealistic crowd behavior such as deadlock situations when crowd density is high.

In this chapter, we present a new crowd-coordination model named *Stream*, which tackles some of these problems. Stream combines the advantages of agent-based and flow-based paradigms while only relying on local information. It is designed in a modular way, and it can therefore be used as an additional step in the *local-movement* level of the simulation pipeline (see Chapter 1). In particular, it can

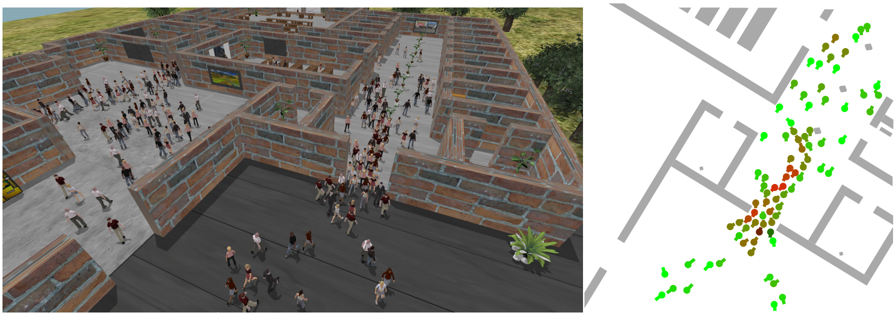


FIGURE 9.1: *Left*: Example scene of our Stream model: A dense crowd of agents collaboratively moves through a narrow doorway. *Right*: A $2D$ representation of the doorway shows that each agent interpolates between individual behavior (green) and coordinated behavior (red).

be combined with any global path-planning and local collision-avoidance method within such a planning hierarchy. Stream handles the coordination of virtual crowds at arbitrary densities. When using Stream as an additional step within the local-movement level of the planning hierarchy, we show that it can significantly reduce the occurrence of deadlock situations in extremely dense crowds. It also enables the simulation of varying agent profiles that determine an agent's willingness to coordinate with a crowd, based on a small set of factors. Figure 9.1 shows an example scene in which Stream is being used.

In Section 9.1, we discuss work that is related to our Stream model. We give preliminary definitions in Section 9.2, and we present the details of our Stream model in Section 9.3. Lastly, we conduct experiments and validate our model in Section 9.4.

This chapter is based on the following publications:

[136] A. van Goethem, N. Jaklin, A. Cook IV, and R. Geraerts. On streams and incentives: A synthesis of individual and collective crowd motion. In *28th International Conference on Computer Animation and Social Agents (CASA 2015)*, pages 29–32, 2015.

[137] A. van Goethem, N. Jaklin, A. Cook IV, and R. Geraerts. On streams and incentives: A synthesis of individual and collective crowd motion. Technical Report UU-CS-2015-005, Utrecht University, 2015.

9.1 | RELATED WORK

For a general overview of crowd simulation topics, we refer the reader to the books by Thalmann and Musse [128] and Pelechano et al. [107]. In the remainder of this section, we focus on selected work that is related to our Stream model.

One of the first flow-based models was proposed by Hughes [51]. Hughes represented pedestrians as a continuous density field, and crowd dynamics were described using partial differential functions. Treuille et al. [130] proposed a continuum-based crowd simulation model. They used a dynamic potential field to simulate large crowds in real-time. This model yields emergent phenomena such as lane formation. Individual autonomous agents can be added to the crowd as dynamic obstacles. Lee et al. [78] presented a regression-based model. The model is able to simulate particular crowd behavior that has been learned from recorded video data of real crowds.

Other flow-based approaches come from the robotics community. Kerr and Spears [69] use a simulation model based on gas-kinetics for mobile robots. Pimenta et al. [109] propose a method for swarms of mobile robots that is based on Smoothed Particle Hydrodynamics.

All of these flow-based models are able to solve high-density scenarios, but they are not well-suited for low- to medium-density scenarios where the individuality of single agents has a large impact on the overall behavior of the crowd. Furthermore, these flow-based methods usually have high computational costs when many different goal states are involved.

In addition to flow-based models, a wide range of agent-based crowd simulation models is available. Helbing et al. introduced a social-force model for pedestrian dynamics in [45] and subsequent work [43, 44, 46]. Torrens [129] has proposed a crowd simulation framework that aims at handling higher-level trip planning computations, medium-level computations such as vision and steering, and low-level computations for locomotion and physical collision detection. Similar to our model, the framework accounts for following behavior and agents aligning their direction of movement. Contrary to Torrens' work, our model lets agents automatically interpolate between following and aligning behavior for agents that have a desire to coordinate with the crowd. This desire is based on a unique set of factors including the locally perceived crowd density. The HiDAC system by Pelechano et al. [106] combines psychological and geometrical rules with a social- and physical-forces model. Shao and Terzopoulos [118] show how to integrate motor, perceptual, behavioral and cognitive components within one model to simulate pedestrians in an urban environment.

Lemercier et al. [79] have conducted an experimental study on herding and following behaviors. Models based on real-world pedestrian movements have been proposed by Antonini et al. [4] and Paris et al. [102]. Vizzari et al. [143] combine a group-cohesion force with a goal force. Their environment is discrete and uses a floor-field to guide the pedestrians. Another approach is the *PLEdestrians* algorithm by Guy et al. [40]. Based on the Principle of Least Effort [153], the authors propose a local greedy strategy that approximates the minimum of a biomechanical energy function in order to compute trajectories for individual agents. The method exhibits desirable emergent behaviors.

Unlike flow-based models, agent-based models struggle when coordinating the movements of dense crowds. This can lead to non-desired phenomena such as deadlocks, oscillations, slow movements with unnecessary turns and detours, or a high number of collisions.

Due to the gap between flow-based models and agent-based models, hybrid methods combining both paradigms have recently been considered. The method by Narain et al. [97] uses a dual representation of the crowd that is based on both individual agents and continuum dynamics. Like our model, agents have individual goals, but they can be forced to deviate from their preferred direction by the flow of the crowd. Contrary to their approach, our model omits continuum dynamics and simulates the tendency of humans to follow each other on a local and agent-based level. Our resulting herding behavior can therefore be related to Reynold's well-known model on flocks, herds and schools [114], while still allowing individual agent behaviors. Furthermore, our model measures local crowd density based on an agent's vision, and this overcomes problems that can occur with a grid-based approach. Examples of such problems are finding a feasible cell size, and abruptly changing densities caused by agents moving from one cell to another. Kountouriotis et al. [73] proposed to combine flow-based models and agent-based models with a local approach that is similar to ours. In their work, the interpolation between individual and coordinated movement is based solely on crowd density. By computing a perceived local crowd flow that can differ from agent to agent, we achieve a simple yet extensive interpolation. Our model also introduces a unique set of factors on which the interpolation is based, which enables the simulation of different agent profiles. In addition, our model allows the inclusion of different collision avoidance and global path-planning methods.

Schuerman et al. [117] proposed a method to add complex steering logic through the external use of *situation agents*. The authors claim that incorporating such logic in a steering algorithm itself leads to overly complicated algorithms that have to deal with many special cases. Our Stream model, by contrast, is based on an intuitive and simple interpolation which can be combined with existing steering methods and which does not pollute agents with additional computation. Still, our model shows an improved coordination and a significant reduction of deadlocks in high-density situations without the need of pre-processing the environment to identify narrow passages as in [117].

Other related work involves global path planning and collision avoidance among agents. Such methods can be used as black boxes within our model. A global path planning method related to our work is the *Indicative Route Method* (IRM) [63], which we have also discussed in Chapter 6. Given a global indicative route from an agent's start position to a goal position, the IRM computes an attraction point on the route in each step of the simulation and makes the agent approach this point using steering forces. Collision avoidance among agents is available via a range of velocity-based methods. One of the more popular ones is the ORCA method [133], which is based on *reciprocal velocity obstacles.* Another collision-avoidance method was presented by Karamouzas and Overmars [64]. It predicts future collisions for each agent and lets an agent take an action that guarantees collision-free movement. Park et al. [104] predict future collisions using a gaze movement angle. Vision-based approaches such as [88, 100] use a *field of view* (FOV) for each agent to detect and prevent collisions. In this work, we combine our model with the IRM, and we test it with several velocity- and vision-based collision avoidance methods [64, 88, 133].

9.2 | PRELIMINARIES

9.2.1 | AGENT REPRESENTATION

Similar to our modifications of MIRAN as described in Chapter 7, we represent each agent as a disk with a variable radius. The center of the disk is the current position of the agent. Such a disk representation is widely used in agent-navigation and crowd-simulation algorithms; see e.g. [31, 40, 59, 63, 79, 88, 133]. Research on more accurate representations for full-body agent navigation is rather novel and has been addressed only recently [103].

Each agent has a *field of view* (FOV), and an agent's steering behavior is based on a number of perceived neighboring agents. This is a fundamental difference from continuum-based and other flow-based methods [130] because these methods assume global knowledge of the environment. We assume that real people mainly execute and adapt their movement to visual input without global knowledge of the crowd. We therefore believe that a local vision-based approach is well-suited for approximating realistic crowd behavior and simulating emergent phenomena observed in real crowds.

Similar to Moussaïd et al. [88], we assume that an agent's FOV is a circular segment centered at the agent's current position and bounded by both a maximum look-ahead distance d_{max} and a maximum viewing angle ϕ. See Figure 9.2 for an example and Section 9.4 for the exact values used in our experimental setup.

9.2.2 | OVERVIEW OF THE STREAM MODEL

In each cycle of the simulation, we compute a velocity vector for each agent. This velocity vector is then applied to an agent's current velocity using a time-integration scheme such as Euler integration [10], which guarantees smooth paths [33].

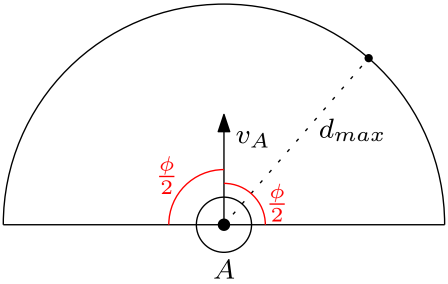


FIGURE 9.2: Agent A's *field of view* with a maximum viewing angle ϕ and a maximum look-ahead distance d_{max}, centered around A's velocity v_A. We set ϕ to 180° for all agents within our experiments, but the value can be adjusted if larger or smaller angles are desired for a particular application.

Let A be an arbitrary agent. We perform the following five steps in each simulation cycle:

1. We compute an *individual velocity* for agent A. This represents the velocity A would choose if no other agents were in sight. Our model is independent of the exact method that is used. Any method that computes a preferred velocity for agent A is a feasible choice, e.g. [52, 63].
2. We compute the *local crowd density* that agent A can perceive; see Section 9.3.1.
3. We compute the locally *perceived stream velocity* of agents near A; see Section 9.3.2.
4. We compute A's *incentive* λ. This incentive is used to interpolate between the *individual velocity* from step 1 and the *perceived stream velocity* from step 3; see Section 9.3.3.
5. The interpolated velocity is passed to a collision-avoidance algorithm. Our model is independent of the exact method that is used, and we have tested it with three popular ones [64, 88, 133].

9.3 | STREAM BEHAVIOR

A central concept in our model is the notion of locally perceived *streams* of agents. Intuitively, streams are flows of people that coordinate their movement by either aligning their paths or following each other. Streams can be observed in real-life situations as crowd density increases; see Figure 9.3. We base our model on the assumption that people tend to move by following a least-effort principle of energy-minimization [40, 124]. We postulate that actively forming and following streams at high densities is a more energy-efficient strategy compared to pursuing individual goals. This follows because the use of streams leads to fewer collisions and abrupt changes in the direction of movement.

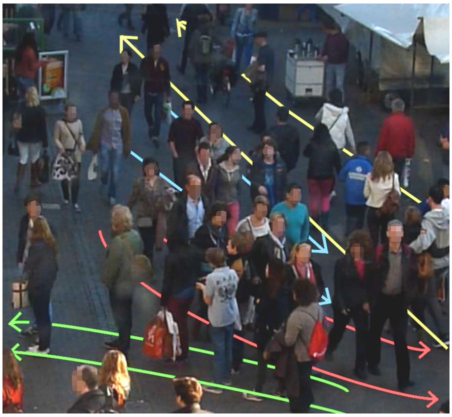


Figure 9.3: Example of stream formation in real-life situations. Arrows indicate the directions of streams. People between arrows of the same color belong to the same stream.

Local crowd density is one of the key factors that determines an agent's behavior in our model. In Section 9.3.1, we discuss three different ways to measure local crowd density, and we motivate why we chose to compute local density information using an agent's FOV. In subsequent sections, we discuss how to compute an agent's perceived stream velocity (Section 9.3.2) and its incentive (Section 9.3.3).

9.3.1 | Computing local density information

After agent A's individual velocity has been computed as an initial step, we calculate the crowd density $\rho \in [0, 1]$. We have tested three different density measures in a set of preliminary experiments: grid-based density, vision-based density, and density based on the cells of the underlying *Explicit Corridor Map* (ECM) navigation mesh [31] in our framework. The results of these preliminary experiments showed no clear practical advantage of one density measure over the others with respect to travel times, collisions, and deadlock situations. However, there are theoretical differences between the three measures, which favor the vision-based approach. In the remainder of this section, we discuss these differences.

In the first approach, we use the ECM to subdivide all navigable space into ECM cells [31]. A similar approach has been taken by van Toll et al. [141]. Each ECM cell is induced by exactly one edge of the medial axis graph of the virtual environment. We can therefore store the amount of space occupied by agents for each edge of the medial axis. We compute the total area of each ECM cell in an initial offline step, and it suffices to update density information whenever an agent leaves an ECM cell and enters an adjacent one. This approach is computationally efficient. However, it is not necessarily a *local* measure of crowd density because ECM cells may vary in size and span large areas of the environment. An agent might therefore switch from individual behavior to coordinated stream formation due to an increase in crowd density far away from the agent's actual position, and vice versa. As our Stream model is based on local density information, this approach turns out to be less suitable.

The second method uses a grid to approximate the navigable space of the environment. We have tested three different cell sizes of $1m^2, 4m^2$ and $9m^2$, and we compute density information for each grid cell. Like the ECM-based approach, it allows for constant-time updates of density information per agent because we only need to update it when an agent switches from one grid cell to an adjacent one. This is a more local approach compared to the ECM-based method, and it is therefore better suited for our streams model. However, the performance strongly depends on the grid cell size. Small cells increase the computational requirements and may lead to incorrect density measurements. The area around an agent might be densely packed except for certain gaps, and an incorrect density value is computed if such a gap happens to be evaluated. Large cells, however, yield the same issues as discussed for the ECM-approach because locality of the measurement is lost.

The third approach uses the agent's FOV to locally measure crowd density information. Since our model is based on the agent's FOV to a great extent, the extra overhead of using it to measure local density is small. With this approach, agents next to or slightly behind A will be considered when calculating density information. We determine the set $\mathcal{N}$ of all neighboring agents that have their current position (i.e. the center point of the disk representing the agent) inside A's FOV of $\phi = 180°$. By summing up the area $\Delta(N)$ occupied for each such agent $N \in \mathcal{N}$ and dividing it by the total area $\Delta(FOV)$ of A's FOV, we get the percentage value indicating how much of the FOV is occupied. In practice, it is highly unlikely that the FOV will ever be entirely occupied, though. Fruin [28] was the first to formalize the impact crowd density has on the safety of pedestrians. Fruin has introduced a six-stage *Level-of-Service* system, ranging from free movement without collisions to highly dense situations. According to this system, an FOV occupied to one third can already be considered a highly crowded situation. We therefore take the aforementioned percentage value, multiply it by 3 and cap it at a maximum of 1. This yields a maximum density value of 1 as soon as at least one third of A's FOV is occupied by other agents. Formally, we define the crowd density ρ as follows:

$$\rho := \min\left(\frac{3}{\Delta(FOV)}\sum_{N\in\mathcal{N}}\Delta(N),\ 1\right). \tag{9.1}$$

While agents with a position outside A's FOV might intersect its FOV, they will not be taken into consideration. However, agents with a position inside but close to the boundaries of A's FOV will be counted with their full area. On average, we expect the number of falsely selected agents to cancel out the number of falsely neglected ones. The results of our experiments in Section 9.4 show that this approximation works well in practice.

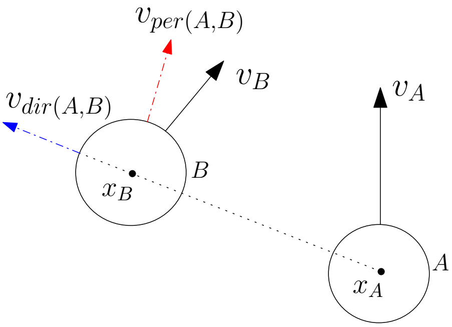


FIGURE 9.4: An example of the *perceived velocity* $v_{per(A,B)}$ based on an interpolation between v_B and $v_{dir(A,B)}$.

9.3.2 | THE PERCEIVED STREAM VELOCITY

The next step is to compute the direction and speed of the stream of neighboring agents as perceived by agent A. In situations where A is willing to coordinate with the crowd, our model lets A approach the perceived stream whenever the distance from A to the stream members is still large. If, by contrast, A is close to the stream, it will align its direction with the other members and follow the stream. We motivate this in Section 9.3.2.1, where we initially consider the case where only one single agent is perceived. In Section 9.3.2.2, we generalize the single agent procedure for multiple perceived agents. The overall perceived stream velocity is the average of the single stream directions and speeds for each agent.

9.3.2.1 | PERCEIVING A SINGLE AGENT

Let B be a single agent in A's FOV, and let x_A and x_B be the current positions of A and B, respectively. We define the perceived velocity $v_{per(A,B)}$ as an interpolation between B's actual velocity v_B and a vector $v_{dir(A,B)}$ of the same length that points along the line of sight between A and B. Formally, we let $v_{dir(A,B)} = \overline{(x_B - x_A)} \cdot \|v_B\|$ be the normalized vector between x_A and x_B scaled to the speed of B; see Figure 9.4. A factor $f_{A,B} = \rho \cdot d_{A,B}$ is used to angularly interpolate between the two vectors. Here, $\rho \in [0, 1]$ is the local density in A's FOV, and $d_{A,B} = \frac{\|x_B - x_A\|}{d_{max}}$ is the relative distance between A and B. Thus, $f_{A,B} \in [0, 1]$, and we use it to interpolate between v_B and v_{dir} along the smallest angle between the two.

If we assume a density of 1 in the above definition, the factor $f_{A,B}$ only depends on the relative distance between A and B. If B is on the edge of the view-distance of A, i.e. $\|x_B - x_A\| = d_{max}$, then $v_{per(A,B)}$ equals $v_{dir(A,B)}$. This makes A pursue a *follow strategy* [114] because A is attracted to B's current position. If, by contrast, A is close to B, then $v_{per(A,B)}$ is close to v_B, and A picks an *alignment strategy*.

The local density ρ is an extra factor that interpolates between the *follow strategy* and the *alignment strategy*: The higher the number of agents intersecting A's field of view, the more A is inclined to pick the *follow strategy*. This yields more compact crowd formations at higher densities and a wider crowd spread across the available free space at lower densities, which is a phenomenon observed in real crowds [46].

9.3.2.2 | PERCEIVING THE LOCAL STREAM

We define the local stream velocity perceived by agent A as the average of all perceived velocities that are taken into consideration, both with respect to direction and speed. We limit the total number of potential neighbors of A and only consider its five nearest neighbors that are currently in its FOV. This comparably small number corresponds to findings in research for flocks of birds [6], and has been used in related work, e.g. [64].

In theory, there can be arbitrarily many closest neighbors with equal distance from A, with positions on the border of a circle centered on A's position x_A. We do not propose a fixed rule of which five neighbors to pick in this situation. We need to ensure that our model computes a feasible stream velocity whenever there is an actual stream of agents. Since there is no such stream in this example, all possible choices of neighbors are equally feasible.

Let $\mathcal{N}_5$ be a set of up to 5 nearest neighbors of A. We define the *average perceived stream speed* s for A as follows:

$$s := \frac{1}{|\mathcal{N}_5|} \cdot \sum_{N \in \mathcal{N}_5} ||v_{per(A,N)}||. \tag{9.2}$$

The locally *perceived stream velocity* v_{stream} perceived by agent A is then defined as follows:

$$v_{stream} := s \cdot \frac{\sum\limits_{N \in \mathcal{N}_5} v_{per(A,N)}}{||\sum\limits_{N \in \mathcal{N}_5} v_{per(A,N)}||}, \tag{9.3}$$

which is the average perceived stream speed times the average direction of all perceived velocities scaled to unit length.

Since we define the local stream velocity as the average of a set of velocities, it can result in a null-vector when the corresponding velocities cancel each other out. In this case, agent A cannot adapt to the stream velocity, even if there is an actual stream of neighboring agents it should coordinate with. To avoid perceived stream velocities canceling each other out, we restrict the maximum angle between the velocities of agents A and B to strictly less than $\frac{\pi}{2}$. Perceived neighbors yielding a larger angle are not taken into consideration. This choice is further justified by the fact that an agent should only consider streams that are going roughly in its

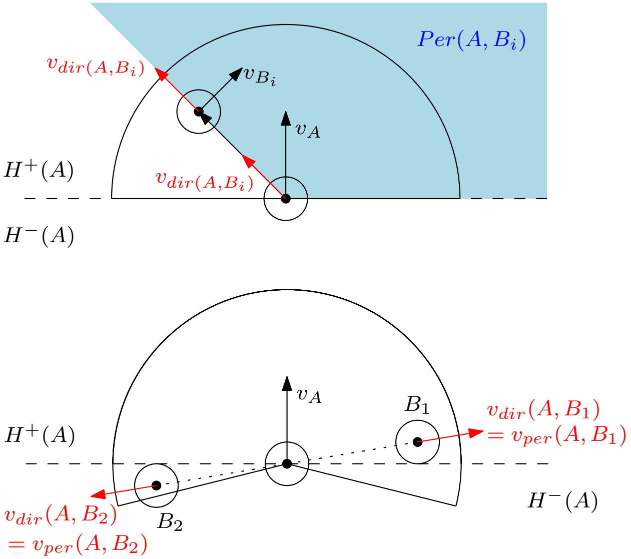


FIGURE 9.5: *Top:* An example constellation of two agents and the corresponding set $Per(A, B_i)$ of all possible perceived velocities. *Bottom:* An example of two agents in A's FOV with perceived velocities canceling each other out.

own preferred direction. Furthermore, due to the FOV with its viewing angle of π, perceived neighbors reside only in the closed halfplane $H^+(A)$ in front of A, which is induced by the line through x_A perpendicular to A's current velocity v_A. If either of these two restrictions is violated, perceived velocities may cancel each other out.

For any neighbor B_i of A, we let $\beta_i := \min(\angle(v_A, v_{B_i}), 2\pi - \angle(v_A, v_{B_i}))$ be the angle between A's current velocity and B_i's current velocity. By $H^-(A)$, we denote the complement of $H^+(A)$, which is the open halfplane behind A. Figure 9.5 (bottom) shows an example of two neighbors B_1, B_2 with $B_1 \in H^-(A)$, $B_2 \in H^+(A)$, and $\beta_1 > \frac{\pi}{2}$, which leads to $-v_{per(A,B_1)} = v_{per(A,B_2)}$. We now give a proof that the perceived velocities of any two neighbors of A cannot cancel each other out if the above restrictions are met.

Lemma 3. Let B_1, B_2 be two distinct perceived neighbors of agent A. Then the following property holds: For $i \in \{1, 2\} : x_{B_i} \in H^+(A)$ and $\beta_i < \frac{\pi}{2} \Rightarrow -v_{per(A,B_1)} \neq v_{per(A,B_2)}$.

