Navigation for Characters and Crowds in Complex Virtual Environments

Wouter van Toll

Navigation for Characters and Crowds in Complex Virtual Environments

Wouter van Toll

© 2017 Wouter van Toll

Cover design by Wouter van Toll

This work is licensed under a Creative Commons Attribution 4.0
International License. To view a copy of this license, visit:
`https://creativecommons.org/licenses/by/4.0/`

ISBN: 978-90-393-6711-7

Navigation for Characters and Crowds in Complex Virtual Environments

Navigatie voor Karakters en Menigtes in Complexe Virtuele Omgevingen

(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 donderdag 30 maart 2017 des middags te 2.30 uur

door

Wouter Geert van Toll

geboren op 6 maart 1989 te Utrecht

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

Contents

I Introductory Chapters

1 Introduction **3**
1.1 Complex Real-Time Crowd Navigation 3
1.2 Thesis Outline per Chapter 6
1.3 Contributions 8

2 Related Work **11**
2.1 Motion and Path Planning 11
2.2 Navigation Meshes in 2D Environments 13
2.3 Navigation Meshes in 3D Environments 15
2.4 Crowd Simulation 17

3 Preliminaries **21**
3.1 Voronoi Diagrams and the Medial Axis 21
3.2 A* Search 22

II Navigation Meshes

4 The Explicit Corridor Map in 2D Environments **29**
4.1 Introduction 29
4.2 Definitions 30
4.3 Operations and Applications 33
4.4 Implementation 37
4.5 Experiments and Results 39
4.6 Conclusions and Future Work 42

5 The Explicit Corridor Map in Multi-Layered Environments **45**
5.1 Introduction 45
5.2 Definitions of Environments 46
5.3 Definitions of the Medial Axis and ECM 49
5.4 Construction Algorithm Outline 51

5.5 Properties of a Closed Connection 52
5.6 Opening a Connection 55
5.7 Analysis 62
5.8 Implementation 66
5.9 Experiments and Results 67
5.10 Conclusions and Future Work 74

6 The Explicit Corridor Map in Dynamic Environments 77
6.1 Introduction 77
6.2 Related Work 79
6.3 Inserting a Point Obstacle 79
6.4 Inserting a Line Segment or Convex Polygon 86
6.5 Deleting an Obstacle 92
6.6 Extensions 93
6.7 Experiments and Results 94
6.8 Conclusions and Future Work 96

7 A Comparative Study of Navigation Meshes 99
7.1 Introduction 99
7.2 Related Work 101
7.3 Definitions 102
7.4 Properties of Navigation Meshes 104
7.5 Theoretical Comparison 105
7.6 Quality Metrics for Navigation Meshes 109
7.7 Experimental Comparison 115
7.8 Conclusions and Future Work 121

III Path Planning and Crowd Simulation Algorithms

8 Dynamic Re-planning 137
8.1 Introduction 137
8.2 Related Work 138
8.3 Problem Description 139
8.4 Optimal Dynamically Pruned A* 141
8.5 Local Re-planning 144
8.6 Experiments and Results 147
8.7 Conclusions and Future Work 154

9 Density-Based Crowd Simulation 157
9.1 Introduction 157
9.2 Related Work 159
9.3 Density-Annotated Navigation Mesh 162

9.4 Density-Based Path Planning Algorithm . . . 164
9.5 Re-planning . . . 165
9.6 Experiments and Results . . . 168
9.7 Conclusions and Future Work . . . 176

10 A Generic Crowd Simulation Framework 187
10.1 Introduction . . . 187
10.2 Related Work . . . 188
10.3 Multi-Level Planning Hierarchy . . . 189
10.4 Implementation Details . . . 195
10.5 Experiments and Results . . . 199
10.6 Conclusions and Future Work . . . 208

IV Concluding Remarks

11 Summary and Conclusions 213
11.1 Navigation Meshes . . . 213
11.2 Path Planning and Crowd Simulation Algorithms . . . 216

12 Discussion and Future Work 219
12.1 Navigation Meshes . . . 219
12.2 Path Planning and Crowd Simulation Algorithms . . . 221
12.3 Other Topics and Outlook . . . 223

Bibliography 225

Samenvatting 239

Acknowledgments 243

Curriculum Vitae 245

PART I

Introductory Chapters

1 Introduction

Virtual environments (also referred to as *virtual worlds*) play an important role in modern society. Intuitively, the term 'virtual' may make readers think of virtual reality games or futuristic movies. However, a virtual environment is generally an environment that exists in a computer program. The environment can be either a fictional scene (such as a level in a computer game) or a 3D computer model of a real-world environment (such as an office building or a train station).

There are many applications in which a virtual environment is inhabited by crowds of autonomous *characters* that walk, run, or otherwise move along surfaces. Simulating the movement of these characters in a computer program is called *crowd simulation*. Characters are sometimes referred to as *(autonomous) agents*, particularly in the field of artificial intelligence (AI) where the emphasis lies on the intelligent decisions made by each entity. Other possible terms for characters include robots, entities, pedestrians, and NPCs, depending on the application or research area at hand. In this thesis, we will use the term 'characters' to avoid confusion.

In the entertainment industry, crowd simulations occur in e.g. computer games in which non-player characters make the environment appear more lively, movies in which characters and their trajectories are computer-generated, and strategy games in which groups of characters should intelligently move to a location designated by the player.

Crowd simulation is also becoming increasingly important for purposes outside the area of entertainment. For instance, simulations can be used to predict and prevent dangerous situations during a crowded event such as a festival, to estimate how quickly a building can be evacuated in case of an emergency, or to teach public safety personnel how to interact with crowds of people. In all of these cases, a simulation is a safe and cost-effective alternative to a real-life setup.

Many of these applications require that the crowd simulation runs in *real-time*, i.e. that the behavior of the crowd is generated on the fly, and fast enough to allow users to interact with the environment and the crowd.

1.1 Complex Real-Time Crowd Navigation

This thesis studies various research problems related to real-time crowd simulation. In such a crowd simulation, each character should efficiently compute a path to its goal location, and the characters should smoothly traverse their individual

paths while avoiding obstacles and other members of the crowd. To simulate this behavior for large crowds in *real-time*, we need efficient data structures that represent the environment, and efficient algorithms for path planning and for the other aspects of crowd simulation.