Proof: Let B_i be an arbitrary perceived neighbor of A with $x_{B_i} \in H^+(A)$ and $\beta_i < \frac{\pi}{2}$. Let V_{B_i} be the set of all possible velocities v_{B_i} for agent B_i. Since $\beta_i < \frac{\pi}{2}$, it follows that $V_{B_i} = H^+(A)$. Let $Per(A, B_i)$ be the set of all possible perceived velocities $v_{per(A,B_i)}$. Then $Per(A, B_i) \subset V_{B_i} = H^+(A)$; see Figure

9.5 (top). Since this holds for all neighbors, it holds for neighbors B_1, B_2 in particular. From $Per(A, B_1) \subset H^+(A)$, we can conclude that $v_{per(A,B_1)} \in H^+(A)$ and $-v_{per(A,B_1)} \in H^-(A)$. This yields $-v_{per(A,B_1)} \notin Per(A, B_2)$ and therefore $-v_{per(A,B_1)} \neq v_{per(A,B_2)}$. □

In the degenerate case of all neighbors standing still, we set the perceived velocities to a minimum-threshold value as described in Section 9.3.2.1. With the above definitions and Lemma 3, we can now show that $v_{stream} \neq 0$.

Lemma 4. The perceived stream velocity $v_{stream} \neq 0$.

Proof: We prove this by induction on the number of neighbors $|\mathcal{N}_5|$. If $|\mathcal{N}_5| = 1$, the perceived stream velocity equals the perceived velocity of that one neighbor. Since we assume a minimum threshold on the perceived velocity, it follows that $v_{stream} \neq 0$. If $|\mathcal{N}_5| = k > 1$, we pick $k - 1$ neighbors, and we let v_{stream}^{k-1} be the perceived stream velocity for these $k-1$ neighbors. From the induction assumption, we know that $v_{stream}^{k-1} \neq 0$. Since the sum of velocities residing in $H^+(A)$ is in $H^+(A)$, too, it follows that $v_{stream}^{k-1} \in H^+(A)$. Furthermore, the angle between v_A and v_{stream}^{k-1} is strictly less than $\frac{\pi}{2}$. We can therefore use Lemma 3 for v_{stream}^{k-1} and the perceived velocity of the kth neighbor, and conclude that $v_{stream} \neq 0$. □

9.3.3 | Incentive

Now that A's individual velocity v_{indiv} and the perceived stream velocity v_{stream} have been computed, we define the *incentive* $\lambda \in [0, 1]$ of A to interpolate between v_{indiv} and v_{stream}. In Section 9.3.3.1, we discuss the set of factors that enable the simulation of various character profiles, and we describe how these factors influence λ. In Section 9.3.3.2, we describe how to interpolate between v_{indiv} and v_{stream} using λ.

9.3.3.1 | Computing the incentive

The incentive λ is defined by four different factors: *internal motivation* γ, *deviation* Φ, *local density* ρ, and *time spent* τ. We simulate the behavior of an agent A in a way such that – aside from the internal motivation factor – the most dominant factor among Φ, ρ and τ has the highest impact on the behavior of A. We define the incentive λ as follows:

$$\lambda := \gamma + (1 - \gamma) \cdot \max\big(\Phi, (1 - \rho)^3, \tau\big). \tag{9.4}$$

Internal motivation $\gamma \in [0, 1]$ determines a minimum incentive that an agent has at all times. This enables the simulation of various agent profiles such as a hurried agent or a strolling agent.

The local density factor ρ is defined in Section 9.3.1. For this factor, a non-linear relation with the incentive is desired, which makes the incentive drop rapidly when the local crowd density increases. To account for this, we induce a cubic descent by using $(1-\rho)^3$ in Equation 9.4. The *deviation* factor Φ makes agent A leave a stream when v_{stream} deviates too much from its preferred individual velocity v_{indiv}. We introduce a minimum threshold angle ϕ_{min}. Whenever the angle between v_{stream} and v_{indiv} is smaller than ϕ_{min}, the factor Φ will be 0. This yields stream behavior unless the other factors determine a different strategy. If the angle is greater than ϕ_{min}, we gradually increase Φ up to a maximum deviation of $2\phi_{min}$. Angles greater than this threshold correspond to a deviation factor of 1, thus yielding individual steering behavior. Let $\phi_{dev} := \min(\angle(v_{indiv}, v_{stream}),\, 2\pi - \angle(v_{indiv}, v_{stream}))$ be the angle between the velocities. We define the deviation factor Φ as follows:

$$\Phi := \min\left(\max\left(\frac{\phi_{dev} - \phi_{min}}{\phi_{min}}, 0\right), 1\right). \tag{9.5}$$

The *time spent* factor τ is used to make stream behavior less attractive the longer it takes the agent to reach its goal. We initially calculate the expected time τ_{exp} agent A will need to get to its destination. How this is done depends on how A's individual velocity is calculated, i.e. what method is used as a black box in the initial step of our model. If, for instance, an indicative route is used [63], the expected time can be calculated by weighting the length of the route with the local density ρ. This value can then be mapped to an expected time value according to the agent's preferred speed. We keep track of the actual simulation time τ_{spent} that has passed since A has started moving. We then define the *time spent* factor τ as follows:

$$\tau := \min\left(\max\left(\frac{\tau_{spent} - \tau_{exp}}{\tau_{exp}}, 0\right), 1\right). \tag{9.6}$$

9.3.3.2 | USING THE INCENTIVE

Given the incentive λ, we interpolate between v_{indiv} and v_{stream} as follows: We rotate v_{stream} towards v_{indiv} in a similar manner as the interpolation between v_{dir} and v_B is performed for a single neighbor; see Section 9.3.2. Let β be the smallest angle between the two vectors, and let $\beta_{rot} = \beta\lambda$ be the rotation angle between 0 and β, based on the incentive λ. We then rotate v_{stream} towards v_{indiv} by β_{rot}. In this step, however, the lengths of v_{indiv} and v_{stream} are not the same in general. Therefore, we also linearly interpolate the lengths of these vectors.

This concludes the description of our Stream model. In the following sections, we discuss the validation of our model and the set of experiments we have conducted.

9.4 | EXPERIMENTS

Our Stream model has been implemented in a framework based on the *Explicit Corridor Map* (ECM) [31]. The ECM is a time- and space-efficient navigation mesh for crowd simulation. All experiments have been conducted on a laptop running Windows 8.1 with a 2.4 GHz Intel Core i7-4700HQ 2-Core CPU, 8 GB RAM and an NVIDIA GeForce GTX 870M graphics card with 6 GB of GDDR5 memory. We used one CPU core for the computations. To compute a preferred individual velocity for each agent, we combined our model with the Indicative Route Method by Karamouzas et al. [63].

We used the following parameter values for our experiments. We set the agents' radii to 0.24 meters, the look-ahead distance d_{max} to 1.5 meters, and the maximum viewing angle ϕ to 180°. The latter reflects the approximate viewing range people can perceive in real life [147]. Preferred speeds were randomly chosen between 0.85 and 2.05 meters per second. For the simulation itself, we defined the time step between any two simulation cycles as 0.1 seconds.

We have validated our model both qualitatively and quantitatively. For a qualitative inspection of the crowd motions generated by Stream, we refer the reader to the corresponding video[1]. For a quantitative evaluation, we used the following metrics: The average number of collisions between agents, the average travel time per agent, the average expended energy per agent, the average traveled distance per agent, and the number of runs in which agents were are not able to reach their goal positions (deadlock situations). To approximate the expended energy per agent, we compute the average change in an agent's kinetic energy over its travel time, measured over the discrete time steps of our simulation.

9.4.1 | SCENARIOS

For the qualitative evaluation, we created a scenario that represents a virtual university, in which agents autonomously walk around under dynamically changing crowd-density conditions; see Figure 9.1. This scenario can be inspected in the corresponding video[1].

For our quantitative evaluation, we used five different scenarios, some of which have been proposed in the Steerbench framework for evaluating steering algorithms [121]; see Figure 9.6. Throughout all scenarios, agents that reached their goal were removed from the simulation. This prevents congestion near the goal areas, which would otherwise lead to meaningless results when measuring the average number of collisions.

---

[1] `https://youtu.be/XSusPwT81pI` (accessed January 13, 2016)

In the *merging-streams* scenario, two groups with a total of 250 agents merge to pass through a bottleneck and split again afterward. The goal is to test whether two streams merge and split as an emergent phenomenon within our model. The *crossing-streams* scenario features two groups of 50 agents that approach each other in a perpendicular manner. The goal is to test whether different streams can cross each other without heavy interference. The *hallway1* scenario shows one group of 200 distributed agents, and the *hallway2* scenario shows two groups that each have 100 distributed agents. These agents traverse the hallway in either one direction (*hallway1*) or two opposing directions (*hallway2*). The goal is to test our model in medium-density scenarios. In the *narrow-x* scenarios, we use a narrow hallway of $3m$ and two comparably large groups of x agents ($x = 50$ and $x = 100$). The agents try to reach the opposite ends of the hallway. The goal of this experiment is to test our model in high-density situations.

In addition, we have measured the running times of our model in two scenarios, denoted as *military* and *hallway-stress*. *Military* features a 200 x 200 meters footprint of the McKenna MOUT training site at Fort Benning, Georgia, USA; see Figure 9.6 (bottom). It represents a scene with small passages, open squares and large areas of free space, which could be part of a gaming or simulation application. Agents are placed at the border and pick random goal positions at the opposite side of the scene. Compared to randomly distributing the agents throughout the scene, this setup enforces high crowd densities in the center of the environment as the agents approach their goal positions. The goal is to test whether our model performs at interactive rates in these types of scenarios. In *hallway-stress*, we use a hallway of 30m to provide enough space for a large number of agents. The goal is to test whether our model performs at interactive rates for large numbers of agents when the environment enforces a high level of coordination.

9.4.2 | MODELING VARIOUS AGENT PROFILES

We have tested the effects of the incentive on an agent that wishes to cross a large stream of other agents; see Figure 9.7. This example can be qualitatively inspected in the video that accompanies this work. We turned off the *time spent* computations to enhance the display of the effect of *internal motivation* and *deviation*. With an internal motivation of 1, we get a constant incentive of 1. This makes the agent push through the stream to reach its goal position at the opposite side of the crowd. With an internal motivation of 0, and a threshold ϕ_{min} for the deviation factor of $\frac{\pi}{4}$, the agent is dragged away by the stream flow until the deviation factor causes the incentive to rise and makes the agent leave the stream.

This shows that our Stream model enables the simulation of various agent profiles such as impatient or relaxed agents, rude agents that push through a dense crowd, or polite agents that try to avoid pushing behavior and instead coordinate with other agents. When such profiles are based on findings from social-science studies,

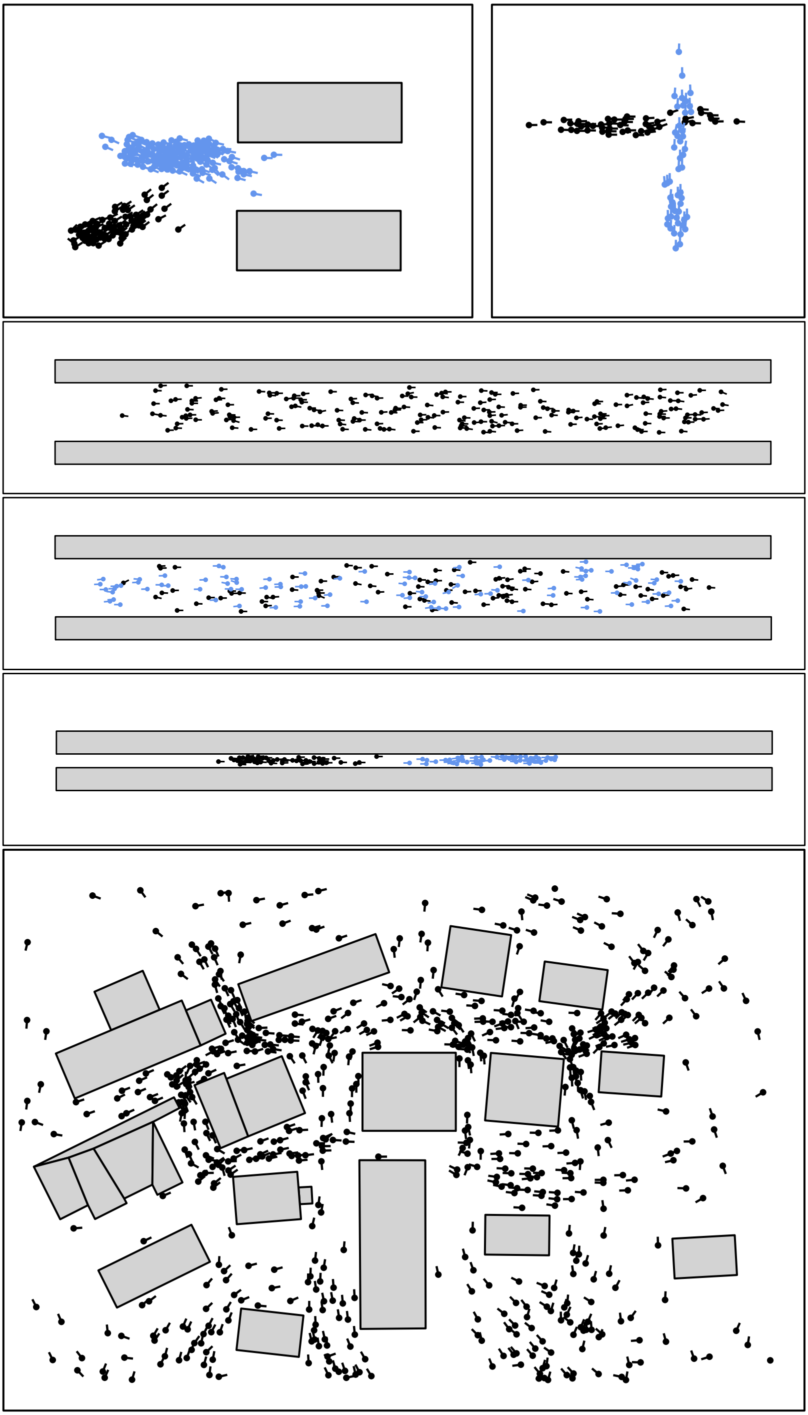


Figure 9.6: The different scenarios in our experiments are (from top to bottom): *merging-streams, crossing-streams, hallway1, hallway2, narrow-50* and *military.*

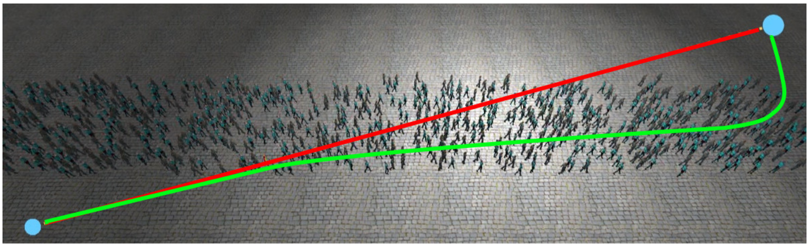


FIGURE 9.7: A stream of agents moving from left to right, and an agent trying to follow a path from the bottom left to the upper right corner. Red path: $\gamma = 1$. Green path: $\gamma = 0, \phi_{min} = \frac{\pi}{4}$. We refer the reader to our accompanying video for an animated sequence.

we believe that Stream can be a useful addition to future crowd simulators. This will display more complex agent behavior than state-of-the-art frameworks while still keeping the underlying computations simple and intuitive.

9.4.3 | COMPARING DIFFERENT COLLISION-AVOIDANCE METHODS

In a first set of experiments, we have tested Streams with three popular collision-avoidance methods [64, 88, 133]. For each method, we have computed the results for each metric mentioned in Section 9.4, averaged over 50 runs for each scenario. The results are depicted in Table 9.1 (average number of collisions) and Figure 9.8 (expended energy, travel times, traveled distances, and number of deadlock runs).

The following conclusions can be drawn from this experiment. In terms of the average expended energy per agent, the method by Moussaïd et al. [88] shows the overall best results. All three methods perform equally well in terms of the average traveled distance per agent. Only the method by Karamouzas and Overmars [64] yields comparably small distances in the *narrow-50* scenario. The reason is that the method uses a personal space radius for each agent, which becomes a problem in this dense scenario and leads to a deadlock situation in 100% of the tested cases. We used a default value of 0.5 meters for the personal space radius, which is in correspondence with [64]. In addition, we have run a preliminary set of experiments with lower values for the personal space radius. With lower values, deadlocks occur less frequently, but the method struggles with an increased number of collisions between the agents. In terms of travel times, all three methods perform equally well in low- to medium-density scenarios. In dense scenarios such as *narrow-50*, however, both ORCA [133] and the method by Karamouzas and Overmars result in a deadlock in almost all cases. This is the reason why no corresponding data is plotted in Figure 9.8. The method by Moussaïd et al. by contrast, resolves all tested scenarios without any deadlocks. ORCA struggles with a significantly higher number of collisions compared to the other two methods; see Table 9.1. This becomes especially apparent in the *merging-streams* scenario.

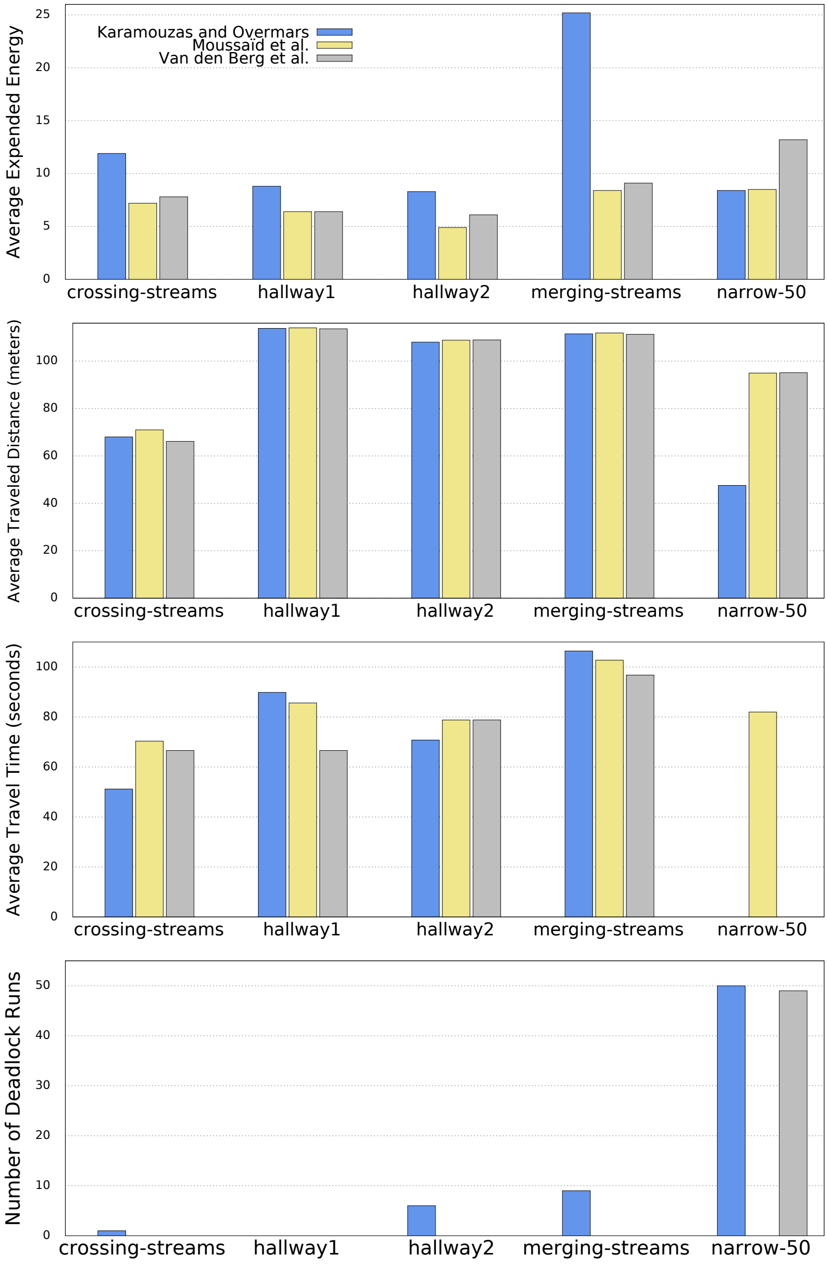


FIGURE 9.8: Results for our Stream model with three different collisions-avoidance methods in five different scenarios. The results are averaged over 50 runs of each method-scenario combination. Lower scores correspond to better results. Missing travel times data in *narrow-50* indicates deadlock situations in more than half of the runs.

	crossing-streams	*hallway1*	*hallway2*	*merging-streams*	*narrow-50*
Karamouzas and Overmars	**17.5**	**5.9**	**5.3**	**95.7**	**32.4**
Moussaïd et al.	111.9	28.7	16.6	290.4	170.3
Van den Berg et al.	1970.9	612.2	138.6	16555.8	3876.7

Table 9.1: The average number of collisions for our Stream model with three different collisions-avoidance methods in five different scenarios. The results are averaged over 50 runs of each method-scenario combination.

For the remainder of our experiments, we used the method by Moussaïd et al. The main reason is that this method performed significantly better in high-density scenarios. While the other two methods lead to a deadlock in almost all cases, the method by Moussaïd et al. was able to resolve all of them. Since Stream aims at improving high-density crowd flow while still being able to simulate low-density scenarios, this is a critical factor. Furthermore, the method does not show any significant disadvantages in any of the other metrics. The fact that our model already uses a FOV further justified the choice for the vision-based collision avoidance method by Moussaïd et al.

9.4.4 | Testing the effect of streams

We have compared our streams approach to the same scenarios when stream-behavior is turned off and only individual behavior is being displayed. We use the Indicative Route Method (IRM) [63] for path following, and we used the collision-avoidance method by Moussaïd et al. [88] because it yielded the best results in our experiments; see Section 9.4.3. We tested both low-density scenarios such as *hallway1* and *hallway2* and high-density scenarios such as *narrow-50* and *narrow-100.*

Table 9.2 (average number of collisions) and Figure 9.9 (expended energy, travel times, traveled distances, and number of deadlock runs) show the results of this experiment, averaged over 50 runs per scenario. It turned out that our Stream model causes slightly more collisions in low-to medium-density scenarios such as *crossing-streams* or the *hallway* scenarios. In high-density scenarios such as *merging-streams* and the *narrow* scenarios, however, the Stream model shows a significantly lower number of collisions due to the improved coordination among agents. A similar observation can be made for the expended energy. While the agents spend slightly more energy in *crossing-streams* and the *hallway* scenarios when Stream is used,

	crossing-streams	*hallway1*	*hallway2*	*merging-streams*	*narrow-50*	*narrow-100*
With Stream	111.9	28.7	16.6	**290.4**	**170.3**	**4787.3**
Without Stream	**34.7**	**10.5**	**13.0**	677.9	15799.2	43070.9

TABLE 9.2: The average number of collisions with and without our Stream model in six different scenarios. The results are averaged over 50 runs of each method-scenario combination.

the expended energy is lower in high-density scenarios due to the improved coordination among agents. The traveled distance is slightly higher when Stream is used because agents do not take the shortest path to their goals, but instead accept small detours to improve crowd coordination and flow. The increase in travel distances is comparably small, though. The most significant difference between the two methods shows when we compare the average travel times and occurrence of deadlocks. With our Stream model, the average travel times are slightly higher in scenarios that can still be handled without Stream. This is caused by the longer travel distances and aligning of motions among the agents. However, when we turn off the Stream model, only *crossing-streams* and *merging-streams* could be resolved without any deadlocks while all other scenarios resulted in a deadlock in 40 out of 50 runs for the *hallway* scenarios, and in 49 and 50 runs for *narrow-50* and *narrow-100*, respectively. With Stream, by contrast, no deadlocks occurred in all scenarios expect for *narrow-100*, in which a comparably small number of 15 out of 50 runs could not be resolved.

From these observations, we conclude that, overall, our Stream model performs equally well as a purely individualistic crowd model in low- to medium-density scenarios. In high-density scenarios, Stream significantly reduces the occurrence of deadlocks and improves the overall coordination and flow of a crowd.

9.4.5 | PERFORMANCE

In a final set of experiments, we tested our Stream model with respect to its real-time performance. Figure 9.10 shows the average running times needed to compute one step of the simulation for an increasing number of agents in the *military* and *hallway-stress* scenarios. Each measurement shows the average step time of 10 runs.

The results show that we could simulate up to 1700 agents at interactive rates in the *military* scenario. For higher numbers, deadlocks frequently started to occur, which is what we expected given the size of the scene compared to the number of agents. In the *hallway-stress* scenario, we could simulate up to 1100 agents simultaneously

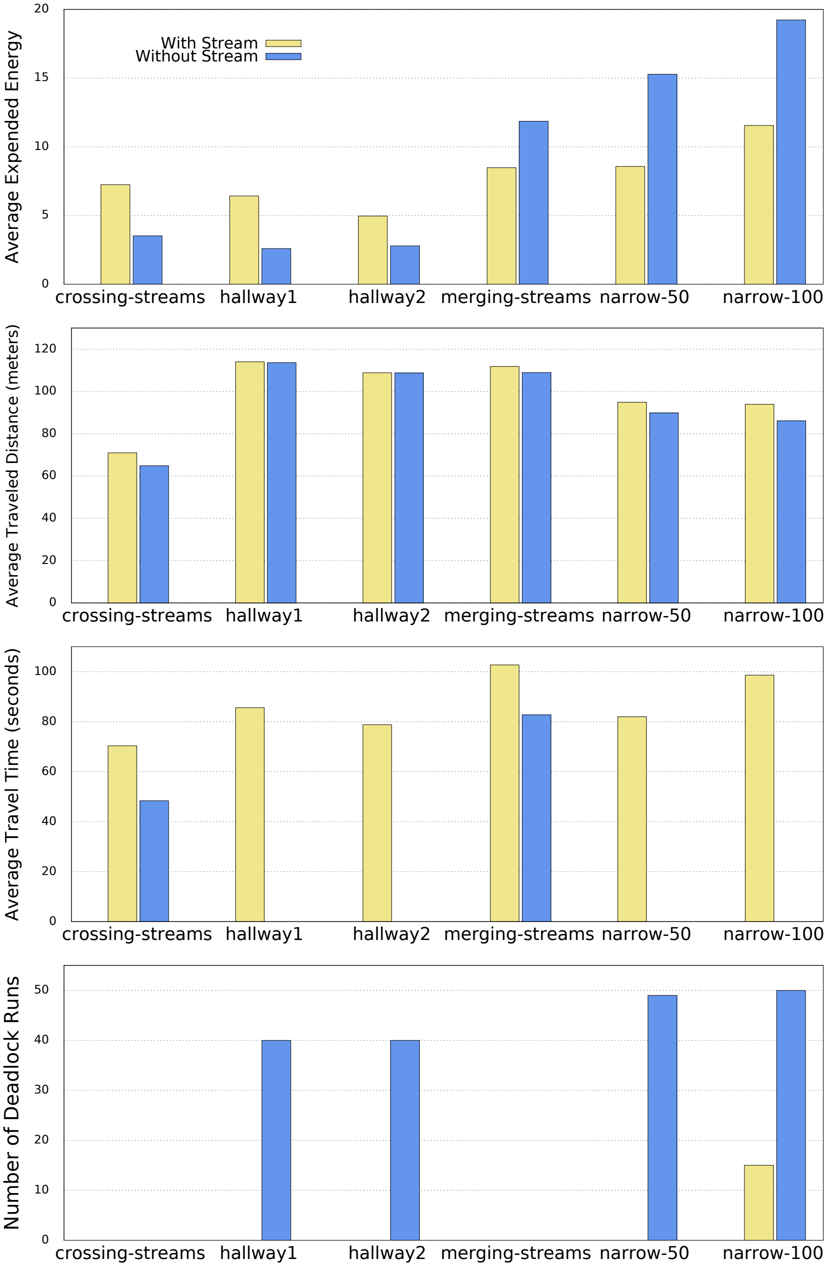


Figure 9.9: Comparison between crowd behavior with and without our Stream model in six different scenarios. The results are averaged over 50 runs of each method-scenario combination. Missing travel times data in the *hallway* and *narrow* scenarios indicates deadlock situations in more than half of the runs.

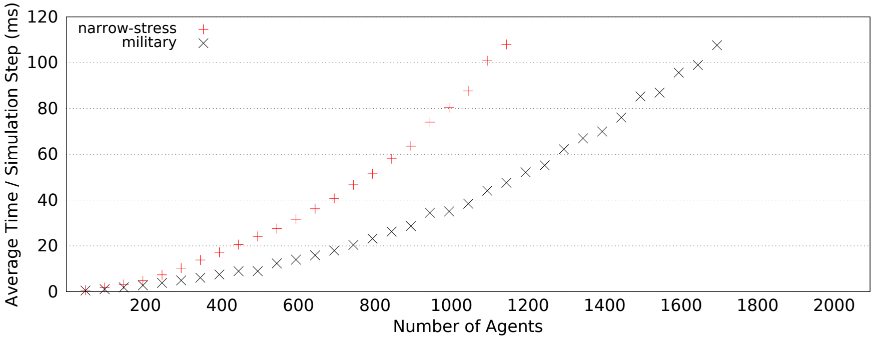


Figure 9.10: Average running times needed to compute one step of the simulation for an increasing number of agents in the *military* and *hallway-stress* scenarios. Each measurement shows the average step time of 10 runs.

at interactive rates on a single CPU. We conclude that our model runs at interactive rates for larger numbers of agents, even when coordination among the agents is high.

9.5 | LIMITATIONS

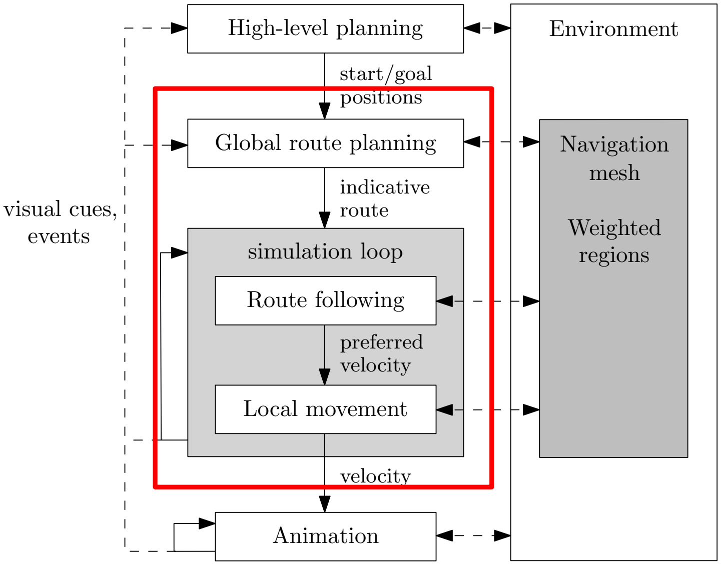
Many problems with dense crowds are caused by the global path planning step of the planning hierarchy; see Section 1.2. Whenever the global paths of a large number of agents intersect in the same point of the environment, the probability for deadlocks and an overall low throughput is high. This happens at the corners of obstacles when many agents are following the shortest path around these obstacles. Our model cannot prevent deadlocks entirely because it is designed to resolve problems on a local level. It is not designed to let agents dynamically re-plan their global path, or to make use of the full free space around obstacles. Improvements on a higher planning level are the subject of current research. Results in that field may strengthen the applicability of our model even more in the future.

We have shown that our model allows real-time simulation for up to 1700 autonomous agents on one CPU core in a medium-sized environment that contains both narrow passages and areas of open space; see Figure 9.10. However, since agents are simulated as individuals, computation is still expensive. When the application requires tens of thousands of agents with only a few distinct goals, a flow-based model may be a better choice.

Lastly, the coordination of real crowds depends significantly on social factors and group behavior. Stream provides a unique set of factors to simulate various agent profiles that could be based on social-science studies. However, it simulates how an agent perceives and reacts to neighboring agents on a purely geometrical level, and it does not yet take higher-level aspects into account.

This concludes the chapter on our Stream model. In the next chapter, we will present another novel method, which – similarly to the model presented in this chapter – can also be related to coordinating multiple virtual agents. Instead of coordinating arbitrary numbers of agents in dense situations, the method discussed in the next chapter aims at introducing social behavior for small pedestrian groups to establish more coherent and socially-friendly walking patterns.

SOCIAL GROUP BEHAVIOR



With the Stream model as described in the previous chapter, we are able to improve the coordination among agents in arbitrary crowd-density situations. The Stream model itself is based on the walking behavior of individual agents with individual parameter settings that can be used to simulate particular agent profiles such as impatient agents that display pushing behavior, or agents that are willing to comply with the local crowd flow around them. Another aspect that is important for the simulation of believable crowds and that is missing in the previous model is social-group behavior. Empirical research shows that up to 70% of crowd members walk in small social groups in urban environments and public places [15, 57, 89]. Existing methods model explicit formations to keep groups coherent [70] and in socially-friendly formations [65, 149]. Such formations have been observed in real crowds [89], but they are not strictly kept at all times due to the wide range of factors that influence a group's walking behavior in real-life situations. We therefore believe that explicitly modeling such formations may yield artificial-looking group behavior. Groups may lack flexibility and put too much emphasis on maintaining an explicit formation. For instance, groups might not be able to temporarily split and instead take unrealistic detours to keep a particular formation.

In this chapter, we present a novel method named *Social Groups and Navigation* (SGN) to simulate the walking behavior of small pedestrian groups. SGN is based on the social-force model by Moussaïd et al. [89] and the vision-based collision avoidance method by Moussaïd et al. [88], which we have modified and extended to yield more coherent and socially-friendly walking behavior. We do not explicitly model social formations. We instead introduce quantitative metrics to measure the *coherence* and *sociality* of small pedestrian groups, and we use these metrics to let formations emerge from the group members' attempts to stay coherent and social. The generated group behavior is more flexible and diverse than with existing methods. For instance, SGN allows groups to temporarily split to avoid dynamic obstacles such as other agents or groups, and groups automatically re-organize themselves when coherence is lost. Thus, SGN handles social group behavior on both global and local levels of a crowd simulation framework.

In Section 10.1, we discuss work that is related to our SGN method. In Section 10.2, we present basic settings, we give an overview of SGN and its simulation loop, and we discuss how to integrate SGN in a larger crowd simulation framework. In Section 10.3, we present the details of the method itself. In Section 10.4, we conduct experiments and compare our SGN to the methods by Moussaïd et al. [88, 89] with respect to social-group behavior. Furthermore, we show that SGN can be used to simulate several thousands of agents in real-time.

This chapter is based on the following publication:

[54] N. Jaklin, A. Kremyzas, and R. Geraerts. Adding sociality to virtual pedestrian groups. In *21st ACM Symposium on Virtual Reality Software and Technology (VRST 2015)*, pages 163–172, 2015.

10.1 | RELATED WORK

For a general overview of the field of crowd simulation, we refer the reader to the books by Thalmann and Musse [128] and Pelechano et al. [107]. In this section, we focus on selected work related to our SGN method.

Musse and Thalmann [91] described a model to simulate group behavior based on inter-groups relationships. Their model uses a small set of simple parameters such as *interests, emotional status,* and *domination* for the agents. Their work can be seen as an early attempt to capture real-life crowd behavior with a focus on social relationships between groups and their members.

Kamphuis and Overmars [60] presented a method to simulate coherent groups. They focus on large groups such as military armies. Their method handles both global path planning and local steering. Socially-friendly formations are not supported, and coherence is not re-established when it is lost during the simulation.

Qiu and Hu [111] presented a model to simulate pedestrian groups based on utility theory and social comparison theory. Their method enables agents to switch between different groups during the simulation. It does not explicitly model coherence and socially-friendly formations of the generated groups.

Moussaïd et al. [89] use video recordings of urban areas to collect empirical data of pedestrian crowds. They also describe a social-force model to simulate the walking behavior of small pedestrian groups. Our SGN method is based on this social-force model.

Inspired by Moussaïd et al. [89], Karamouzas and Overmars [65] presented a velocity-based approach to simulate small pedestrian groups. Socially-friendly formations are explicitly modeled, and the method optimizes a cost function to maintain group coherence and guarantee collision-free movement.

Kimmel et al. [70] presented an extension to the *Velocity Obstacle* (VO) approach [26] to simulate social-group behavior. The authors define a geometrical *Loss of Communication Obstacle* (LOCO) that can be combined with a VO to generate collision-free movement for small groups. Such groups try to stay close to each other during the simulation. Coherence is handled such that no agent is further away from the group than a particular threshold distance. There is no explicit formulation of socially-friendly formations, and the method works only locally as an extension of the VO method and its reciprocal variants, e.g. van den Berg et al. [133].

Park et al. [105] presented a model that considers higher-level social interactions between the group members. It assigns a *leader* to each group, and it handles group-coordination strategies based on common ground theory.

Wu et al. [149] combined the work by Karamouzas and Overmars [65] with the vision-based steering approach by Ondřej et al. [100]. They validated their method by comparing the distortion, dispersion, and out-of-formation metrics of their simulation with data from a real crowd.

Huang et al. [50] presented a path planning method to simulate coherent and persistent groups. The method is based on the *Local Clearance Triangulation* by Kallmann [59], and it handles groups as deformable shapes. Deformations as well as splitting and merging actions of a group influence the overall costs of a path.

Compared to most of the above methods, our SGN method handles social-group behavior on both global and local planning levels. We achieve this by not only adding coherent and socially-friendly walking behavior, but also letting groups re-establish their coherence in case they have to temporarily split during the simulation.

10.2 | PRELIMINARIES

10.2.1 | BASIC SETTINGS

Assume we are given k groups of agents $\mathcal{G}_i$, $1 \leq i \leq k$. We assume group sizes $|\mathcal{G}_i|$ of 2 through 4, which corresponds to observations made in real pedestrian crowds [89]. Note that these group sizes are not a hard constraint, and SGN can be easily modified to simulate bigger groups; see Section 10.3.6.3. The method is designed in a modular way, and it can be used to also simulate individual agents that do not display group behavior by switching off the corresponding group-related parts of the method. It also allows to simulate mixed scenarios with both groups and individuals. For ease of explanation, we assume that all groups are present at the start of the simulation. However, our method can be easily modified to let groups enter the simulation at a later point in time.

For $1 \leq i \leq k$ and $1 \leq j \leq |\mathcal{G}_i|$, we denote by A_{ij} the jth member of group $\mathcal{G}_i$, which is represented as a disc with radius r_{ij} (in m) and a mass $m_{ij} = 320 r_{ij}$ (in kg), following the definition by Moussaïd et al. [88]. By x_{ij}, we denote the center point of the disc that represents the agent, and we refer to it as the agent's *position.* In addition, each agent has a *personal space radius* $r'_{ij} \geq r_{ij}$. Each agent A_{ij} has a *preferred speed* s_{ij}, and each group has a *preferred group speed* $s_i = \min\limits_{1 \leq j \leq |\mathcal{G}_i|} s_{ij}$, which is the smallest preferred speed of its members. Furthermore, each agent A_{ij} has a *field of view* (FOV), which is a circular segment centered in the agent's current position x_{ij} with a maximum viewing distance d_{ij} and a viewing angle of Φ_{ij}. We say that an agent $A_{i'j'}$ is *visible* to an agent A_{ij}, if the FOV of A_{ij} contains at least one point of the disc that represents $A_{i'j'}$. We assume that $d_{ij} \geq 2 \sum\limits_{j'=1}^{|G_i|} r'_{ij'}$ to ensure that an agent can visually perceive all its group members when they are lined up in front of the agent, with the personal spaces of any two consecutive agents overlapping in at least one point (Figure 10.1). This is important for (re-)establishing group coherence; see Section 10.3.3 for details.

Each group $\mathcal{G}_i$ has a goal area G_i. We assume that each group has a feasible global route to its goal area. A feasible global route can be computed with any existing path planning method that ensures clearance from static obstacles for disc-shaped agents, e.g. [31, 59, 99]. To ensure collision-free movement for *all* group members, the global route should keep clearance from obstacles that corresponds to the *largest* disc radius of all group members.

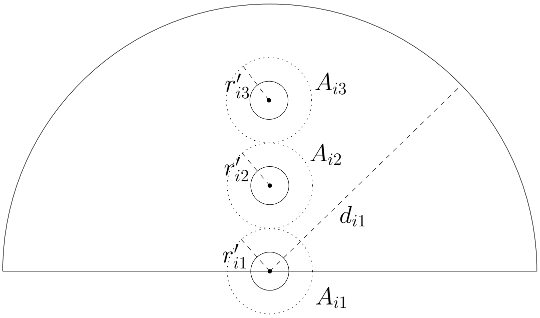


FIGURE 10.1: Example of a group of 3 agents that are lined up. Agent A_{i1} can perceive its fellow group members because its viewing distance d_{i1} is larger than the sum of the personal space diameters of all group members.

10.2.2 | OVERVIEW OF THE SGN METHOD

In short, our method works as follows: A group that enters the simulation first establishes its coherence by letting all members walk individually towards the initial group leader. All coherent groups walk towards their goal along a shared global path that is initially computed. While walking, social forces try to make the group members stay coherent and social. Whenever coherence is lost, a group re-establishes its coherence by letting the leader wait for its fellow members as soon as the local crowd density around the leader is low. Figure 10.2 provides a visual overview of our SGN method and its two main modes: *coordination mode* and *group-walking mode*.

We now give an overview of the initialization step in Section 10.2.2.1, and we continue with the simulation loop in Section 10.2.2.2. In Section 10.2.3, we discuss how our method can be integrated into existing crowd simulation frameworks.

10.2.2.1 | SGN INITIALIZATION

The initialization step for our SGN method is as follows. For each group $\mathcal{G}_i$ in the simulation, we perform the following actions:

- Set $\mathcal{G}_i$ to *coordination mode* (Section 10.3.3).
- Assign to an arbitrary member of $\mathcal{G}_i$ the role of the *leader* L_i (Section 10.3.1).
- Compute an indicative route π_{ij} to L_i for each member A_{ij} that is not L_i.
- Compute a global indicative route π_i from L_i to the goal area G_i using existing path-planning methods.

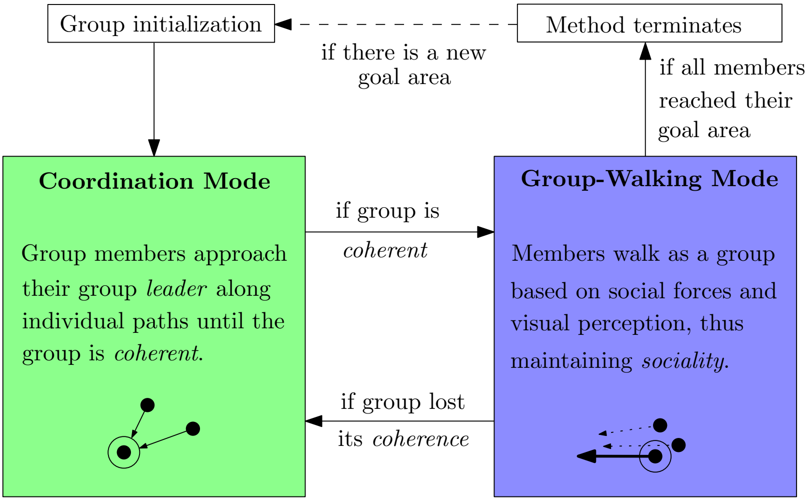

Group initialization
Method terminates
if there is a new goal area
if all members reached their goal area
Coordination Mode
Group members approach their group *leader* along individual paths until the group is *coherent.*
if group is *coherent*
Group-Walking Mode
Members walk as a group based on social forces and visual perception, thus maintaining *sociality*.
if group lost its *coherence*

FIGURE 10.2: Overview of the SGN method and its two main modes: *coordination mode* and *group-walking mode*.

10.2.2.2 | THE SGN SIMULATION LOOP

The simulation loop for our method is as follows. For each group $\mathcal{G}_i$ in *coordination mode* (Section 10.3.3), we perform the following actions:

- Compute a preferred velocity for each non-leading member A_{ij} to move along π_{ij} with the preferred speed s_{ij}. Any existing path following method can be used here.
- Pass the preferred velocities to a modified version of the collision-avoidance method by Moussaïd et al. [88] (Section 10.3.5).
- Check for each *waiting* member whether there is an agent of the same group in its personal space. If so, set that agent to a *waiting* state, too.
- If all group members are in a *waiting* state, then set $\mathcal{G}_i$ to *group-walking mode* (Section 10.3.4).

For each group $\mathcal{G}_i$ in *group-walking mode* (Section 10.3.4), we perform the following actions:

- (Re-)assign the role of L_i to the group member that is closest to G_i, measured via the curve-length distance along π_i (Section 10.3.1).
- Determine the current last member l_i of the group, which is farthest away from G_i, measured via the curve-length distance along π_i (Section 10.3.1).

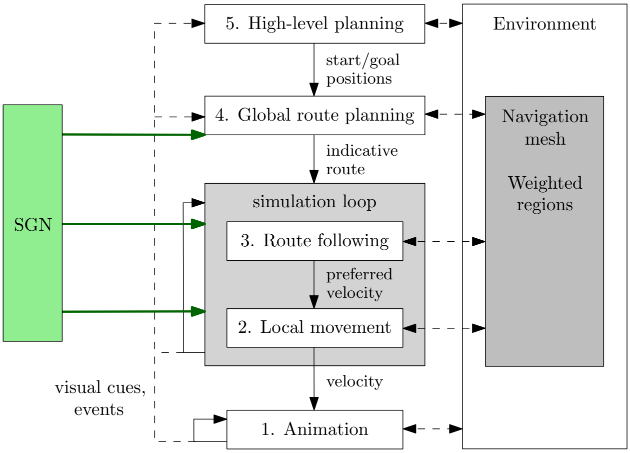


FIGURE 10.3: Example of a multi-level crowd-simulation framework [142] into which our SGN method has been integrated. SGN affects agent behavior on the global-route planning level, the route-following level, and the local-movement level.

- Compute a preferred velocity for each member along the group's global path π_i using the preferred group speed s_i. Any existing path following method can be used here.
- Pass the preferred velocities to a modified version of the collision-avoidance method by Moussaïd et al. [88] (Section 10.3.5).
- Compute the acceleration for each agent using a modified version of the social-force model by Moussaïd et al. [89] (Section 10.3.4).
- If $\mathcal{G}_i$ is not *coherent* (Section 10.3.2) and the density around L_i is smaller than 0.7 pedestrians per m^2, then set $\mathcal{G}_i$ to *coordination mode* (Section 10.3.3).

10.2.3 | INTEGRATION OF SGN INTO A CROWD SIMULATION FRAMEWORK

We assume that SGN is used in the context of a larger crowd simulation framework. We have implemented it within the framework described by van Toll et al. [142]; see Figure 10.3. Note that SGN does not depend on this particular framework. It can be easily integrated into any framework that treats global route planning, local route following, and micro-behavior such as collision avoidance as separate steps in the simulation cycle, e.g. Curtis et al. [17].

This concludes the overview of our method. In Section 10.3, we describe the details of all aforementioned steps.

10.3 | The *Social Groups and Navigation* method

10.3.1 | Leader and last member

For each group $\mathcal{G}_i$, we define a *leader* L_i and a *last member* l_i. These roles are updated and re-assigned at the end of each simulation cycle. The leader is defined to be the group member that is closest to the goal area, measured via the curve-length distance along the group's global path. Similarly, the last member is defined as the member that is farthest away from the goal area; see Figure 10.4 for an example. The only exception is in the initialization phase of our method. Here, no global path has been computed yet, and the role of the leader is therefore assigned to an arbitrary member.

Note that the global path serves as an *indicative route* [63], and the agents' positions are in general *not* located exactly on that route. It depends on the path following method what points on the route are used to determine the roles of the leader and the last member. A feasible option is to define a *reference point* on the global path for each agent, e.g. the point on the global path that is closest to an agent's position [52].

10.3.2 | Coherence and sociality

Let x_{ij} be the position of an agent A_{ij} with viewing distance d_{ij} and radius r_{ij}. Given the roles of a leader and a last member for each group, we define the *coherence*

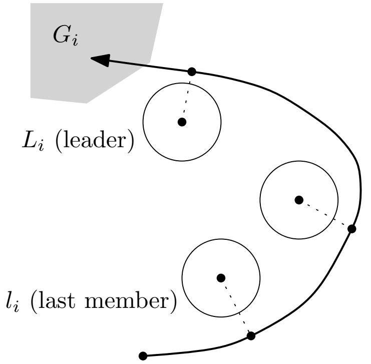


FIGURE 10.4: Example of a group $\mathcal{G}_i$ with a leader L_i and a last member l_i, following an indicative route to a goal area G_i. The discs resemble the group members themselves, and we do not display their personal space discs. The dotted line segments indicate the distance from each agent to its reference point on the indicative route.

and *sociality* of a group in the following way:

Definition 10.1. Let $\mathcal{G}_i$ be a group with leader A_{ij} and last member $A_{ij'}$. We say that $\mathcal{G}_i$ is *coherent* iff $||x_{ij} - x_{ij'}|| \leq d_{ij'} + r_{ij}$.

In other words, a group is coherent when at least one point of the disc that models the leader can potentially be seen by the last member. Note that Definition 10.1 does not reflect whether the leader is actually *inside* the FOV of the last member. As long as their distance is within the defined range, the group is coherent, even when the last member is not looking in the leader's direction.

Furthermore, we define a *social threshold distance* d_{social}. Intuitively, it is a maximum distance that two members of the same group are allowed to keep from each other while still being able to socially interact. This threshold distance is based on empirical observations [16, 27, 89, 153]. It should not be larger than the minimum of the viewing distances of all agents, i.e. $\forall i \in [1, ..., k]\ \forall j \in [1, ..., |\mathcal{G}_i|]\ d_{social} \leq d_{ij}$.

Definition 10.2. Let $\mathcal{G}_i$ be a group. We say that $\mathcal{G}_i$ is in a *partially social* configuration iff $\forall j \in [1, ..., |\mathcal{G}_i|]\ \exists j' \in [1, ..., |\mathcal{G}_i|],\ j \neq j'$, such that A_{ij} and $A_{ij'}$ are mutually visible (inside each other's FOV) and $||x_{ij} - x_{ij'}|| \leq d_{social} + r_{ij} + r_{ij'}$.

Definition 10.3. Let $\mathcal{G}_i$ be a group. We say that $\mathcal{G}_i$ is in a *totally social* configuration iff $\mathcal{G}_i$ is partially social and $\forall j \in [1, ..., |\mathcal{G}_i|]\ \forall j' \in [1, ..., |\mathcal{G}_i|],\ j \neq j'$, A_{ij} and $A_{ij'}$ are mutually visible.

In other words, a group is partially social when each member has at least one mutually visible other member within the social threshold distance, and it is totally social when, in addition, all members are mutually visible.

10.3.3 | COORDINATION MODE

Whenever a group loses its coherence, it enters *coordination mode*. In this mode, the members of a group will gather around their leader to (re-)establish coherence. The leader enters a *waiting* state and does not move until group coherence is established. *Coordination mode* is also the default mode of each newly spawned group in the simulation. In other words, as soon as a group enters the simulation, it will first start coordinating to establish group coherence before moving towards the goal area as a group.

Each non-leading member of a group in *coordination mode* first computes a route to the leader and starts following it. How this is done is independent of our method. Similar to the global route planning for an entire group, any path planning method that guarantees clearance from obstacles is sufficient in this step [31, 59, 99]. In the same way, any existing route-following method can be used that takes an indicative route or similar guidance path as an input and computes a preferred velocity for each agent. In *coordination mode*, this preferred velocity is then passed to a collision-avoidance method; see Section 10.3.5.

Any non-leading member follows its route to the leader until it detects a member of its own group that is in a *waiting* state. When *coordination mode* starts, only the leader is in a *waiting* state. Members that are sufficiently close to the leader enter a *waiting* state, too. We use an agent's personal space to determine whether another agent is sufficiently close in the following way: At the end of each simulation cycle, each *waiting* member of a group in *coordination mode* checks whether there is a *non-waiting* member in its personal space. If so, that *non-waiting* member switches to a *waiting* state, too. Since we assume the viewing distance d_{ij} of each agent A_{ij} to be at least $2 \sum_{j'=1}^{|G_i|} r'_{ij'}$ (see Section 10.2), the group will always be coherent as soon as all members have switched to a *waiting* state. This ensures that we can safely set the group to *group-walking mode* when there are no *non-waiting* members left at the end of a simulation cycle.

10.3.4 | GROUP-WALKING MODE

In this mode, each group $\mathcal{G}_i$ moves along its global path π_i to the goal area G_i. Any existing route-following method can be used that takes an indicative route or similar guidance path as an input and computes a preferred velocity for each agent. This preferred velocity is then passed to a collision-avoidance method; see Section 10.3.5. Afterwards, contrary to *coordination mode*, the preferred velocity is passed to a social-force model that maintains group-coherence and sociality; see Section 10.3.6.

After the group has moved, we check whether it is still coherent according to Definition 10.1. If not, the group needs to re-establish its coherence. In real-life, we expect the leader to wait for its fellow group members in a non-congested area of the environment. If such an area is not available, the group will not be able to re-establish its coherence until it reaches an area of low crowd density. Thus, in order to prevent a leader from stopping in the middle of a highly dense situation when coherence is lost, we check whether the local crowd density around the leader is smaller than a threshold value of 0.7 agents per m^2. This value is based on the *Pedestrian Level Of Service* (PLOS) system proposed by Fruin [28]. Only when both conditions are met, i.e. when the group lost its coherence and when the local crowd density around the leader is small, we set the group back to *coordination mode*.

10.3.5 | COLLISION AVOIDANCE

Within our SGN method, we use the vision-based collision-avoidance method by Moussaïd et al. [88] with some modifications. We keep the following core concepts as proposed in the original method:

Let v_{ij} be the preferred velocity of agent A_{ij} as computed by the path-following algorithm that is used. Let α_0 be the corresponding angle of v_{ij} measured against the agent's line of sight. Let O_{ij} be the last visible point in A_{ij}'s FOV that lies in the direction α_0. Let $\alpha \in [-\frac{\Phi_{ij}}{2}, +\frac{\Phi_{ij}}{2}]$ be a candidate angle direction, and let Ω_α be the last visible point in A_{ij}'s FOV that lies in the direction of α. Let T_α be the point in the direction of α that is the last collision-free position within the agent's FOV. Figure 10.5 shows an example of the situation. The desired direction is then defined as

$$\alpha_{des} = \underset{\alpha \in [-\frac{\Phi_{ij}}{2}, +\frac{\Phi_{ij}}{2}]}{argmin} \quad d(\alpha),$$

where $d(\alpha) = \sqrt{d_{ij}^2 + f(\alpha)^2 - 2d_{ij}f(\alpha)\cos(\alpha_0 - \alpha)}$.

In the original method, the term $f(\alpha)$ is defined as the distance from the agent's position to T_α. If no collision occurs within the distance of d_{ij}, then T_α coincides with Ω_α, and $f(\alpha)$ is therefore set to d_{ij}. For our SGN method, we modify the definition of $f(\alpha)$ in the following way: We let $F(\alpha)$ be the perpendicular foot of O_{ij} on the straight-line segment between x_{ij} and Ω_α (Figure 10.5). We then define

$$f(\alpha) = \min(|x_{ij}T_\alpha|, |x_{ij}F_\alpha|).$$

The moment when the next directional change occurs should not solely be based on the impending collisions, but also on the distance to O_{ij}. In other words, an

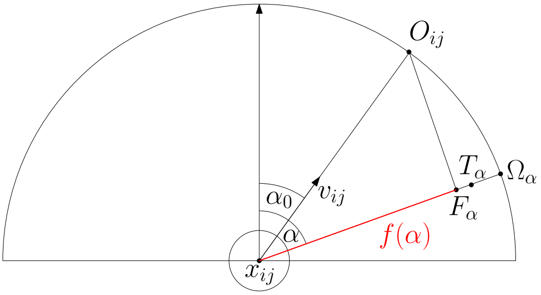


FIGURE 10.5: Example of the situation during the modified collision-avoidance method for a candidate angle α. Here, the point $F(\alpha)$ is closer to x_{ij} than T_α. Thus, $f(\alpha)$ is set to $|x_{ij}F_\alpha|$.

agent should also change its direction when it reaches a point where moving on in its current direction would increase the distance to O_{ij}, even with no impending collisions. The point where this happens is $F(\alpha)$. Without that option, an agent might 'overshoot' in the desired direction.
Now that α_{des} is computed, we can compute the desired walking speed s_{des}. From the original method, we adopt the concept of a relaxation time τ. This relaxation time ensures that an agent chooses its speed such that there is enough time to avoid a collision within the given time frame. Let s_{ij} be the agent's preferred speed, and let d_{col} be the distance between the agent and the first collision in the direction of α_{des}. We define

$$s_{des} = \min(s_{ij}, d_{col}/\tau).$$

This ensures that the agent moves at its preferred speed when there are no impending collisions in the given time frame, and it slows down accordingly when needed. Finally, let v_{des} be the resulting desired velocity in the direction of α_{des} and scaled to the desired speed s_{des}.

10.3.6 | SOCIAL-FORCE MODEL

We apply social forces to each group in *group-walking mode* after the collision-avoidance step. The social forces are based on the model by Moussaïd et al. [89], with some modifications. We will first explain the model and then discuss what details have been modified compared to the original method.

Given an arbitrary agent A_{ij}, its desired velocity v_{des} after the collision-avoidance computations, and the actual velocity v from the previous simulation step, we compute the acceleration $\frac{dv}{dt}$ in the following way:

$$\frac{dv}{dt} = \frac{v_{des} - v}{\tau} + \frac{1}{m} \sum_{u=1}^{k} \sum_{v=1}^{|\mathcal{G}_u|} f_{uv} + \frac{1}{m} \sum_{w=1}^{\mathcal{W}} f_w + \frac{f_{group}}{m}, \tag{10.1}$$

where f_{uv} is a repelling force to avoid physical contact with another agent A_{uv} (Section 10.3.6.1), f_w is a repelling force to avoid physical contact with one of the $\mathcal{W}$ obstacle segments in the environment (Section 10.3.6.2), and f_{group} is a group force to maintain coherent and socially-friendly formations (Section 10.3.6.3).

10.3.6.1 | PHYSICAL-CONTACT FORCE WITH ANOTHER AGENT

The force f_{uv} is applied to agent A_{ij} when there is physical contact with agent A_{uv}. By $dist(A_{ij}, A_{uv})$, we denote the Euclidean distance between A_{ij} and A_{uv}.

Let $n(A_{ij}, A_{uv}) = \frac{x_{ij} - x_{uv}}{dist(A_{ij}, A_{uv})}$ be the unit vector pointing from A_{uv} to A_{ij}. Furthermore, let S be a global parameter that defines the strength of the force. We then define the force as follows:

$$f_{uv} = \begin{cases} S \cdot \max\Big(0, r_{ij} + r_{uv} - dist(A_{ij}, A_{uv})\Big) \cdot n(A_{ij}, A_{uv}) & \text{if } u \neq i \text{ or } v \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

10.3.6.2 | Physical-contact force with obstacles

The force f_w is applied to agent A_{ij} when there is physical contact with one of the obstacle segments w in the environment. By $dist(A_{ij}, w)$, we denote the Euclidean distance between A_{ij} and w. Let $n(A_{ij}, w)$ be a vector that is perpendicular to w and points from w to A_{ij}, normalized to unit length. Furthermore, let S be the global force-strength parameter as described in Section 10.3.6.1. We then define the force f_w as follows:

$$f_w = S \cdot \max\Big(0, r_{ij} - dist(A_{ij}, w)\Big) \cdot n(A_{ij}, w).$$

10.3.6.3 | Group force

The group force f_{group} is defined as $f_{group} = f_{vis} + f_{att}$, where f_{vis} is a deceleration force that represents the desire of A_{ij} to keep its fellow group members in its FOV, and f_{att} is an attractive force that represents the desire of A_{ij} to maintain group coherence.

To define f_{vis}, let $\theta_{ij'}, j \neq j'$, be the minimum rotation angle (in degrees) that is required to let the position $x_{ij'}$ of agent $A_{ij'}$ be inside agent A_{ij}'s FOV. Let $\theta = \max\limits_{1 \leq j' \leq |\mathcal{G}_i|} \theta_{ij'}$ be the maximum of the minimum rotation angles. Furthermore, let S_{vis} be a global parameter that defines the strength of f_{vis}. We then define f_{vis} as follows:

$$f_{vis} = -S_{vis} \cdot \theta \cdot v_{des}$$

This means that we scale the desired velocity v_{des} by the rotation angle θ and the strength parameter S_{vis} in negative direction of v_{des} to compute the first part of the group force.

The force f_{att} describes agent A_{ij} being attracted to the centroid

$$C_i = \frac{1}{|\mathcal{G}_i|} \sum_{1 \le j \le |\mathcal{G}_i|} x_{ij}$$

of the group $\mathcal{G}_i$ (viewed as a given set of points) to maintain group coherence. Let $dist(A_{ij}, C_i)$ be the distance from the agent to the centroid. Similar to Moussaïd et al. [89], we define a threshold distance $d = 0.5 \cdot (|\mathcal{G}_i| - 1)$, such that A_{ij} is attracted to C_i as soon as its distance to C_i exceeds d. Let $n(A_{ij}, C_i)$ be the vector pointing from C_i to A_{ij}, normalized to unit length. Furthermore, let S_{att} be a global parameter that defines the strength of f_{att}. We then define f_{att} as follows:

$$f_{att} = \begin{cases} S_{att} \cdot n(A_{ij}, C_i), & \text{if } dist(A_{ij}, C_i) \ge d \text{ and } v_{des} \neq \mathbf{0} \\ \mathbf{0}, & \text{otherwise} \end{cases}$$

In the above definition, we check whether the desired velocity v_{des} given by the collision avoidance method is **0**. If so, this means that the agent has reached its goal, and we therefore let the attraction force be **0**, too. This yields an overall group force of **0**, and it disables social behavior for agents that have reached their goal.

According to Costa [16], large social groups in real-life tend to split up into smaller sub-groups of up to 3 members. Our SGN method could be adjusted to account for this behavior in the computation of the visual group force f_{vis}. We can split up each group into subgroups of at most 3 members. Instead of computing f_{vis} with respect to all group members, only the members of agent A_{ij}'s sub-group are taken into consideration. All other steps of the method remain unchanged.

10.3.6.4 | Differences to the original model

In the original social-force model by Moussaïd et al. [89], the acceleration term for agent A_i is defined as

$$\frac{dv_i}{dt} = f_i^0 + f_i^{wall} + \sum_j f_{ij} + f_i^{group}.$$

Here, f_i^0 is an attractive force to move agent A_i in a particular direction at a preferred speed, f_i^{wall} is a repulsive force to avoid static obstacles, f_{ij} is a repelling force to avoid physical contact with another agent A_j from a different group. The resulting behavior is reactive and lacks anticipation. To add a more predictive avoidance behavior within our SGN method, we have replaced the above forces by the avoidance forces of Moussaïd et al. [88]; see Equation 10.1 in Section 10.3.6.

Another modification is that we use the centroid C_i of the group $\mathcal{G}_i$ when computing the group force f_{group}. In the original method, the center of mass of the group is used instead of the centroid. We assume that a variation in mass among the group members should not have an effect on the group force, which is why we consider the centroid a better choice.

Similarly, we modified the computation of the force f_{vis}: In the original method, the force is defined via the required rotation angles for each agent to keep the center of mass of the group in its FOV. Instead, we define the force via the required rotation angles to keep the group members themselves in an agent's FOV. Again, we believe that a variation in mass should not have an effect on this step. Furthermore, the original method does not guarantee that group members effectively slow down when a fellow member is left behind in dense situations, which our modification does.

Another change in the force f_{vis} is that we use the desired velocity v_{des} that already takes predictive avoidance behavior into account. In the original method, the actual velocity of an agent is used here, which lacks anticipation.

Finally, we changed the repulsive forces between agents: In the original method, the group force f_i^{group} contains a repulsive term to model the interaction between members of the same group. In our SGN model, we skip this term. Physical contact between agents are generally resolved by our definition of f_{uv} in Equation 10.1, independent of whether the agents are from the same group or not.

10.4 | EXPERIMENTS

10.4.1 | EXPERIMENTAL SETUP

We have validated SGN and compared it against the social-force model by Moussaïd et al. [89], which we have combined with the collision-avoidance method by Moussaïd et al. [88]. The goals of these experiments are threefold: First, we aimed at determining whether SGN yields more coherence and socially-friendly formations than the combination of [89] with [88]. Second, we have used SGN in a room-evacuation scenario, for which ground-truth data of a real-life experiment [80] was available. Third, we have measured the running times of SGN for varying group sized to validate its real-time performance. All three experiments have been performed on a PC with an Intel Core $i7$ 860 processor with 2.8 GHz, an Nvidia GeForce GTX 285 video card and 8 GB of RAM, running Windows 7 Ultimate 64bit. We have used one single core for all experiments.

We integrated the method into the crowd simulation framework described by van Toll et al. [142]. For each agent, we used a radius of 0.24 m, an FOV of $\Phi_{ij} = \pi$

with maximum viewing distance of 10 m. The personal space radius and the social threshold distance were set to 1 m each. Following Weidmann et al. [144], we used a normal distribution with a mean of 1.34 m/s and a standard deviation of 0.26 m/s to randomly choose the preferred speed for each agent. Each goal area was modeled as a disc with radius 0.6 m. The relaxation time τ used in the social-force model was set to 0.5 s. Following Moussaïd et al. [88], the strength parameter S of the physical forces was set to 5000, and the strength parameters S_{vis} and S_{att} were set to 1 and 3, respectively. Furthermore, we set the time for one simulation step to 0.1 s.

We tested our method with group sizes of 2, 3, and 4 in five different scenarios: *bidirectional corridor*, *bottleneck*, *corners*, *building evacuation*, and *room evacuation*. The scenarios are displayed in Figures 10.6 and 10.7.

Bidirectional corridor features a 20 m long corridor that is 10 m wide. Three groups are placed on each end of the corridor, and each group has its goal areas at the opposite end of the corridor. We use this scene to test whether groups stay coherent and in socially-friendly formations when they encounter other groups moving in the opposite direction.

Bottleneck features a 50 m long corridor that linearly decreases in width towards the right side. On the left, the corridor is 40 m wide, and on the right it is 10 m wide. Twelve groups are placed on the left end and have their goal areas on the right end. We use this scene to test whether groups stay coherent and in socially-friendly formations when the environment becomes more narrow and crowd density increases.

Corners features an empty square room with four social groups. Each group is placed near a different corner and has its goal position near the opposite corner of the room. We use this scene to test whether groups stay coherent and in socially-friendly formations when having to cross the center point of a room with other groups approaching from different directions.

Room evacuation features a room with one exit, and a crowd of 180 agents subdivided into groups of varying size. The agents have to evacuate the room through the exit. This experimental setup was proposed by Köster et al. [72]. It is based on a controlled laboratory experiment performed by Liddle et al. [80]. We use this scene to test whether our SGN method generates group behavior that is in line with empirical data, and what effect the group size has on evacuation times.

Building evacuation features a building that spans an area of 95 m $\times$ 128 m. The building has ten rooms that are connected via one large corridor. The corridor has an exit at each end. A total of 490 groups is placed in the rooms, and each group has to leave the building through the nearest exit. The members of a group are all placed at random positions in the same room. We use this scene to test whether groups stay coherent and in socially-friendly formations in a high-density evacuation situation.

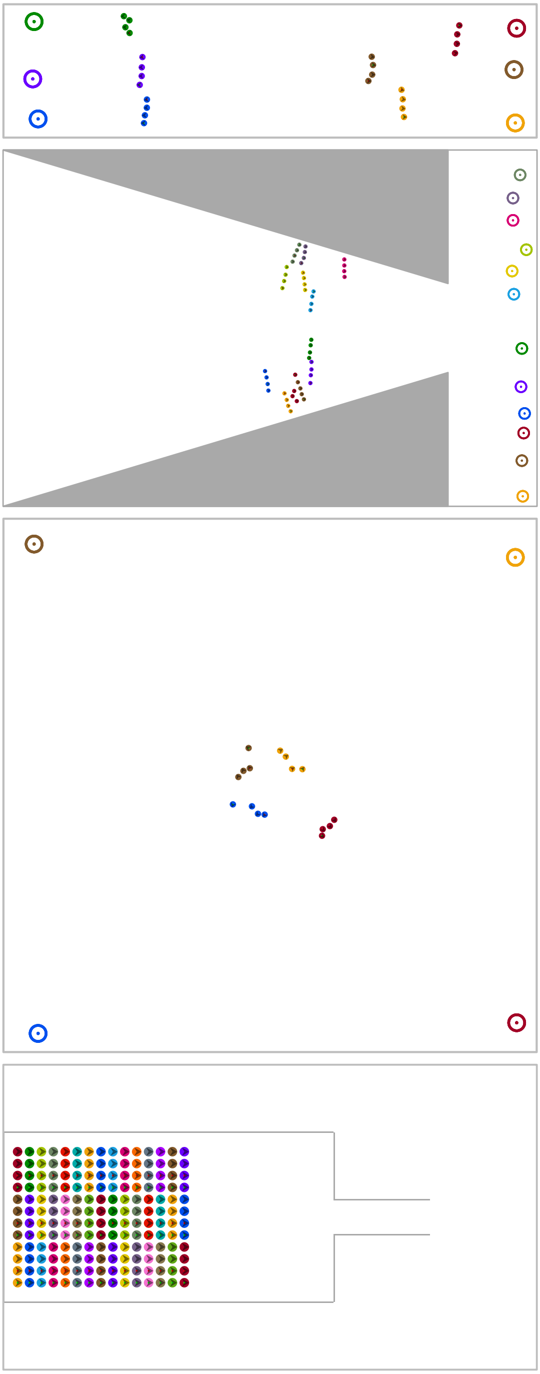


FIGURE 10.6: The first four scenarios we used for our experiments, shown with groups of 4. From top to bottom: *bidirectional corridor*, *bottleneck*, *corners*, and *room evacuation*. Small discs indicate the agents, grouped by color, and large discs indicate the corresponding goal areas.

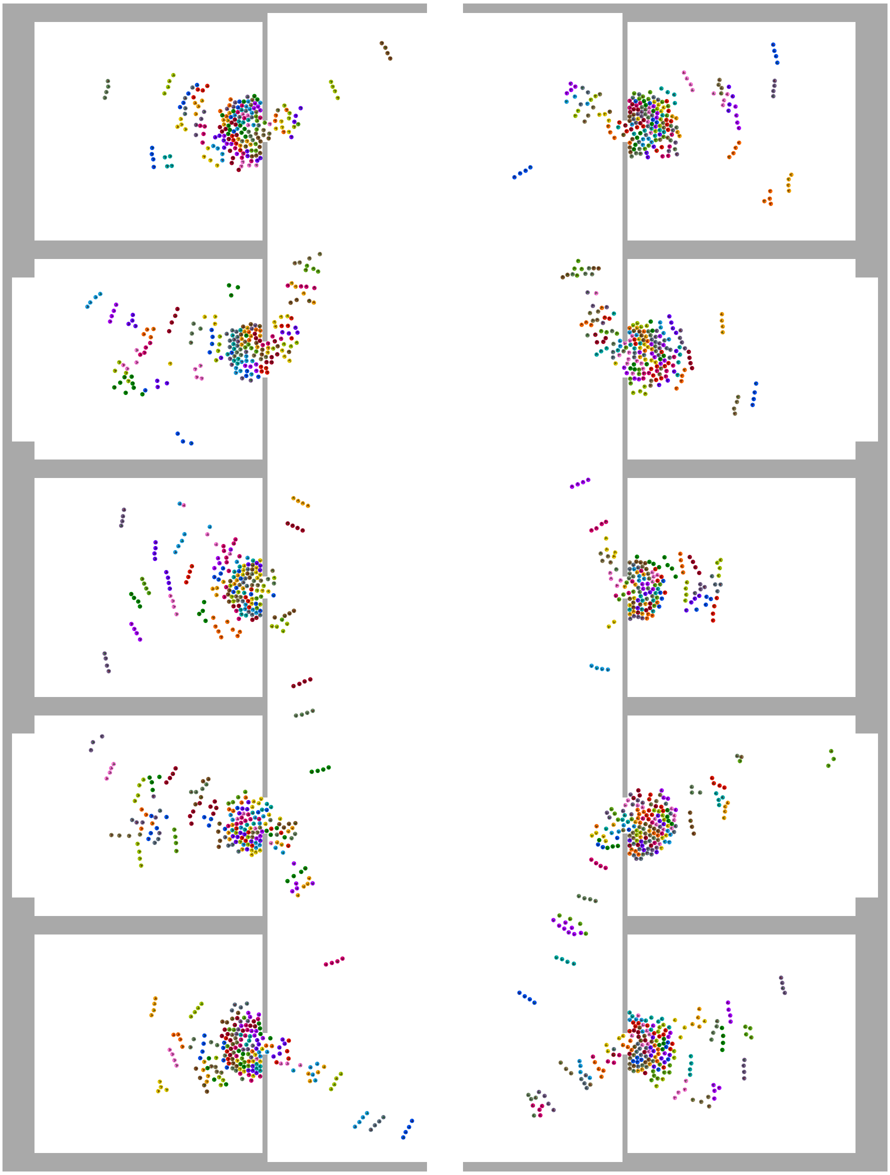


FIGURE 10.7: The largest scenario that we have used for our experiments, shown with groups of 4: *building evacuation.*

10.4.2 | Effects of SGN on coherence and sociality

In a first set of experiments, we compared our SGN method against the methods by Moussaïd et al. [88, 89]. To this end, we integrated both the collision-avoidance method [88] and the social-force model [89] into the *local movement* layer of the framework (Figure 10.3) by van Toll et al. [142].

The goal was to test whether SGN with its additions to the combined work by Moussaïd et al. yields group behavior that better reflects real-life situations than the original methods. An overall assumption is that agents do not switch groups during the simulation. Thus, real-life behavior in a corresponding situation means that each person tries to stay in coherent and socially-friendly formations as much as possible while approaching the goal area. We therefore compared the frequency of coherence and sociality in our simulated groups for the SGN method and for the work by Moussaïd et al. We measured the percentage of simulation steps over the *lifetime* of a group in which it is coherent according to Definition 10.1. By *lifetime*, we refer to the number of simulation steps that it takes a group to reach its goal area. Similarly, we measured the percentage of simulation steps over the lifetime of a group in which it is in a partially-social and totally-social formation according to Definitions 10.2 and 10.3, respectively. We ran each scenario 100 times and took the average coherence and sociality over all runs.

Table 10.1 shows the average coherence (%) and the corresponding standard deviation (%) for both methods and varying group sizes in the first four scenarios. Similarly, tables 10.2 and 10.3 show the average partial and total sociality (%), respectively, and the corresponding standard deviation (%).

The results show that our SGN method improves over the work by Moussaïd et al. in all cases with respect to partial and total sociality. Regarding coherence, our method improves in all cases except the *bottleneck* scenario with groups of 3. In that scenario, coherence is lost in one single run for our SGN method. A Welch's t-test on the difference between the two coherence results (for SGN and Moussaïd et al.) yielded a p-value of 0.3198, and the difference is thus not considered statistically significant.

10.4.3 | Evacuation times

In a second set of experiments, we used the *room evacuation* scenario (Figure 10.6, bottom) to simulate an evacuation scene with a total of 180 agents. First, we measured the evacuation times achieved by our SGN method, by Moussaïd et al. [88, 89] and by Köster et al. [72]. For this experiment, we set the radius of each agent to 0.2 m. All other settings were kept as described in Section 10.4.1. Since the constrained space for this scenario does not allow for much variation in the initial spacial distribution of the groups, we used a fixed initial configuration for the 180 agents. With

Scenario	**Group size**	**Coherence**			
		Average (%)		**StDev (%)**	
		SGN	**Moussaïd et al.**	**SGN**	**Moussaïd et al.**
Bidirectional corridor	2	100	100	0.0	0.0
	3	100	100	0.0	0.0
	4	100	100	0.0	0.0
Bottleneck	2	100	100	0.0	0.0
	3	99.9	**100**	0.2	0.0
	4	100	100	0.0	0.0
Corners	2	100	100	0.0	0.0
	3	100	100	0.0	0.0
	4	100	100	0.0	0.0
Building evacuation	2	**96.0**	92.7	0.4	0.6
	3	**90.6**	84.1	0.8	0.9
	4	**83.3**	75.7	0.9	1.0

Table 10.1: Results from our comparison of SGN with Moussaïd et al. in the first four of our scenarios. We show the average group coherence (%) and corresponding standard deviation (%) for both methods and varying group sizes.

no randomness left, we ran our SGN method once per group size and measured the total time needed to evacuate the room.

Table 10.4 shows the result of this experiment. The corresponding real-life experiment by Liddle et al. [80] was performed with 180 individuals, for which a total evacuation time of 80 s was reported. There is no corresponding ground truth data for bigger group sizes. However, according to empirical data obtained by Xu and Duh [150], the evacuation times should increase when the group size increases. With the method by Moussaïd et al. a decrease in evacuation times can be observed for groups of 4, which contradicts the empirical observations. By contrast, both the method by Köster et al. and our SGN method indeed show this trend.

In addition to the group size, we tested the effect that the radius of an agent's disc has on evacuation time when using SGN. To this end, we repeated the scenario four times with all radii increased, ranging from 0.21 m up to 0.24 m with a step size of 0.01 m. Furthermore, we ran a variant of this scenario with mixed radii that were randomly chosen in the range of 0.20 m to 0.24 m for each agent. Table 10.5 shows the result of these experiments. We conclude that an increase in the radius increases the evacuation times for all group sizes. This is an expected result because higher radii yield less free space in the environment, which increases the overall crowd density. Coordination and re-establishing coherence thus takes more time because

Scenario	**Group size**	**Partial sociality**			
		Average (%)		**StDev (%)**	
		SGN	**Moussaïd et al.**	**SGN**	**Moussaïd et al.**
Bidirectional corridor	2	**91.7**	89.0	2.2	2.6
	3	**77.4**	51.1	5.1	13.2
	4	**64.3**	21.9	5.3	6.7
Bottleneck	2	**92.4**	90.4	1.0	1.2
	3	**82.5**	61.0	3.6	9.6
	4	**72.2**	20.5	4.6	6.0
Corners	2	**91.3**	89.6	2.5	2.7
	3	**80.5**	52.7	4.7	19.0
	4	**70.7**	19.4	4.7	8.0
Building evacuation	2	**54.5**	50.4	0.9	1.0
	3	**31.2**	15.6	0.8	0.7
	4	**20.5**	6.9	0.7	0.4

Table 10.2: Results from our comparison of SGN with Moussaïd et al. in the first four of our scenarios. We show the average group partial sociality (%) and corresponding standard deviation (%) for both methods and varying group sizes.

leaders wait for fellow members only when the local crowd density is low (Section 10.3.3).

10.4.4 | PERFORMANCE

In a final set of experiments, we tested the performance of our SGN method. We used an extended variant of the *room evacuation* scenario, which consists of eleven copies of the scenario, i.e. eleven rooms as displayed in Figure 10.7. Each room is initially occupied by 180 agents, yielding a total of 1980 agents in this stress-test scenario. The agents are subdivided into groups, and each group has to evacuate the room it is starting in. We ran the scenario 100 times for group sizes of 1, 2, 3, 4, and mixed sizes, and we measured the average time needed to compute one simulation step.

Tables 10.6 and 10.7 show the average time per simulation step and standard deviation we achieved for a *serial* and *parallel* execution our method, respectively. For the parallel execution, we used 4 CPU cores and a total of 8 threads. The results show that the average running times are all close to each other for the varying group sizes, and mixed group sizes yield intermediate running times. For all group sizes, our SGN method only yields a small increase in average running times over the simulation of individual agents. When executing the method in parallel, one simulation step is performed about 4.5 times as fast as with a serial execution. For all tested group sizes, we achieved an average rate of slightly less than 20 steps per second. Since we set the time for one simulation step to 0.1 s, we can conclude that

Scenario	**Group size**	**Total sociality**			
		Average (%)		**StDev (%)**	
		SGN	**Moussaïd et al.**	**SGN**	**Moussaïd et al.**
Bidirectional corridor	2	**91.7**	89.0	2.2	2.6
	3	**66.7**	25.1	6.4	10.7
	4	**41.5**	0.3	7.3	1.0
Bottleneck	2	**92.4**	90.4	1.0	1.2
	3	**75.5**	27.2	4.2	10.1
	4	**56.2**	0.6	6.0	1.2
Corners	2	**91.3**	89.6	2.5	2.7
	3	**71.6**	25.4	5.5	15.2
	4	**50.4**	0.2	6.5	1.3
Building evacuation	2	**54.5**	50.5	0.9	1.0
	3	**26.5**	7.2	0.7	0.4
	4	**12.4**	0.5	0.6	0.1

Table 10.3: Results from our comparison of SGN with Moussaïd et al. in the first four of our scenarios. We show the average group total sociality (%) and corresponding standard deviation (%) for both methods and varying group sizes.

Method	**Evacuation time (s)**				
	Group size				
	1	2	3	4	Mixed
SGN	73.50	98.10	112.20	115.70	99.20
Moussaïd et al.	82.10	93.50	95.40	84.30	84.10
Köster et al.	73.63	86.93	-	90.59	-
Liddle et al.	80.0	-	-	-	-

Table 10.4: Evacuation times for the room-evacuation scenario. Due to spacial constraints, for SGN and Moussaïd et al. we used a fixed starting configuration for all agents with no randomness involved. The results show the evacuation times for a single run of each of the two methods. For comparison, we list the mean evacuation times of the method by Köster et al. [72] for individuals and for groups of 2 and 4, and the ground-truth data for individuals as obtained by Liddle et al. [80].

our SGN method achieves real-time performance for large numbers of agents when using parallel computation.

10.5 | LIMITATIONS

While our SGN method generates coherent and socially-friendly group behavior for a large number of agents in real time, it has some limitations. In its current version, SGN does not include avoidance behavior with respect to entire groups. Furthermore, the method does not give the user control over the temporary splitting behavior of a group. Groups may split and re-establish coherence after a successful avoidance maneuver. However, in the actual splitting phase, the group members are treated as individuals.

In addition, the synthetic vision of an agent is still a rough approximation and does not reflect the influence of the environment. For instance, agents in wide open spaces should be able to see hundreds of meters ahead, while their vision should be limited in narrow corridors and indoor environments. The computational complexity of maintaining actual vision during the simulation is still a bottleneck that justifies the usage of a rougher approximation of an agent's vision as used in previous methods, e.g. by Moussaïd et al. [88].

This concludes our chapter on social-group behavior. The SGN method is the last contribution in the form of a novel algorithm that is made in this thesis. In the next chapter, we will present practical implementation details and experimental results of the underlying crowd simulation framework [142], in which the novel algorithms presented in this thesis have been developed.

Agent radius (m)	**Evacuation time (s)**			
	Group size			
	1	2	3	4
0.20	73.5	98.1	112.2	115.7
0.21	82.9	103.6	120.8	129.6
0.22	86.5	113.8	135.9	135.0
0.23	93.8	293.2	145.0	151.6
0.24	98.9	337.7	170.7	251.8
Mixed	86.3	112.5	131.8	142.8

TABLE 10.5: The evacuation times for the *room-evacuation* scenario using our SGN method for different agent radii and group sizes.

	Time per step (msec)		**Frame-rate (#steps/sec)**	
	Average	**StDev**	**Average**	**StDev**
Individuals	228.6	0.7	4.4	0.01
Groups of 2	235.1	0.9	4.3	0.01
Groups of 3	235.6	0.9	4.2	0.01
Groups of 4	236.4	0.7	4.2	0.01
Mixed	233.8	0.9	4.3	0.01

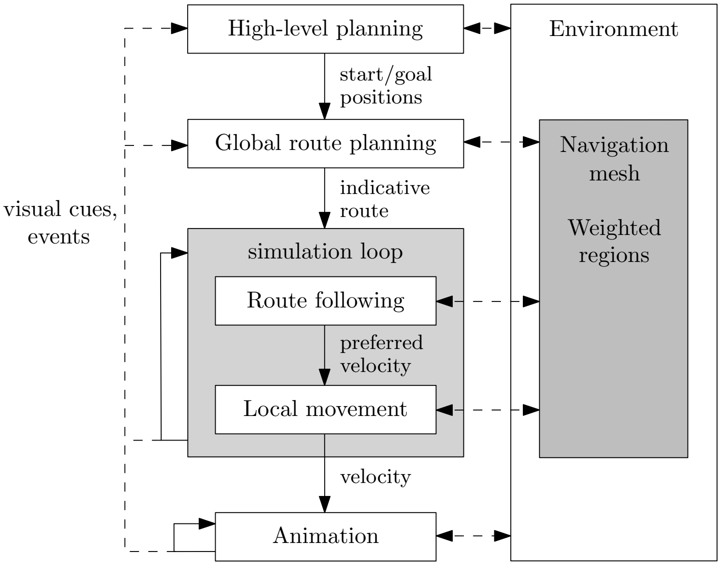
Table 10.6: The average time per step and frame rate we achieved with a *serial* execution of SGN for 1980 agents and different group sizes.

	Time per step (msec)		**Frame-rate (#steps/sec)**	
	Average	**StDev**	**Average**	**StDev**
Individuals	52.4	1.4	19.1	0.5
Groups of 2	56.8	9.2	17.9	1.7
Groups of 3	54.2	1.1	18.5	0.4
Groups of 4	54.1	1.3	18.5	0.4
Mixed	53.3	1.0	18.8	0.4

Table 10.7: The average time per step and frame rate we achieved with a *parallel* execution of SGN on 4 CPU cores and a total of 8 threads for 1980 agents and different group sizes.

CHAPTER 11

COMBINING IT ALL: THE ECM CROWD-SIMULATION FRAMEWORK



All novel algorithms that we have presented in the previous chapters are designed in a modular way. They can be combined to form a larger crowd-simulation framework that follows a five-level hierarchy as discussed in Chapter 1. We have designed all algorithms in a way such they can be used in any framework that follows such a multi-level planning approach. While we have discussed the algorithms from an abstract and general point of view, we omitted particular implementation details of some of the sub-methods whenever these sub-methods made use of a framework-dependent feature. As an example, we omitted the details in Chapters 6 and 7 of how to perform efficient visibility checks between an agent's current position and a candidate attraction point. As long as the underlying framework supports such visibility-checks, the details of how they are performed are not key to the overall idea of a method.

In this chapter, we close this gap by discussing all implementation details of the framework in which the described algorithms have been developed. We start with describing the *Explicit Corridor Map* (ECM) [31] navigation mesh and its properties in Section 11.1. This section expands on the brief discussion of the ECM that we have given in Chapter 2, Section 2.1.3. Subsequently, we present implementation details on the overall framework and its sub-methods in Section 11.2. To show the functionality and efficiency of the ECM framework, we conduct experiments in Section 11.3.

This chapter is based on the following publications:

[52] N. Jaklin, W. van Toll, and R. Geraerts. Way to go – a framework for multi-level planning in games. In *Proceedings of the 3rd International Planning in Games Workshop (ICAPS'13 | PG2013)*, pages 11–14, 2013.

[142] W. van Toll, N. Jaklin, and R. Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN*, 2015.

11.1 | THE EXPLICIT CORRIDOR MAP

We now provide more details on the navigation mesh called the *Explicit Corridor Map* (ECM) [31, 138]. The ECM is an annotated data structure based on the medial axis of the environment, which is the set of all points that are equidistant from at least two distinct closest obstacle points. An example of an ECM in a scene with a U-shaped obstacle polygon and four obstacle-line segments as the boundary of the scene is shown in Figure 11.1.

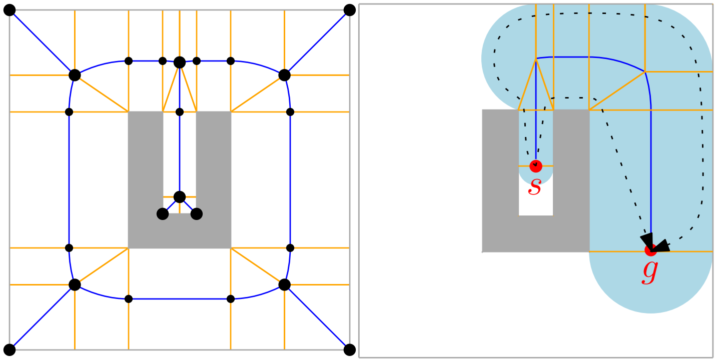


FIGURE 11.1: A simple environment with obstacles (shown in gray). *Left:* The ECM is the medial axis (blue) annotated with closest-obstacle information (orange). This subdivides the walkable space into polygonal regions. *Right:* A path along the medial axis induces a corridor (light blue) due to the ECM's annotations. Within the corridor, we can define any indicative route from s to g; two examples are shown in dotted black.

The medial axis is usually associated with its corresponding medial-axis graph, which can be seen as a special type of waypoint graph. There are two types of vertices in the medial-axis graph: The first type are the points on the medial axis that have three or more nearest obstacles in the environment (the larger black discs in Figure 11.1 (left)). As a degenerate case, such a point can also be located in a non-convex corner of an obstacle, when the boundary of such an obstacle is treated as separate obstacle line segments. The second type of vertices are so-called *event points* at which the medial axis locally changes from a straight-line segment to a parabolic arc (the small black discs in Figure 11.1 (left)). The medial-axis graph is closely related to the *Generalized Voronoi Diagram* (GVD) with the obstacle polygons being the corresponding Voronoi sites [8]. A medial-axis edge consists of a series of event points. Between each pair of consecutive event points, the medial axis is a straight-line segment or a parabolic arc, depending on the type of corresponding obstacles to its left and right (with respect to a given orientation of the edges).

The ECM is based on the medial-axis graph, but in addition, it stores the left and right closest obstacle points for each vertex. When imagining the vertices being connected with their associated closest-obstacle points via straight-line segments (the orange segments in Figure 11.1 (left)), the ECM partitions a $2D$ environment into a set of walkable areas called *ECM cells*. This partitioning can be obtained in $O(n \log n)$ time and with $O(n)$ space, where n is the total number of obstacle vertices of the given scene. Each ECM cell corresponds to one particular obstacle polygon, as all points in that area are closer to that obstacle than to all other obstacles (similar to the relation between a Voronoi cell and its corresponding Voronoi site).

Technically, the ECM can be seen as a variant of the GVD, in which all edges are pruned that the GVD might contain due to treating obstacles as sets of line segments, while adding information about closest obstacles for each event point. The GVD depends on how the given Voronoi sites are defined. For the exact same geometrical scene, different variants of the GVD can be obtained depending on whether obstacle polygons are treated as separate lines or as whole (convex or non-convex) polygonal sites. The medial axis, by contrast, is independent of this choice because any two distinct obstacle points induce a point of the medial axis, no matter what the structure of the overall obstacle is.

The differences between a GVD, a medial axis, and an ECM are only subtle, though, and existing approximation techniques for computing a GVD efficiently using graphics hardware [48] can also be used to compute ECMs: First, for each two-dimensional site (i.e. the obstacle polygons, lines or points) a three-dimensional distance mesh is computed and drawn by the graphics hardware, each mesh in a different color. By projecting the distance meshes back onto the $2D$ plane and tracing the boundary lines of the different regions in the color buffer, a feasible approximation of the GVD can be obtained. This approach requires the obstacle polygons to be convex, so concave polygons are first subdivided into convex ones.

There are software libraries available for computing GVDs: *Vroni* [47] and *Boost*[1]. As a pre-processing step, undesired intersections and overlaps caused by imprecision in the geometry data can be detected and removed using corresponding functions of the Boost library. Vroni or Boost can then be used to robustly compute an ECM.

The ECM has many advantages that make it well-suited for a real-time crowd-navigation framework: It is a sparse graph with only $O(n)$ vertices and edges. Hence, it requires little storage space. As a consequence, global paths can be computed efficiently when using an optimal graph-search strategy such as A^* with an admissible heuristic; see Section 2.2. Furthermore, an ECM can be constructed in $O(n \log n)$ time, and as such, the construction times scale well with increasing numbers of obstacle vertices. Consequently, an ECM can be efficiently computed even for larger or more detailed environments. As discussed in Chapter 2, it represents the exact geometry of the traversable space of an environment. This resolves the issues that are inherent to approximated representations such as grids.

An ECM enables path planning for disc-based agents of arbitrary size, using only a single data structure. This means that agents of different sizes do not need to store their own version of the world geometry that depends on which areas are accessible for them and which are not. Because the ECM stores clearance information, a search method such as A^* can determine in real-time whether an agent is small enough to traverse an edge. Most other navigation meshes artificially inflate the obstacles and work well for only one agent size. Another useful property of an ECM is the fact that the ECM cells are non-overlapping, which makes it well-suited for point-location queries. For any point in the free space, the ECM cell that contains it can be found in $O(\log n)$ time, after which the nearest obstacle can be found in $O(1)$ time. This also allows efficient collision checking with static obstacles. An ECM can be used to compute a variety of indicative routes, et al. routes that stay on the left and right side of the medial axis, or short paths with a preferred amount of clearance [31]; see Figure 11.1 (right). Van Toll et al. [140] have shown that the ECM can also be efficiently updated in response to insertions and deletions of obstacles, such that it allows crowd simulation in dynamic environments. Furthermore, it is well-defined for multi-layered $3D$ environments [138] that consist of multiple connected $2D$ layers. A multi-layered ECM locally has the same properties as the two-dimensional ECM, so many $2D$ algorithms (e.g. dynamic updates, visibility queries, or computing short paths with clearance) also work in the multi-layered ECM. Overall, the ECM is a generic basis for efficient path planning and crowd navigation in simulations and gaming applications.

---

[1] The Boost C++ Library; `http://www.boost.org/` ; accessed January 13, 2016.

11.2 | IMPLEMENTATION DETAILS

In this section, we provide implementation details on the crowd-simulation software framework that is based on the ECM navigation mesh. As discussed in the context of the five-level planning hierarchy that we presented in Section 1.2, the software can be applied to *geometric* planning problems induced by a *semantic* high-level planner. The framework was written in C++ using Visual Studio 2013, but the code has recently been made platform-independent and has been successfully tested on Unix systems, too.

11.2.1 | INPUT AND OUTPUT

The framework supports $2D$ environments and multi-layered $3D$ environments that consist of multiple, usually (but not necessarily) connected $2D$ layers [138]. All environments are encoded as XML files in a format that describes the geometry of each $2D$ layer. Per layer, the geometry can consist of *walkable areas*, which are polygonal regions that form the traversable space of the environment, *obstacles*, which are non-passable polygons, and *openings*, which are (walkable) polygons that can be used to ’cut holes’ into already defined obstacles. The latter concept is useful for creating environments that mainly consist of non-walkable space with only a few walkable corridors, such as a maze. An example environment is shown in Figure 11.2 (left). Layers can also contain *connections*; a connection is a straight-line segment that connects the walkable space of two adjacent layers. Figure 11.2 (right) shows a multi-layered environment with connections. Finally, each layer can contain *weighted regions* as discussed in Parts I and II of this thesis. A weighted region is a simple polygon with a certain type (et al. ‘grass’ or ‘road’) to which each agent can associate an individual weight value. These weight values are stored in separate *agent profiles*, which are blocks within an XML file that describe all agent-based parameters required for running a simulation. These parameters comprise an agent’s radius, the preferred walking speed, the algorithms to use for global planning, path following, and collision avoidance, and the weight values that indicate an agent’s region preferences.

For a given environment, a corresponding ECM can be computed and then saved as an XML file that describes its vertices, edges, and closest-obstacle annotations. Running a simulation in the ECM framework requires such an ECM navigation mesh, a set of agent profiles, and an environment. All results (the environment, the ECM, or the state of the simulation at any point in time) can also be exported to a vector file for the *Ipe* drawing editor[2]. As an optional feature, a simulation can also take an additional scenario file as an input. Such a scenario file is an XML file that describes individual agents or agent groups, their start and goal positions, or

---

[2] The Ipe extensible drawing editor; O. Cheong; `http://ipe.otfried.org/`; accessed January 13, 2016.

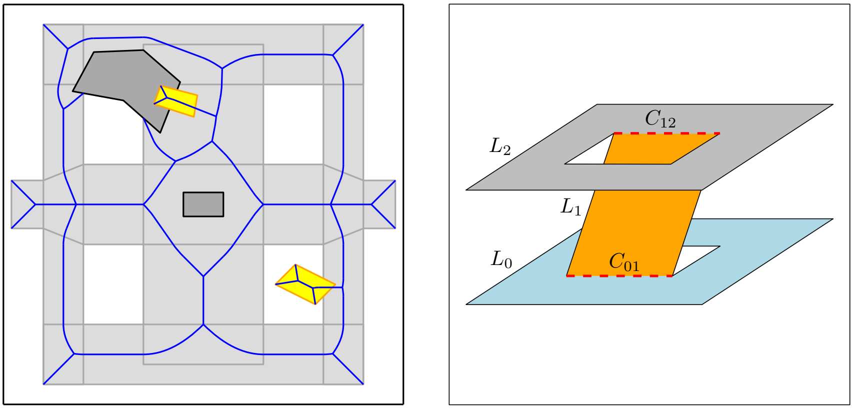


FIGURE 11.2: Constructing an environment. *Left:* Users can define the walkable space of a layer in terms of walkable areas (light gray), obstacles (white and dark gray for two different layers), and openings (yellow). The medial axis of the combined walkable space is shown in blue. *Right:* Multi-layered environments consist of 2D layers L_i and connections C_{ij}. The layers are drawn in different colors for clarity.

rectangular areas in which start and goal positions should be randomly created at the beginning of a simulation. Using such a scenario file with a fixed random seed (which can also be specified within the XML scenario file) allows running a scenario with the same start and goal positions multiple times.

11.2.2 | COMPUTING NAVIGATION MESHES

We now go into more details on how the ECM software framework can compute the ECM of a walkable environment. For a $2D$ environment, we first convert the geometry into a set of disjoint non-walkable polygons, by applying Boolean operations using the OpenGL[3] tessellator.

The resulting polygons are sent to an *ECM generator* of choice. Three such generators are currently implemented within the framework:

1. The first implementation renders an approximated Voronoi diagram on the GPU [48], which is converted to an ECM on the CPU [31]. It requires the user to set the rendering resolution, e.g. at 20 pixels per meter. See Figure 11.3 (left) for an example that was computed with the GPU-based ECM generator.
2. The second implementation uses *Vroni* [47], a library that is widely used in geometry-related research. It computes a topologically correct Voronoi diagram for a set of line segments, which are the boundaries of the obstacle

---

[3] OpenGL; `https://www.opengl.org/`; accessed January 13, 2016.

polygons. From the result, the medial axis can be extracted, and the ECM's closest-obstacle annotations can be added. In a subsequent filtering step, graph components that lie inside the obstacle space are discarded. Figure 11.3 (right) shows an example that was computed with the Vroni-based ECM generator.

3. The third implementation works similarly to the second, but it uses the Voronoi-diagram functionality of the open-source *Boost* library.

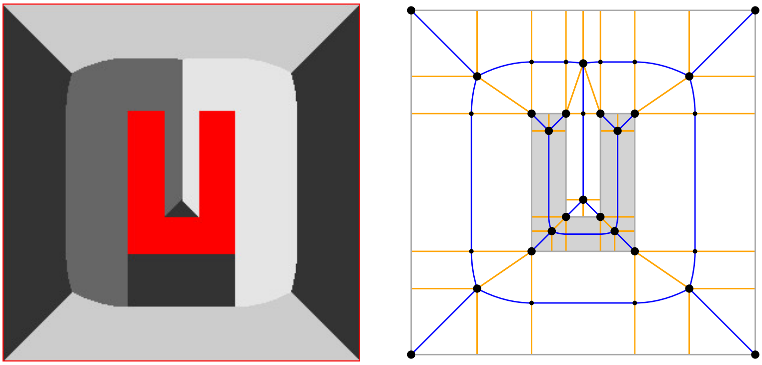


FIGURE 11.3: Two ways to generate the ECM. *Left:* Using rendering techniques, we can approximate the Voronoi diagram on the GPU frame buffer. *Right:* External libraries such as Boost and Vroni can compute the Voronoi diagram, which we convert to an ECM. Graph elements inside obstacles are filtered out in a post-processing step.

Boost and Vroni have the advantage that they do not depend on a resolution parameter. However, since these exact methods respond to imprecision in the input, their resulting ECMs may contain undesired details. Additional filtering steps (e.g. filling small openings, merging line segments that are close together, and resolving intersections between segments) are needed to handle such imprecision for arbitrary environments.

For *multi-layered 3D* environments, an iterative algorithm is used that first computes the $2D$ ECM for each layer and then stitches these ECMs together to obtain a continuous multi-layered navigation mesh [138]. The construction algorithm consists only of a sequence of $2D$ steps, so any of the above mentioned generators can be used. The ECM generator can either write the navigation-mesh data to an XML file or keep it in memory. This way, the user can run a simulation from the ECM in memory as an alternative to loading an external XML file.

11.2.3 | ALGORITHMS OF THE PLANNING HIERARCHY

Within the context of the five-level planning hierarchy as described in Section 1.2, the ECM framework contains implementations for each of the center levels: global-route planning, route following, and local movement. The following algorithms have been implemented:

For the global-route planning level, the framework contains an implementation of A^* [42] to compute shortest paths on the medial axis. The resulting path can be converted to various types of indicative routes: short paths with a preferred amount of clearance, or paths with side preference (e.g. for staying on the left and right side of the free space). It also contains a re-planning algorithm that recomputes a medial axis path efficiently after an obstacle has been inserted or deleted [139]. When the environment features weighted regions, agents can perform an A^* search on a weighted grid instead. In addition, an implementation of the VBP method as described in Chapter 4 is available.

For the route-following level, our framework includes the IRM [63] and both versions of the MIRAN method as described in Chapters 6 and 7. At the local-movement level, the framework includes implementations of the velocity-based collision-avoidance algorithms by Moussaïd et al. [88] and Karamouzas et al. [64], as well as the *Stream* model as described in Chapter 9 for improved coordination between agents at high crowd densities. In addition, it includes the popular ORCA collision-avoidance library by van den Berg et al. [133], which is publicly available online[4].

The framework also includes an implementation of the SGN method as described in Chapter 10. The SGN method is not contained in a single layer of the planning hierarchy, but it affects global-route planning, route-following, and local movement.

11.2.4 | ARCHITECTURE

We will now highlight a number of details concerning the architecture of the ECM software. This discussion focuses on aspects that emphasize the modularity of the framework and its efficiency.

Modularity: For each of the three geometric levels in the hierarchy (route planning, route following, local movement), any algorithm can be plugged in as long as it implements the required abstract methods. For instance, all route-following implementations should compute a preferred velocity for a given agent, but the programmer can decide on the internal details. The framework uses the *factory* design pattern so that new implementations can easily be added. Users can assign any combination of algorithms to an agent (e.g. short global paths with clearance, MIRAN for path following, and ORCA for collision avoidance) using a settings file.

---

[4] `http://gamma.cs.unc.edu/ORCA/`; accessed January 13, 2016.

A similar architecture is used for the ECM generators as described in Section 11.2.2, so that users can easily switch between implementations.

Sequence of simulation loops. Instead of performing all computations at once for each agent, one simulation step is subdivided into multiple loops. We first compute the preferred velocity of an agent. Next, we compute an agent's actual velocity. Finally, we update an agents' position. This ensures that the correctness of the used methods does not depend on the order in which agents are being stored. The first agent in the ordering uses the exact same information as the last agent, and the result is deterministic.

Multi-threading. Each of the loops in a simulation step can easily be parallelized because an individual agent only needs to *read* properties of the environment or neighboring agents. Hence, the calculations for different agents are completely independent. The framework uses basic OpenMP[5] instructions to automatically divide multiple agents over multiple threads, without having to lock parts of the code to prevent conflicts and deadlocks between threads.

API / Library. The framework has also been built as a Windows library (DLL) with a number of basic API functions such as loading an environment, computing the ECM, or adding an agent. The API function that performs a single simulation step fills an array of wrapper objects (C structs) that contain the new positions and orientations of each agent, and it returns a pointer to this array. If an external program is linked to the DLL and defines the exact same wrapper object, both programs can share the array. Using this technique, we have linked our DLL to the Unity3D game engine[6] to simulate moving crowds in $3D$. Examples of such a $3D$ visualization can be seen in the videos for our Stream model[7] as described in Chapter 9 and for the SGN method[8] as described in Chapter 10. The Pedestrian Dynamics crowd analysis software[9] uses our framework in a similar fashion.

11.2.5 | Visibility checks

Some of the novel methods (MIRAN, Stream, and SGN) that we have presented in this thesis require efficient visibility checks between two points in a virtual environment. For all methods that assume an agent to be represented as a disk, these checks also require that an agent's radius is taken into account. We have argued that the methods themselves do not depend on a particular data structure as long as the given data structure supports such checks. We now discuss how visibility

---

[5] OpenMP; `http://openmp.org/`; accessed January 13, 2016.

[6] Unity3D; `http://www.unity3d.com/`; accessed January 13, 2016.

[7] `https://youtu.be/XSusPwT81pI` ; accessed January 13, 2016.

[8] `https://youtu.be/nuGyOLW_6eE` ; accessed January 13, 2016.

[9] Pedestrian Dynamics; InControl Simulation Solutions;
`http://www.pedestrian-dynamics.com/`; accessed January 13, 2016.

checks are implemented in the ECM framework, which we have omitted when describing the methods in their corresponding chapters. Throughout this section, we let n be the total number of obstacle vertices in the environment.

For a given query point p, we would like to know whether point p is visible from an agent's current position x. There are two types of *visibility* checks that we want to perform: The first type is point-point visibility, for which only the straight-line segment between x and p needs to be fully contained in the traversable space of the environment. The second type is disk-point visibility, for which the entire capsule shape (see Chapter 7), which is induced by a sliding disk from x to p, needs to be fully contained in the traversable space of the environment. Both types of visibility checks are performed in the same way, where the first type can be seen as a special case of the second type with a disc radius of 0 around x.

The first step for determining the visibility of p from x is to perform a *point-location query* for x. This point-location query yields the correct ECM cell in which x is being contained. To answer such point-location queries, a regular grid is initially computed and overlain after the ECM has been constructed. Each cell in this grid then stores pointers to all ECM cells that intersect the grid cell. To this end, we iterate over the $O(n)$ many ECM cells and add a pointer to an ECM cell from each grid cell that is being intersected by it. This can be done in $O(1)$ time per intersected grid cell, so the time needed for this step depends on the overall number of intersected grid cells, which again depends on the resolution of the grid. With this point-location grid, we can then find the correct ECM cell in which x is contained in $O(k)$ time, where k is the number of ECM cells that intersect a grid cell.

The second step for determining the visibility of p from x is to trace the ECM structure and traverse all ECM cells that are intersected by the straight-line segment between x and p. We start with the ECM cell that contains x, which we determine using the point-location grid in the first step. Since each ECM cell has only constant complexity [31], we can find in $O(1)$ time the point q where the line segment between x and p intersects the boundary of the current ECM cell, if such a q exists. If not, we know that p is in the same cell as x. For a point-based agent with no radius, we can directly conclude that p and x are mutually visible. If q does exist, we do the following: Let o be the closest obstacle, which induces the current ECM cell. We then have to compute the point s on the sub-segment between x and q that minimizes the distance to (the boundary of) o. We then have to check whether the shortest distance from s to o is smaller than the agent's radius or not. For the special case of a point-based agent with no radius, it suffices to check whether the point q itself is an obstacle point or not. If the distance between s and o is bigger than the agent's radius, we know that the agent does not intersect any obstacles when moving from x to q. We can then iterate the previous steps inside the next adjacent ECM cell, for which q takes over the role of x. When this process does not terminate due to an agent being too large to fit through a particular ECM cell, we finally find the ECM cell that contains p. We then have to do a final check whether the distance between p and its closest obstacle o is smaller than the agent's radius or not. This last check is needed to determine whether the agent would intersect o if it were located on p (and consequently p would not be visible with respect to disk-point visibility).

Overall, with a point-location grid as described above, a visibility check in the ECM structure can be performed in $O(k)$ time, where k is the number of ECM cells that are intersected by the straight-line segment between the query points. In theory, there can be $O(n)$ many intersected ECM cells, when the environment is one long corridor with query points at the opposite ends. In practice, however, the number of intersected ECM cells is usually small compared to the total number of ECM cells.

11.3 | EXPERIMENTS

In this section, we demonstrate the capabilities of the ECM software using two large environments: *City*, a $2D$ footprint of a virtual city, and *Station*, a multi-layered model of a train station in the Netherlands. Numerical data on the complexity of the environments and their ECMs can be found in Table 11.1. Both environments and their ECMs are visualized in Figure 11.4. All experiments were performed on a Windows 7 PC with a 3.20 GHz Intel $i7-3930K$ CPU, an NVIDIA GeForce GTX 680 GPU, and 16 GB of RAM. In general, only one thread was used, except in the final experiment which shows the benefit of multi-threading.

Environment	Geometry			ECM		
	O	V	Size (m)	V	E	A
City	184	2098	500×500	1444	1623	6310
Station	568	1800	153×111	660	768	2804

TABLE 11.1: The *Geometry* columns show the number of obstacles (O), their combined number of vertices (V), and the width and height of the environment (in meters). The *ECM* columns show the complexity of the ECM: the number of vertices (V), edges (E), and annotations (A) as the number of points with closest-obstacle information.

Environment	Constr. time (ms)		
	Vroni	Boost	GPU
City	84 [1.3]	141 [1.7]	554 [4.6]
Station	368 [4.4]	381 [5.2]	1266 [8.1]

TABLE 11.2: The construction times for the ECM using all three implementations. Running times were averaged over 10 runs each, and the standard deviations are shown in square brackets.

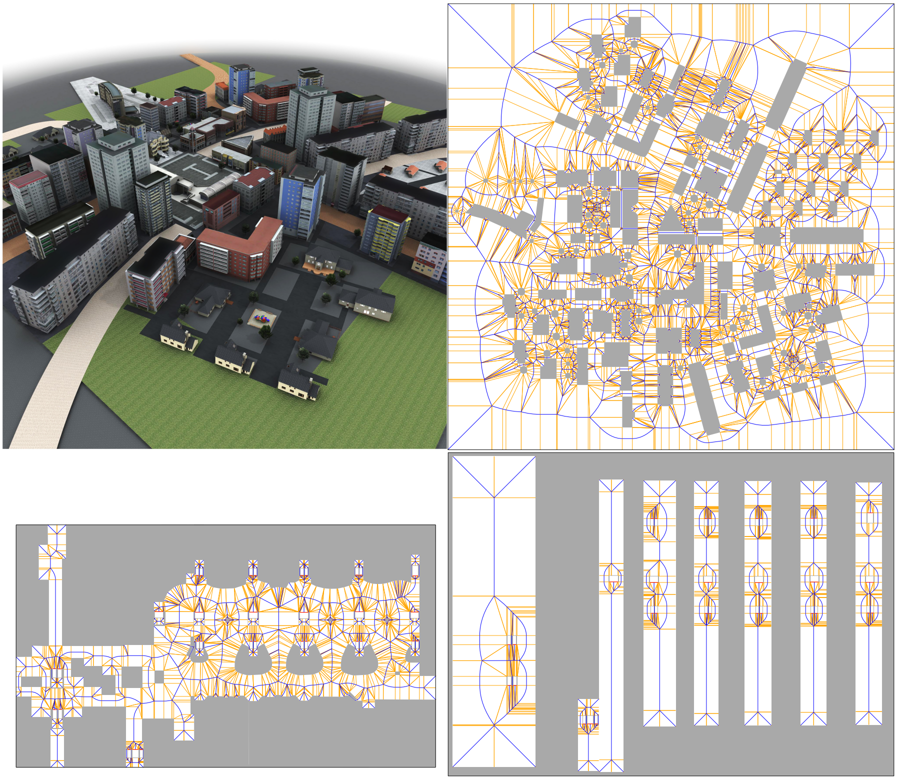


FIGURE 11.4: The two environments used in our experiments. *Top:* The city environment. *Bottom:* The station environment. The medial axis is shown in blue; closest-point annotations are shown in orange. For the *Station* environment, only the two main layers are shown. Layer connections are displayed in red.

We have computed the ECM for both environments using all three ECM-generator implementations: the Vroni-based generator, the Boost-based generator, and the GPU-based generator. For the GPU-based method, we used a resolution of 4000×4000 pixels. All computations were repeated 10 times. The results are shown Table 11.2. This table shows that the Vroni-based implementation is the fastest: on average, it computes the ECM of *City* in 84 ms, and the ECM of *Station* in 368 ms. Hence, even for large multi-layered environments, the ECM is generated well within a second, which allows our framework to be integrated into a modeling tool with interactive feedback.

Next, we have dynamically inserted 100 obstacles into both environments at random free positions. After an insertion, the ECM is updated along with its point-location grid structure (see Section 11.2.5), such that agents are able to use the updated navigation mesh in subsequent simulation steps. For simplicity, we only added square obstacles measuring 2×2 meters. Note, however, that the insertion

Environment	Dynamic insertions (ms)	Visibility (ms)	Indicative routes (ms)
City	3.70 [0.38]	0.15 [0.06]	1.17 [0.70]
Station	1.25 [0.21]	0.10 [0.07]	0.85 [0.47]

FIGURE 11.5: Results of three experiments. Standard deviations are shown in square brackets. The *Dynamic insertions* column displays the time to dynamically insert a square obstacle into the ECM, averaged over 100 obstacles. The *Visibility* column displays the time to compute a visibility polygon, averaged over $1,000$ random positions. The *Indicative routes* column denotes the time to compute a short indicative route with clearance, averaged over $1,000$ pairs of random start and goal positions.

algorithm supports any convex polygon that does not intersect existing geometry [140]. Figure 11.7 (top-left) shows the obstacles and the resulting ECM for *City*. The insertion times (for updating both the ECM and the point-location grid structure) are shown in Table 11.5. On average, an insertion took 3.70 ms in *City* and 1.25 ms in *Station*. The running times in *City* are higher because this environment is more complex in the sense that a dynamic update affects more ECM cells on average. These results indicate that the ECM can efficiently model dynamically changing environments, e.g. with bridges that may collapse, or parked vehicles that temporarily block roads. More experiments on the dynamic ECM have been conducted by van Toll et al. [140].

11.3.1 | COMPUTING VISIBILITY POLYGONS

In addition to point-point and disk-point visibility checks as described in Section 11.2.5, the ECM software can also compute the $2D$ *visibility polygon* $V(p)$ for any given point p in (traversable space of the) environment. In this context, we assume that all surfaces are flat and the only occluding features are the obstacle polygons. The method also works in multi-layered environments, where the visibility polygon may cover multiple layers. Note that this does not correspond to actual $3D$ visibility, but it can be rather seen as $2.5D$ visibility, which is useful for crowd simulation of $2D$ surfaces (e.g. for letting agents respond to visual input in multi-layered environments).

The visibility polygon can be used to model what agents can visually perceive, e.g. to let them respond to an event in the environment when they see it. To this end – instead of keeping track of the dynamically changing visibility polygon for each agent – we can compute the visibility polygon $V(a)$ for a particular and stationary area a (e.g. for the center point of a newly inserted obstacle that blocks a previously open passage) and keep track of when an agent enters $V(a)$ during the simulation. Similar to the visibility checks for single query points as described in Section 11.2.5, the visibility polygon is computed by traversing the ECM cells in an ordered manner. As such, the running time depends on the number of cells that $V(p)$ intersects.

We computed the visibility polygon for $1,000$ random query points in both environments. On average, the algorithm took 0.15 ms in the *City* environment and 0.10 ms in the *Station* environment. We conclude that the ECM framework can easily answer visibility queries for a large number of agents in real-time. Figure 11.7 (top-right) shows a number of visibility polygons in the *City* environment.

11.3.2 | Computing indicative routes

We have computed global indicative routes for $1,000$ pairs of random start and goal positions per environment. To compute such a route, we first performed an A^* search [42] on the ECM graph structure, which yields a shortest path along the medial axis. We then extract an indicative route through the corridor around this path. In Section 11.1, we stated that there are multiple options for computing an indicative route through a corridor, such as a shortest route with clearance, or a route with a side preference with respect to the medial axis. Since these options have comparable complexity, showing only one option is sufficient for giving a general indication of global-planning times. For this experiment, we have used the shortest-route option [31] with a preferred clearance of 0.5 m.

On average, global planning takes 1.17 ms in the *City* environment and 0.85 ms in the *Station* environment. Examples of indicative routes in the *City* environment are shown in Figure 11.7 (bottom-left).

11.3.3 | Crowd simulation

To show the efficiency of our crowd simulation software, we have generated increasingly large crowds of agents in our environments. Figure 11.7 (bottom-right) shows a crowd in the *City* environment. We measured the running time of each simulation step as long as all agents were still traversing a path, i.e. up to and including the step in which the first agent reached its goal. All agents received random start and goal positions, with a minimum $2D$ Euclidean distance of 50 meters between start and goal to ensure a miminum traversal time for each agent. Note that we excluded the time required for computing the indicative routes because this aspect was already covered in the previous experiment.

We ran all simulations using fixed timesteps of 0.1 seconds, which is a common value in comparable crowd simulation software [63]. Thus, whenever the simulation steps take at most 100 ms to compute, we say that the simulation runs in real time. In line with real-life measurements [144], we used an agent radius of 0.24 meters and a preferred walking speed of 1.4 m/s. For path following, we used the

MIRAN method as described in Chapter 6 with the modifications for computing candidate attraction points as described in Chapter 7. The MIRAN parameters we used were a sampling distance of 1 m and a shortcut parameter of 5 m. We used the vision-based collision algorithm by Moussaïd et al. [88] because – similar to our experiments for the Stream model in Chapter 9 – it yielded the best results in most of the tested scenarios. However, since collision avoidance is inherently the most expensive phase (because it requires agents to compute nearest-neighbor information on the fly), we have run this experiment both with and without collision avoidance.

In addition, we have run all simulations in the *City* environment with and without *multi-threading*. As described in Section 11.2.4, a step of the simulation loop consists of multiple substeps: computing the preferred velocities of all agents, computing the actual velocities to avoid collisions, and updating all positions using these velocities. We used OpenMP to divide the workload of each substep over a total of 8 threads. Note that the three substeps are still executed in sequence to maintain consistency among agents.

When collision avoidance is disabled, the running time of a simulation step scales linearly with the number of agents, as indicated by the red line in Figure 11.6 (top). For example, with $100,000$ agents, a step took 82 ms on average. With *one million* agents, it is worth noting that the framework used 2.3 GB of memory in this scenario, which is a small memory footprint considering the size of the crowd. A crowd of this size cannot be simulated in real-time yet, but multi-threading techniques and hardware improvements will make this possible in the future. Using 8 parallel threads greatly improves the running times, as indicated by the green line in Figure 11.6 (top). The simulation does not become 8 times as fast, because it is not possible to parallelize the entire simulation step, i.e. the three subtasks need to be performed sequentially. Still, we achieved a speed-up factor of 3 to 4 for large crowds. For instance, simulating $200,000$ agents took 50 ms per step on average. Future work will show how the number of threads influences the improvement ratio.

Figure 11.6 (bottom) shows the running times for the same scenario with collision avoidance turned on. Note that this figure has a different scale than Figure 11.6 (top) on both axes. The reason is that we could not model the largest numbers of agents because the environment became too congested. Simulating up to a million agents is only feasible in a much larger environment. A multi-threaded simulation with $10,000$ agents requires 63 ms per step on average. The computation time appears to scale quadratically with the number of agents. This is most likely due to the grid used for nearest-neighbor queries, which has cells of a fixed size. If large clusters of agents appear in one cell, agents need to evaluate many potential neighbors. A data structure such as a kd-tree could overcome this problem, although such a structure would need to be rebuilt in each simulation step. We leave a comparison of these data structures for future work.

In conclusion, the ECM framework can simulate tens of thousands of agents in real-time, including collision avoidance. Many simulations will include more steps than the ones we measured, such as global (re-)planning and high-level planning. Also, if real-time visualization is desired (such as in a game or an interactive simulation), less time is available for the simulation itself. The details depend on the application at hand.

11.4 | Conclusion and future work

In this chapter, we have presented the Explicit Corridor Map (ECM) framework as an implementation of the three center levels of the five-level planning hiearchy as described in Chapter 1. These three center levels comprise the *geometric* aspects of navigation: global route planning, route following, and local movement. The ECM is a *navigation mesh* that has many advantages over classical graph or grid representations. For instance, it enables fast and flexible planning of global paths due to an underlying sparse graph structure. Furthermore, it supports disk-based agents of all sizes using only one data structure, and it allows fast collision checking with obstacles due to its cell decomposition. Experiments show that the ECM software allows many operations at interactive rates, and that it can simulate large crowds in complex $2D$ and multi-layered environments in real-time.

For future work, one possible extension of the framework is to take agents of different heights into account. For instance, big vehicles may not fit through small tunnels that regular agents can use. Another challenging research topic is the *validation* of crowd simulation models, i.e. measuring how closely the simulated behavior resembles real-life behavior. Validation software for the *steering* aspects of a crowd simulation does exist [120], and the *Menge* framework by Curtis et al. [18] can be seen as a first attempt to provide a unified framework to validate crowd-simulation models. However, the question of how to compare higher-level aspects of simulated crowds to real-life crowds is still largely open.

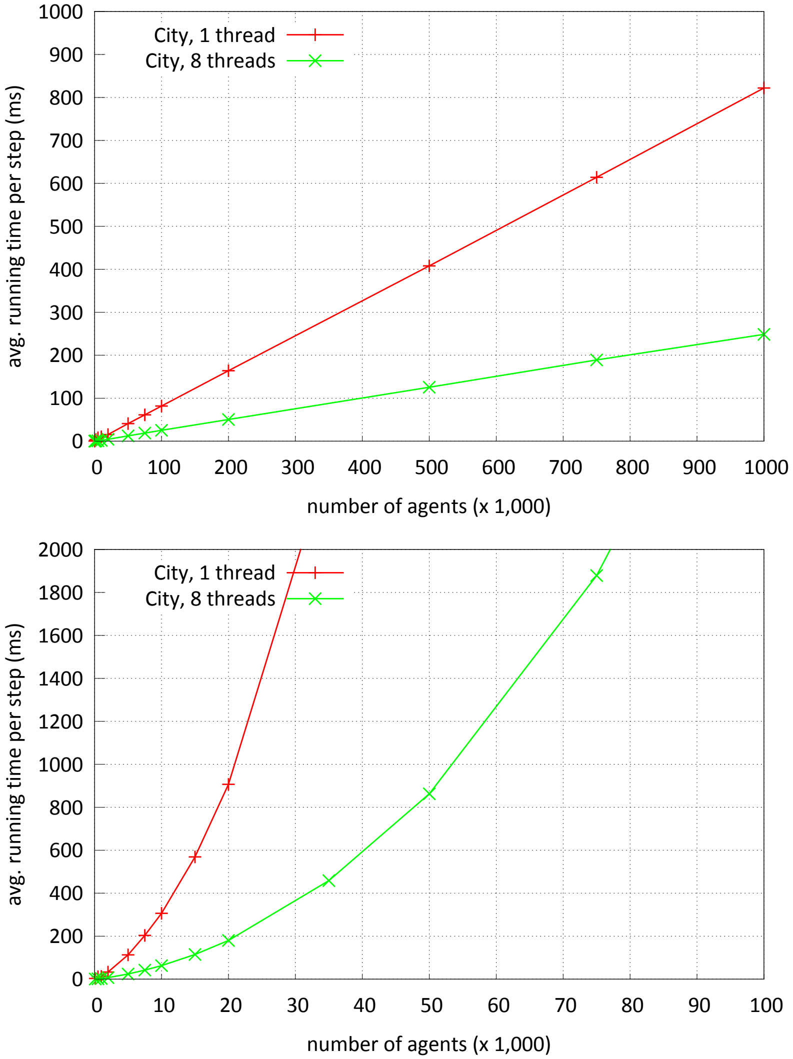


FIGURE 11.6: Running times of crowd simulations in the *City* environment. The horizontal axis shows the number of agents ($\times 1,000$); the vertical axis shows the average running time of a simulation step, which models 100 milliseconds of simulation time. *Top:* Without collision avoidance, the running time is proportional to the number of agents. Multi-threading improves the results by a factor of 3 to 4. *Bottom:* With collision avoidance, the running time increases at a higher rate.

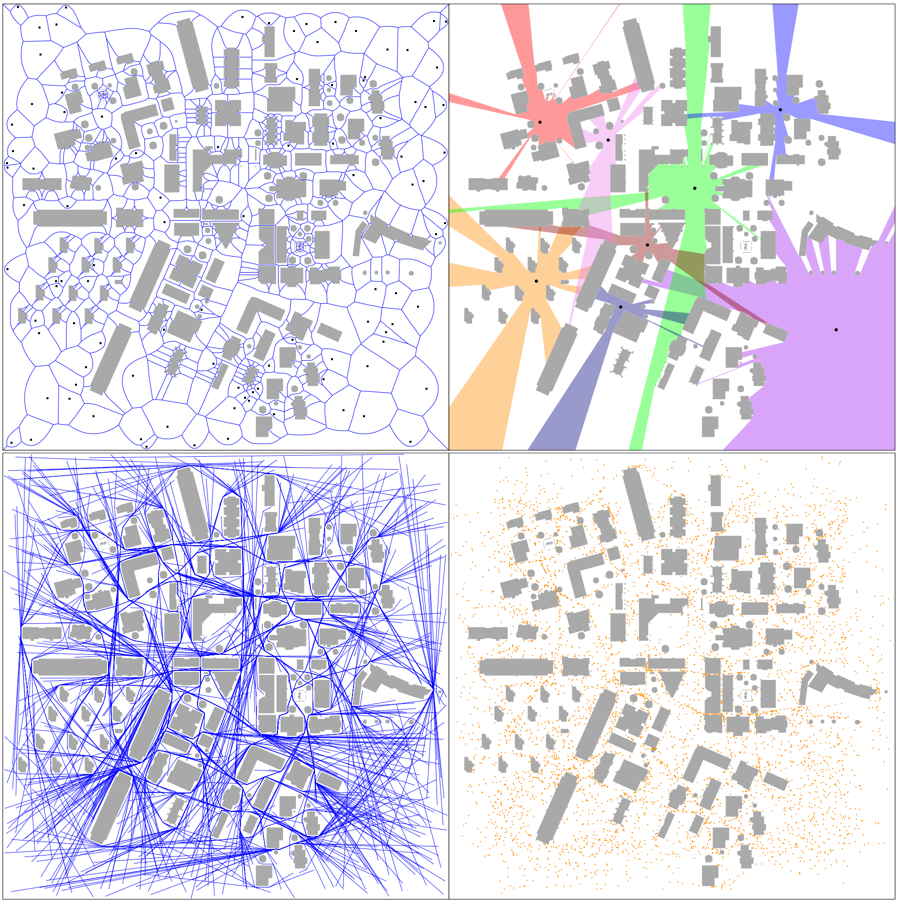
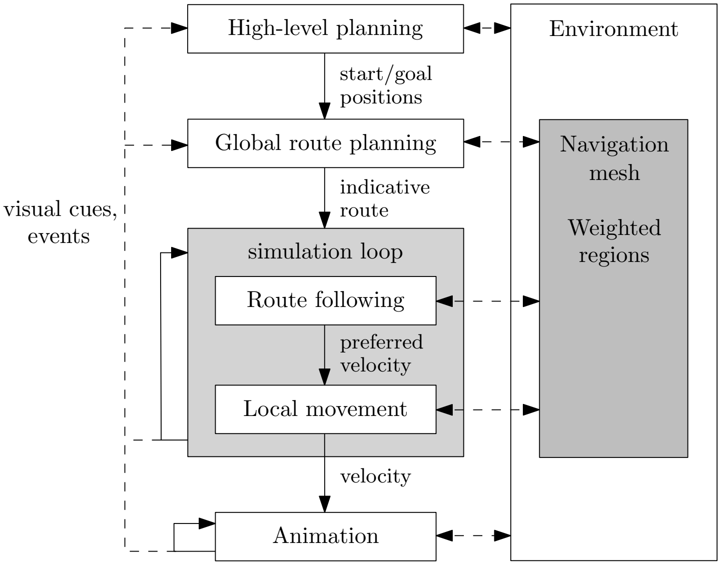


FIGURE 11.7: Experiments in the *City* environment. *Top-Left:* We have dynamically inserted 100 square obstacles (shown in black) at random positions. The updated medial axis is shown in blue. *Top-Right:* Visibility polygons. Query points are shown in black; their visibility polygons are shown in different colors. *Bottom-Left:* Examples of 500 indicative routes (shown in blue) between random start and goal positions, computed by performing A* on the ECM and computing a short route within the resulting corridor. *Bottom-Right:* A crowd of $5,000$ agents, shown as orange disks. Agents have been enlarged for illustrative purposes.

Chapter 12

CONCLUSION PART III



This chapter concludes the third part of this thesis: crowd simulation. We have discussed the problem of coordinating virtual crowds in high-density scenarios, as well as maintaining socially-friendly formations and the coherence of small social groups.

In Chapter 9, we have presented *Stream*, a novel crowd-simulation model that combines the advantages of agent-based and flow-based paradigms while relying only on local information. The core idea of Stream is to let each agent perceive an average *stream velocity* of the perceived velocities of neighboring agents in its vicinity. We then compute a new velocity for that agent, which is an interpolation of the agent's currently preferred velocity (under the assumption that no other agents are present) and the perceived stream velocity. This interpolation is based on an agent's *incentive*, which is a dynamically changing value that models an agent's willingness to comply with the local crowd flow. The incentive is based on factors such as local crowd density, deviation from the agent's currently preferred direction, the travel time spent so far, and a base-incentive value called *internal motivation*.

As we have shown in our experiments, the Stream model reduces the occurrence of deadlock situations in high-density scenarios such as a narrow hallway with a large number of agents. In such scenarios, the model improves coordination among agents and yields the formation of lanes. This usually comes at the cost of slightly increased travel times and travel distances because coordinating agents do not always follow the shortest path to their goals. Another contribution of Stream, which we have demonstrated in a corresponding video[1], is that internal motivation is a fixed parameter that can be set to simulate various behavioral profiles. Such profiles can be a strolling person in a shopping mall, who is not in a hurry and thus willing to comply with the local crowd flow. Another example is a policeman in a riot scenario, who might need to push through a crowd that is moving in opposite direction.

In Chapter 10, we have presented *Social Groups and Navigation* (SGN), a method that enables the simulation of small pedestrian groups. SGN is based on the social-force model by Moussaïd et al. [89], which we have modified and extended to generate more socially-friendly and more coherent group behavior. To this end, we define two quantitative metrics: *coherence* and *sociality*. In addition, SGN incorporates social-group behavior on the global-planning level by letting a group follow a shared global path, and by letting agents wait for each other when coherence is lost during the simulation. This waiting behavior to re-establish a group's coherence is based on the local crowd density around the waiting members: Only when crowd density is low, group members will wait for their fellow members to catch up with them.

We have shown experimentally that SGN yields more coherent and socially-friendly group behavior than a combination of the social-force model by Moussaïd et al. [89] with the collision-avoidance method by Moussaïd et al. [88]. Furthermore, we have compared SGN with existing ground-truth data of a real-world evacuation experiment [80]. We concluded that SGN is in line with the real-world observation [150] that evacuation times increase when the group sizes increase. Lastly, we have shown that SGN yields only a small increase in average running times over the simulation of individual agents. When executing the method in parallel on 4 CPU cores and a total of 8 threads, one simulation step was performed about 4.5 times as fast as with a serial execution. As a consequence, a parallel execution allows the simulation of a few thousand agents at interactive rates on current hardware.

The work that we have presented in Part III has its limitations, which give rise to interesting future-research questions. We have shown that Stream reduces the occurrence of deadlocks in narrow-hallway environments and similar areas within a virtual scene (such as the virtual university that we show in the corresponding video). However, Stream still does not fully resolve problems that are caused by the global-route planning step of the hierarchy. Examples of such problems are crowd congestion near obstacle corners when a large number of agents is trying to

---

[1] `https://youtu.be/XSusPwT81pI` (accessed January 13, 2016)

follow a shortest path around obstacle polygons, or agents that start in a circular arrangement and try to walk to the opposite position on the circle. The common problem in both these examples is that large numbers of agents try to follow global paths that all intersect in a particular point or small area at the same point in time during the simulation. Overcoming these problems is therefore an open question related to global path planning rather than local-movement models such as Stream.

Regarding social-group behavior, it would be interesting to further validate our SGN method by comparing it against more social-group methods. It would also be interesting to collect and use more ground-truth data from real-life experiments and see how SGN performs in this regard. Such ground-truth data could also be used to derive more behavioral features from the real world, similar to the waiting behavior we have already incorporated in the method. Another conceptual future extension could be to allow larger groups and incorporate a recursive approach for simulating dynamically changing subgroups that can merge and split.

CHAPTER 13

OVERALL CONCLUSION AND FUTURE RESEARCH

All work that we have presented in this thesis contributes to one single long-term goal: The realistic simulation of the navigational aspects of autonomous virtual humans, both for individual agents and large crowds. A general and mathematically exact formulation of what we mean by 'realistic' cannot be given because it heavily depends on the context. Formulating it within a particular context will always be a simplification of real-world behavior and omit factors that are less relevant for a given application. As such, the novel algorithms that we have presented each pick a particular aspect of real-world human navigation and aim at letting these aspects emerge from a set of geometric rules and properties. In the remainder of this conclusion chapter, we discuss the overall limitations of the contributions made in this thesis and of current approaches in crowd-simulation research in general. From these limitations, we derive some interesting challenges for future research.

One interesting question in the context of this thesis is how 'realistic' the results are when we use particular combinations of the described methods. The *Explicit Corridor Map* framework [142], in which all methods have been developed, allows the combination of methods from all layers of the planning hierarchy. Future research should indicate whether this free-combination approach has limitations, which methods do not work well with others, and how the order in which particular sub-steps are executed influences the outcome.

Another aspect that will affect future research is the fact that the underlying planning hierarchy is based on the concept of *indicative routes.* The work in this thesis shows that we can gain promising results from subdividing the planning hierarchy into computing a rough indication of a preferred path first, and then refining this indicative route. However, there are still problems that are caused by the inflexibility of that approach, e.g. crowd congestion near obstacle corners when a large number of agents uses a shortest path (with clearance) as an indicative route. We believe that the future lies in overcoming the concept of a fixed indicative route and take it to the next level by either allowing it to dynamically change or by using a two-dimensional *indicative surface* instead of a one-dimensional indicative route.

Second, some of the methods that we discussed – such as the VBP and MIRAN methods in Chapters 4 and 6, respectively – take a set of weights for the weighted regions as an input. While the methods themselves do not aim at finding concrete weight values, the right choice of weight values still has a significant impact on the results that we can obtain when using these methods. As such, an interesting future research direction would be to derive methods that automate the process of finding reasonable weights. Deep-learning techniques or neural-network approaches could be used to automatically learn weight values from recorded video data of real-world locations, or from recorded user-behavior in virtual environments.

When we look even further into the future, the current focus in crowd simulation literature on aspects that can be modeled via geometric rules will certainly have its limitations, too. Simulating human behavior in a *virtual environment* – be it walking behavior and navigation or any other aspect of human life – essentially means understanding and mimicking *real-world* human behavior in its full complexity. In the long run, it involves more than mathematical modeling and combining velocity-based steering behaviors. In this context, crowd simulation can essentially be seen as a artificial-intelligence (AI) research, even though typical AI paradigms and learning techniques are not yet the focus in the field as a whole. Furthermore, research fields that study human nature – such as psychology, sociology, or anthropology – should be involved in this long-term challenge to a great extent. Questions such as how cultural differences influence the walking behavior of a crowd cannot be answered without such expertise. Over the past years, the still comparably young field of crowd-simulation research has started to acknowledge the need for finding proper ways to validate its models [18, 121]. However, how to validate a crowd-simulation model is still an open research question, which is highly challenging and eventually interdisciplinary in its nature.

Overall, it can be said that the field of crowd simulation is gaining more and more popularity in both the academic world and in society and the media as a whole. Crowd simulations have been used to validate environments with respect to safety of real-world places and events[1] for more than a decade now [19], and we believe this trend to continue while the simulations are becoming more reliable. Due to the challenging nature of the field and because simulated crowds are attractive and fascinating to watch for both experts and non-experts, the field of crowd simulation will certainly stay a ‘hot topic’ in the decades to come.

---

[1] For instance, the work presented in this thesis has been used to prepare for the *Grand Depart* of the Tour de France, which happened in Utrecht in July 2015.

BIBLIOGRAPHY

[1] L. Aleksandrov, H. Djidjev, H. Guo, A. Maheshwari, D. Nussbaum, and J.-R. Sack. Algorithms for approximate shortest path queries on weighted polyhedral surfaces. *Discrete & Computational Geometry*, 44:762–801, 2010.

[2] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An epsilon-approximation for weighted shortest paths on polyhedral surfaces. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory (SWAT '98)*, pages 11–22. Springer-Verlag, 1998.

[3] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM*, 52(1):25–53, 2005.

[4] G. Antonini, M. Bierlaire, and M. Weber. Discrete choice models of pedestrian walking behavior. *Transportation Research Part B: Methodological*, 40(8):667–687, 2006.

[5] C. Bajaj. The algebraic degree of geometric optimization problems. *Discrete & Computational Geometry*, 3:177–191, 1988.

[6] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic. Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study. *Proceedings of the National Academy of Sciences*, 105(4):1232–1237, 2008.

[7] B. van Basten, J. Egges, and R. Geraerts. Combining path planners and motion graphs. *Computer Animation and Virtual Worlds*, 22:59–78, 2011.

[8] M. d. Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 3rd ed. edition, 2008.

[9] Y. Björnsson, M. Enzenberger, R. Holte, J. Schaejfer, and P. Yap. Comparison of different grid abstractions for pathfinding on maps. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence)IJCAI'03)*, pages 1511–1512. Morgan Kaufmann Publishers Inc., 2003.

[10] J. C. Butcher. *Numerical Methods for Ordinary Differential Equations.* John Wiley & Sons, 2008.

[11] J.-L. D. Carufel, C. Grimm, A. Maheshwari, M. Owen, and M. Smid. Unsolvability of the weighted region shortest path problem. In *European Workshop on Computational Geometry (EuroCG)*, pages 65–68, 2012.

[12] D. Chen, R. Szczerba, and J. J. Uhran. Planning conditional shortest paths through an unknown environment: a framed-quadtree approach. In *Proceedings of the 8th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '95)*, volume 3, pages 33–38, 1995.

[13] S.-W. Cheng, J. Jin, A. Vigneron, and Y. Wang. Approximate shortest homotopic paths in weighted regions. In O. Cheong, K.-Y. Chwa, and K. Park, editors, *Algorithms and Computation*, volume 6507 of *Lecture Notes in Computer Science*, pages 109–120. Springer Berlin Heidelberg, 2010.

[14] J. Chestnutt, K. Nishiwaki, J. Kuffner, and S. Kagami. An adaptive action model for legged navigation planning. In *Proceedings of the 7th IEEE-RAS International Conference on Humanoid Robots*, pages 196–202, 2007.

[15] J. S. Coleman and J. James. The equilibrium size distribution of freely-forming groups. *Sociometry*, 24(1):36–45, 1961.

[16] M. Costa. Interpersonal distances in group walking. *Journal of Nonverbal Behavior*, 34(1):15–26, 2010.

[17] S. Curtis, A. Best, and D. Manocha. Menge: A modular framework for simulating crowd movement. Technical report, University of North Carolina at Chapel Hill, 2014.

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

[19] W. Daamen. SimPed: A pedestrian simulation tool for large pedestrian areas. In *Proceedings of the 2002 EuroSIW (European Simulation Interoperability Workshop)*, 2002.

[20] M. de Berg, J. Matoušek, and O. Schwarzkopf. Piecewise linear paths among convex obstacles. *Discrete and Computational Geometry*, 14:9–29, 1995.

[21] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[22] P. Drews, D. Macharet, and M. Campos. A terrain-based path planning for mobile robots with bounded curvature. In *Proceedings of 2012 the Robotics Symposium and Latin American Robotics Symposium (SBR-LARS)*, pages 202–207, 2012.

[23] D. Dummit and R. Foote. *Abstract Algebra.* John Wiley & Sons, 2003.

[24] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The field d* algorithm. *Journal of Field Robotics*, 23:79–101, 2006.

[25] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3—4):189–208, 1971.

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

[27] N. Fridman, G. A. Kaminka, and A. Zilka. The impact of culture on crowd dynamics: An empirical approach. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS '13)*, pages 143–150. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[28] J. J. Fruin. *Pedestrian Planning and Design*. Metropolitan Association of Urban Designers and Environmental Planners, 1971.

[29] J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, pages 29–38, 1999.

[30] H. J. Q. García and L. Garrido. Towards exploration of unknown dynamic worlds using multiple robots. In *Proceedings of the 6th Mexican International Conference on Artificial Intelligence (MICAI '07), Special Session*, pages 407–417, 2007.

[31] R. Geraerts. Planning Short Paths with Clearance using Explicit Corridors. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.

[32] R. Geraerts and M. Overmars. Sampling and node adding in probabilistic roadmap planners. *Journal of Robotics and Authonomous Systems (RAS)*, 54:165–173, 2006.

[33] R. Geraerts and M. Overmars. The corridor map method: Real-time high-quality path planning. In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation*, pages 1023–1028, 2007.

[34] R. Geraerts and E. Schager. Stealth-based path planning using corridor maps. In *Proceedings of the 23rd International Conference on Computer Animation and Social Agents (CASA 2010)*, 2010.

[35] A. Gheibi, A. Maheshwari, and J.-R. Sack. Weighted region problem in arrangement of lines. In *Proceedings of the 25th Canadian Conference on Computational Geometry (CCCG 2013)*. Carleton University, Ottawa, Canada, 2013.

[36] S. Ghosh. *Visibility Algorithms in the Plane.* Cambridge University Press, 2007.

[37] S. Ghosh and D. M. Mount. An output sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20:888–910, 1991.

[38] Y. Guo, L. Parker, D. Jung, and Z. Dong. Performance-based rough terrain navigation for nonholonomic mobile robots. *IEEE Industrial Electronics Society*, pages 2811–2816, 2003.

[39] Y. Guo, M. Yang, and J. Cheng. Knowledge-inducing global path planning for robots in environment with hybrid terrain. *International Joirnal of Advanced Robotics Systems*, pages 239–248, 2010.

[40] S. J. Guy, J. Chhugani, S. Curtis, P. Dubey, M. C. Lin, and D. 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.

[41] D. Harabor and A. Botea. Hierarchical path planning for multi-size agents in heterogeneous environments. *Computational Intelligence and Games*, pages 258–265, 2008.

[42] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100 –107, 1968.

[43] D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39(1):1–24, 2005.

[44] D. Helbing, I. J. Farkas, P. Molnar, and T. Vicsek. Simulation of pedestrian crowds in normal and evacuation situations. *Pedestrian and Evacuation Dynamics*, 21:21–58, 2002.

[45] D. Helbing and P. Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, 1995.

[46] D. Helbing, P. Molnar, I. J. Farkas, and K. Bolay. Self-organizing pedestrian movement. *Environment and Planning B*, 28(3):361–384, 2001.

[47] M. Held. Vroni and arcvroni: Software for and applications of voronoi diagrams in science and engineering. In *Proceedings of the 8th International Symposium onVoronoi Diagrams in Science and Engineering (ISVD 2011)*, pages 3–12, June 2011.

[48] K. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*,

SIGGRAPH ’99, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999.

[49] S. Hoogendoorn. Microscopic simulation of pedestrian flows. In *Proceedings of the 82nd Annual Meeting at the Transportation Research Board*, pages 1–11, 2003.

[50] T. Huang, M. Kapadia, N. Badler, and M. Kallmann. Path planning for coherent and persistent groups. In *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA 2014)*, pages 1652–1659, May 2014.

[51] R. L. Hughes. The flow of human crowds. *Annual Review of Fluid Mechanics*, 35(1):169–182, 2003.

[52] N. Jaklin, A. Cook IV, and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds (CAVW)*, 24:285–295, 2013.

[53] N. Jaklin and R. Geraerts. Navigating through virtual worlds: From single characters to large crowds. In D. Russel and J. M. Laffey, editors, *Handbook of Research on Gaming Trends in P-12 Education*, chapter 25, pages 527–554. IGI Global, 2015.

[54] N. Jaklin, A. Kremyzas, and R. Geraerts. Adding sociality to virtual pedestrian groups. In *21st ACM Symposium on Virtual Reality Software and Technology (VRST 2015)*, pages 163–172, 2015.

[55] N. Jaklin, M. Tibboel, and R. Geraerts. Computing high-quality paths in weighted regions. In *Proceedings of the 7th International ACM SIGGRAPH Conference on Motion in Games (MIG 2014)*, pages 77–86, 2014.

[56] N. Jaklin, W. van Toll, and R. Geraerts. Way to go – a framework for multi-level planning in games. In *Proceedings of the 3rd International Planning in Games Workshop (ICAPS’13 | PG2013)*, pages 11–14, 2013.

[57] J. James. The Distribution of Free-Forming Small Group Size. *American Sociological Review*, 18(5):569–570, 1953.

[58] M. Kallmann. Navigation queries from triangular meshes. In *Proceedings of the 3rd international conference on Motion in Games (MIG 2010)*, pages 230–241. Springer-Verlag, 2010.

[59] M. Kallmann. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the 9th Eurographics / SIGGRAPH Symposium on Computer Animation (SCA 2010)*, 2010.

[60] A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. In *Proceedings of the 3rd ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2004)*, pages 19–28, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[61] A. Kamphuis, M. Rook, and M. Overmars. Tactical path finding in urban environments. In *Proceedings of the 1st International Workshop on Crowd Simulation*, pages 51–60, 2005.

[62] S.-J. Kang, Y. Kim, and C.-H. Kim. Live path: adaptive agent navigation in the interactive virtual world. *The Visual Computer*, 26(6-8):467–476, June 2010.

[63] I. Karamouzas, R. Geraerts, and M. Overmars. Indicative routes for path planning and crowd simulation. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 113–120, 2009.

[64] I. Karamouzas and M. Overmars. Simulating human collision avoidance using a velocity-based approach. In *Proceedings of the 7th Workshop on Virtual Reality Interactions and Physical Simulations (VRIPHYS 10)*, pages 125–134. Eurographics Association, 2010.

[65] I. Karamouzas and M. Overmars. Simulating and evaluating the local behavior of small pedestrian groups. *IEEE Transactions on Visualization and Computer Graphics*, 18(3):394–406, 2012.

[66] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, Aug 1996.

[67] K. Kedem, R. L. andJ. Pach, and M. Sharir. On the union of jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, 1:59–70, 1986.

[68] J. Kelly, A. Botea, and S. Koenig. Offline planning with Hierarchical Task Networks in video games. In *Proceedings of the 2008 Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 60—-65, 2008.

[69] W. Kerr and D. Spears. Robotic simulation of gases for a surveillance task. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2905–2910, 2005.

[70] A. Kimmel, A. Dobson, and K. Bekris. Maintaining team coherence under the velocity obstacle framework. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12, pages 247–256, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[71] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21:354–363, 2005.

[72] G. Köster, F. Treml, M. Seitz, and W. Klein. Validation of crowd models including social groups. In U. Weidmann, U. Kirsch, and M. Schreckenberg, editors, *Pedestrian and Evacuation Dynamics 2012*, pages 1051–1063. Springer International Publishing, 2014.

[73] V. Kountouriotis, S. Thomopoulos, and Y. Papelis. An agent-based crowd behaviour model for real time crowd behaviour simulation. *Pattern Recognition Letters*, 44(0):30 – 38, 2014.

[74] J. Laumond. Obstacle growing in a nonpolygonal world. *Information Processing Letters*, 25(1):41–50, 1987.

[75] S. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

[76] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[77] S. LaValle and J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2000.

[78] K. Lee, M. Choi, Q. Hong, and J. Lee. Group behavior from video: A data-driven approach to crowd simulation. In *Proceedings of the 6th ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA 2007)*, pages 109–118, 2007.

[79] S. Lemercier, A. Jelic, R. Kulpa, J. Hua, J. Fehrenbach, P. Degond, C. Appert-Rolland, S. Donikian, and J. Pettré. Realistic following behaviors for crowd simulation. In *Computer Graphics Forum*, volume 31, pages 489–498, 2012.

[80] J. Liddle, A. Seyfried, B. Steffen, W. Klingsch, T. Rupprecht, A. Winkens, and M. Boltes. Microscopic insights into pedestrian motion through a bottleneck, resolving spatial and temporal variations. *arXiv preprint arXiv:1105.1532v1*, 2011.

[81] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Proceedings of the 16th Conference on Advances in Neural Information Processing Systems (NIPS 2003)*. MIT Press, 2003.

[82] W.-Y. Lo, C. Knaus, and M. Zwicker. Learning motion controllers with adaptive depth perception. In *Proceedings of the 11th ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2012)*, pages 145–154. Eurographics Association, 2012.

[83] B. Lowerre. The harpy speech understanding system, 1990.

[84] E. Luczak and A. Rosenfeld. Distance on a hexagonal grid. *IEEE Transactions on Computers*, 25:532–533, 1976.

[85] E. Masehianm and M. Amin-Naseri. A voronoi diagram-visibility graph-potential field compound algorithm for robot path planning. *Journal of Field Robotics*, pages 275–300, 2004.

[86] C. Mata and J. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract), 1997.

[87] J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.

[88] M. Moussaïd, D. Helbing, and G. Theraulaz. How simple rules determine pedestrian behavior and crowd disasters. *Proceedings of the National Academy of Sciences*, 108(17):6884–6888, 2011.

[89] M. Moussaïd, N. Perozo, S. Garnier, D. Helbing, and G. Theraulaz. The walking behaviour of pedestrian social groups and its impact on crowd dynamics. *PLoS ONE*, 5(4):e10047, 2010.

[90] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[91] S. Musse and D. Thalmann. A model of human crowd behavior : Group inter-relationship and collision detection analysis. In D. Thalmann and M. van der Panne, editors, *Computer Animation and Simulation '97*, Eurographics, pages 39–51. Springer Vienna, 1997.

[92] A. N., S. Koenig, and C. A. Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Proceedings of the 24th Conference on Artificial Intelligence (AAAI 2010)*, 2010.

[93] B. Nagy. Finding shortest path with neighbourhood sequences in triangular grids. In *Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis (ISPA 2001)*, pages 55–60. IEEE, 2001.

[94] B. Nagy. Metrics based on neighbourhood sequences in triangular grids. *Pure Mathematics and Applications*, 13:259–274, 2002.

[95] B. Nagy. Shortest paths in triangular grids with neighbourhood sequences. *Journal of Computing and Information Technology*, 11(2):111–122, 2003.

[96] B. Nagy. Weighted distances on a triangular grid. In *Combinatorial Image Analysis*, volume 8466 of *Lecture Notes in Computer Science*, pages 37–50. Springer International Publishing, 2014.

[97] R. Narain, A. Golas, S. Curtis, and M. C. Lin. Aggregate dynamics for dense crowd simulation. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 122:1–122:8. ACM, 2009.

[98] A. Nash. *Any-Angle Path Planning*. PhD thesis, University of Southern California, 2012.

[99] R. Oliva and N. Pelechano. Clearance for diversity of agents' sizes in navigation meshes. *Computers & Graphics*, 47(0):48 – 58, 2015.

[100] J. Ondřej, J. Pettré, A.-H. Olivier, and S. Donikian. A synthetic-vision based steering approach for crowd simulation. *ACM Transanctions on Graphics*, 29(4):123:1–123:9, July 2010.

[101] S. Paris and S. Donikian. Activity-driven populace: a cognitive approach to crowd simulation. *IEEE Computer Graphics and Applications*, 29:34–43, 2009.

[102] S. Paris, J. Pettré, and S. Donikian. Pedestrian reactive navigation for crowd simulation: A predictive approach. *Computer Graphics Forum*, 26(3):665–674, 2007.

[103] C. Park, A. Best, S. Narang, and D. Manocha. Simulating high-dof human-like agents using hierarchical feedback planner. In *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology (VRST 2015)*, pages 153–162, 2015.

[104] J. H. Park, F. A. Rojas, and H. S. Yang. A collision avoidance behavior model for crowd simulation based on psychological findings. *Computer Animation and Virtual Worlds*, 24(3-4):173–183, 2013.

[105] S. I. Park, F. Quek, and Y. Cao. Modeling social groups in crowds using common ground theory. In *Proceedings of the Winter Simulation Conference (WSC 2012)*, pages 113:1–113:12, 2012.

[106] N. Pelechano, J. Allbeck, and N. Badler. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2007)*, pages 99–108, 2007.

[107] N. Pelechano, J. Allbeck, and N. Badler. *Virtual Crowds: Methods, Simulation, and Control (Synthesis Lectures on Computer Graphics and Animation)*. Morgan and Claypool Publishers, 2008.

[108] J. Pettré, J.-P. Laumond, and D. Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *Proceedings of the 1st International Workshop on Crowd Simulation*, 2005.

[109] L. C. A. Pimenta, N. Michael, R. C. Mesquita, G. A. S. Pereira, and V. Kumar. Control of swarms based on hydrodynamic models. In *Proceedings of the 2009 IEEE Internation Conference on Robotics and Automation (ICRA 2008)*, pages 1948–1953, 2008.

[110] I. Pohl. Heuristic Search viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3):193 – 204, 1970.

[111] F. Qiu and X. Hu. Modeling group structures in pedestrian crowd simulation. *Simulation Modelling Practice and Theory*, 18(2):190–205, 2010.

[112] S. Rabin. *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[113] D. Reece, M. Kraus, and P. Dumanior. Tactical movement planning for individual combatants. In *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*, 2000.

[114] C. W. Reynold. Flocks, herds, and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH '87)*, volume 21, pages 25–34, 1987.

[115] C. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of the 1999 Game Developers Conference*, pages 763–782, 1999.

[116] J. L. Rosenfeld, A.and Pfaltz. Distance functions on digital pictures. *Pattern Recognition*, 1(1):33–61, 1968.

[117] M. Schuerman, S. Singh, M. Kapadia, and P. Faloutsos. Situation agents: agent-based externalized steering logic. *Computer Animation and Virtual Worlds*, 21(3-4):267–276, 2010.

[118] W. Shao and D. Terzopoulos. Autonomous pedestrians. In *Proceedings of the 4th ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA 2005)*, pages 19–28. ACM, 2005.

[119] A. Shoulson, N. Marshak, M. Kapadia, and N. Badler. Adapt: The agent development and prototyping testbed. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 9–18, 2013.

[120] S. Singh, M. Kapadia, P. Faloutsos, and G. Reinman. An open framework for developing, evaluating, and sharing steering algorithms. In *Proceedings of the 2nd International Workshop on Motion in Games (MIG 2009)*, pages 158–169, 2009.

[121] S. Singh, M. Kapadia, P. Faloutsos, and G. Reinman. Steerbench: A benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds*, 20(5-6):533–548, 2009.

[122] G. Song and N. Amato. Randomized motion planning for car-like robots with c-prm. In *Proceedings of the 2001 IEEE International Conference on Intellingent Robots and Systems (IROS 2001)*, pages 37–42, 2001.

[123] A. Stentz and I. C. Mellon. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10:89–100, 1993.

[124] G. K. Still. *Crowd Dynamics*. PhD thesis, University of Warwick, 2000.

[125] R. Strand. Weighted distances based on neighbourhood sequences. *Pattern Recognition Letters*, 28(15):2029–2036, 2007.

[126] X. Sun, S. Koenig, and W. Yeoh. Generalized adaptive a*. In *Proceedings of the 2008 International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 469–476, 2008.

[127] Z. Sun and J. Reif. Bushwhack: An approximation algorithm for minimal paths through pseudo-euclidean spaces. In *Proceedings of the 12th Annual International Symposium on Algorithms and Computation*, pages 160–171. Springer, 2001.

[128] D. Thalmann and S. R. Musse. *Crowd Simulation, Second Edition*. Springer, 2013.

[129] P. M. Torrens. Moving agent pedestrians through space and time. *Annals of the Association of American Geographers*, 102(1):35–66, 2012.

[130] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. In *ACM Transactions on Graphics*, volume 25, pages 1160–1168, 2006.

[131] K. Trovato. *A* Planning in Discrete Configuration Spaces of Autonomous Systems*. PhD thesis, University of Amsterdam, 1996.

[132] B. Ulicny and D. Thalmann. Towards interactive real-time crowd behavior simulation. *Computer Graphics Forum*, 21:767–775, 2002.

[133] J. van den Berg, S. Guy, M. C. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *Robotics Research*, pages 3–19. Springer, 2011.

[134] J. van den Berg, R. Shah, A. Huang, and K. Goldberg. ANA*: Anytime Nonparametric A*. In *Proceedings of the 2011 AAAI Conference on Artificial Intelligence*, 2011.

[135] W. van der Sterren. Tactical path-finding with A*. *Game Programming Gems*, 3:294–306, 2002.

[136] A. van Goethem, N. Jaklin, A. Cook IV, and R. Geraerts. On streams and incentives: A synthesis of individual and collective crowd motion. In *28th International Conference on Computer Animation and Social Agents (CASA 2015)*, pages 29–32, 2015.

[137] A. van Goethem, N. Jaklin, A. Cook IV, and R. Geraerts. On streams and incentives: A synthesis of individual and collective crowd motion. Technical Report UU-CS-2015-005, Utrecht University, 2015.

[138] W. van Toll, A. Cook IV, and R. Geraerts. Navigation meshes for realistic multi-layered environments. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 3526–3532, 2011.

[139] W. van Toll and R. Geraerts. Dynamically pruned a* for re-planning in navigation meshes. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015)*, pages 2051–2057, 2015.

[140] W. van Toll, A. C. IV, and R. Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds (CAVW)*, 23(6):536–546, 2012.

[141] W. van Toll, A. C. IV, and R. Geraerts. Real-time density-based crowd simulation. *Computer Animation and Virtual Worlds (CAVW)*, 23:59–69, 2012.

[142] W. van Toll, N. Jaklin, and R. Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN*, 2015.

[143] G. Vizzari, L. Manenti, and L. Crociani. Adaptive pedestrian behaviour for the preservation of group cohesion. *Complex Adaptive Systems Modeling*, 1(1):1–29, 2013.

[144] U. Weidmann. Transporttechnik der fussgänger. *IVT, Institut für Verkehrsplanung, Transporttechnik, Strassen-und Eisenbahnbau*, 90, 1992.

[145] R. Wein, J. van den Berg, and D. Halperin. The visibility–voronoi complex and its applications. In *Proceedings of the 21st Annual Symposium on Computational Geometry (SCG 2005)*, pages 63–72, 2005.

[146] E. Welzl. Constructing the Visibility Graph for n-Line Segments in $O(n^2)$ Time. *Information Processing Letters*, 20:167–171, 1985.

[147] E. B. Werner and C. G. Rossi. *Manual of visual fields.* Churchill Livingstone New York, 1991.

[148] M. Wolf. Theorizing navigable space in video games. *DIGAREC Keynote-Lectures*, 2009/2010.

[149] Q. Wu, Q. Ji, J. Du, and X. Li. Simulating the local behavior of small pedestrian groups using synthetic-vision based steering approach. In *Proceedings of the 12th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry (VRCAI 2013)*, pages 41–50. ACM, 2013.

[150] S. Xu and H.-L. Duh. A simulation of bonding effects and their impacts on pedestrian dynamics. *IEEE Transactions on Intelligent Transportation Systems*, 11(1):153–161, 2010.

[151] A. Yahja, S. Singh, and A. Stentz. An efficient online path planner for outdoor mobile robots. *Robotics and Autonomous Systems*, 32:129–143, 2000.

[152] K. Yu. Finding a natural-looking path by using generalized visibility graphs. *PRICAI 2006: Trends in Artificial Intelligence*, 4099:170–179, 2006.

[153] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, 1949.

SAMENVATTING

Virtuele omgevingen worden steeds belangrijker voor vele aspecten van de wereld waarin we leven. Meeslepende virtuele werelden komen veel voor in films, computerspellen en online gemeenschappen. Ze zijn ook belangrijk voor andere toepassingen dan entertainment, zoals software voor training en educatie, simulaties van evenementen en evacuaties, planologie en de analyse van menselijke invloed op ongelukken. Trainings- en educatiesoftware kent vele toepassingen, variërend van lesgeven met virtuele karakters tot het trainen van politie, brandweer of soldaten in virtuele scenario's. Andere toepassingen zijn online plattegronddiensten zoals *Open Street Map*, *Google Street View* en *Mapillary.*

Voor een geloofwaardige virtuele wereld zijn goede algoritmen nodig voor de navigatie van virtuele karakters. De paden (looproutes) die deze karakters afleggen moeten soepel zijn, geen onnodige omwegen bevatten, afstand tot obstakels bewaren, terreinen en speciale gebieden in acht nemen en botsingen tussen karakters vermijden. Andere onderdelen zijn het coördineren van grote menigtes karakters in zowel rustige als drukke situaties, en het simuleren van sociaal gedrag in kleine groepen. Bestaande algoritmen voor padplanning en modellen voor *crowd simulation* hebben moeite met deze taken, waardoor het nog moeilijk is om diverse soorten gedrag te modelleren.

Dit proefschrift richt zich op drie computationele taken waar bestaande algoritmen nog moeite mee hebben: paden *plannen* in gewogen gebieden, paden *volgen* in gewogen gebieden, en het *coördineren* van drukke virtuele menigtes en sociale groepen. We laten zien waarom deze taken moeilijk op te lossen zijn met huidige algoritmen in een raster- of graafrepresentatie van de virtuele omgeving. Ook ontwikkelen we nieuwe methodes die deze problemen efficiënt oplossen in een representatie van beloopbare vlakken.

In het eerste deel van dit proefschrift bekijken we het berekenen van een pad in een $2D$-omgeving met meerdere soorten gewogen gebieden (*weighted regions*). Zulke gebieden stellen een bepaald type terrein voor (zoals 'weg', 'gras' of 'water') of een bepaalde psychologische invloed op karakters (zoals 'gevaarlijk' of 'aantrekkelijk'). Het gewicht van een gebied geeft aan hoe moeilijk of ongewenst het voor een karakter is om door dit gebied te lopen. Voor een karakter met een verzameling voorkeuren (een gewicht per type gebied) willen we een pad berekenen van een startpositie naar een doelpositie zodat de gewogen Euclidische lengte wordt geminimaliseerd. We analyseren de kwaliteit van optimale paden berekend in *8-neighbor grids* (rasters waarin karakters horizontaal, verticaal en diagonaal kunnen bewegen). We geven een bovengrens van de kosten van een dergelijk pad vergeleken met het werkelijke optimale pad, onder de aanname dat elke rastercel slechts één type gebied bevat. Ook presenteren we een nieuw padplanning-algoritme genaamd *Vertex-based Pruning* (VBP) dat een A^*-zoekactie op een raster combineert met een bestaand ϵ-benaderend algoritme op de exacte gebieden. We laten zien dat VBP ϵ-benaderingen van het optimale pad sneller berekent dan bestaande methodes als de omgeving voldoende groot is, terwijl het nog steeds acceptabele padkosten garandeert. De paden die VBP oplevert kunnen gebruikt worden als ‘indicatieve route’: een ruwe richtlijn van het pad dat het karakter hoopt te volgen gedurende de simulatie.

In het tweede deel van dit proefschrift presenteren we *Modified Indicative Routes and Navigation* (MIRAN), een nieuwe methode voor het *volgen* van een indicatieve route in een omgeving met gewogen gebieden. De route die als input dient, kan automatisch berekend zijn met onze nieuwe VBP-methode of met een eenvoudiger algoritme, of deze kan met de hand getekend zijn door een gebruiker. De MIRAN-methode gebruikt *sampling* (een benadering van de route met losse punten) om een karakter de route op een soepele manier te laten volgen, op basis van diens persoonlijke voorkeuren (gewichten). Gebruikers kunnen twee parameters instellen: de eerste bepaalt hoe dicht de *sample*-punten bij elkaar liggen en de tweede geeft aan hoe ver het karakter vooruit mag kijken over de route. De tweede parameter bepaalt ook hoeveel afstand het karakter maximaal mag overslaan (oftewel de grootte van de maximale *shortcut*). We nemen eerst aan dat het karakter gemodelleerd wordt als een punt. Vervolgens breiden we de methode uit naar karakters met een schijfvorm van willekeurige grootte: we passen de berekeningen aan zodat de straal van de schijf in acht wordt genomen.

In het derde deel van dit proefschrift ontwikkelen we een nieuw model voor crowdsimulatie en een nieuwe manier om het loopgedrag van kleine sociale groepen te simuleren. Net als in de vorige delen zijn onze methodes *agent-based* (gebaseerd op berekeningen per individueel karakter), maar we gebruiken ze om efficiënt grote aantallen karakters tegelijk te simuleren. Ten eerste introduceren we een nieuw model voor crowdsimulatie genaamd *Stream*. Dit model combineert de voordelen van *agent-based* en *flow-based* modellen door middel van lokale regels per karakter. Het Stream-model geeft significante verbeteringen in de coördinatie en flow van een menigte waarvan de dichtheid willekeurig kan veranderen. Stream is geschikt voor real-time simulaties en gaming-toepassingen. Ten tweede presenteren we een methode met de naam *Social Groups and Navigation* (SGN). SGN simuleert sociale groepjes van karakters die sociale formaties aannemen en wachtgedrag vertonen. Tot slot laten we zien hoe al onze bijdragen gecombineerd kunnen worden in een crowdsimulatie-framework gebaseerd op de *Explicit Corridor Map*. We beschrijven enkele implementatiedetails van dit framework.

Acknowledgements

First of all, I would like to thank my daily supervisor and co-promoter Roland Geraerts. Roland, who was my daily supervisor throughout the whole four-year period of my PhD, always had good advice for overcoming problems and an open ear for questions related to my research and academic work in general. I would also like to thank Roland for giving me the freedom of doing independent research and letting me choose my own research topics while still pointing me into the right direction every now and then. We also shared many exciting conference trips, e.g. to Istanbul and Los Angeles, which were always great experiences and a lot of fun.

Second, I would like to thank my promoter Marc van Kreveld for giving very fruitful comments and advice on the thesis, and for helping with improving its overall value and results. Similar to Roland, Marc always had good advice and an open ear for questions. I would like to thank you both for providing a very pleasant working environment throughout the four years of this project.

Next, I would like to thank all members of the reading committee for their valuable time. Special thanks go to Mark de Berg, who pointed out a flaw in an earlier version of the proof in Chapter 3, and who gave suggestions on how to fix it. The final version of the proof is based on his suggestions.

Last but not least, I would like to thank all current and former members of the Virtual Worlds division for the great working environment, the interesting conversations during our lunch breaks, all the great drinks&dinner events in the city center of Utrecht, and the poker nights at Nicolas', Wouter's, and Angelos' place. Special thanks to Wouter and Arne for the great atmosphere and all the fun we had in our office. Thanks to Wouter in particular for constantly inventing new *sound noises*, and to Arne for not going insane sharing an office with two crazy people ☺.

CURRICULUM VITAE

Norman Jaklin was born on January 9, 1982, in Cologne, Germany. He finished his *Abitur* (pre-university education) at the Kaiserin-Theophanu-Schule in Cologne in 2001. In 2005, he obtained a *Vordiplom* (Bachelor of Science) in Computer Science – with Mathematics as a subsidiary subject – at the Rheinische Friedrich-Wilhelms-Universität Bonn. In 2011, he obtained a *Diplom* (Master of Science) in Computer Science at the same university. In his Diplom thesis *Manhattan Networks in 3D*, he studied mathematical properties and approximation algorithms of the Manhattan-Network Problem in 2D and their applicability in 3D. During his studies, Norman worked as a high-school pupil's coach for English and Mathematics, as a programmer in the Computer Security Group at the Bonn-Aachen International Center for Information Technology (B-IT), and as a tutor for Computational Geometry at the University of Bonn. In 2011, he started a PhD at Utrecht University, for which he completed his thesis in 2016.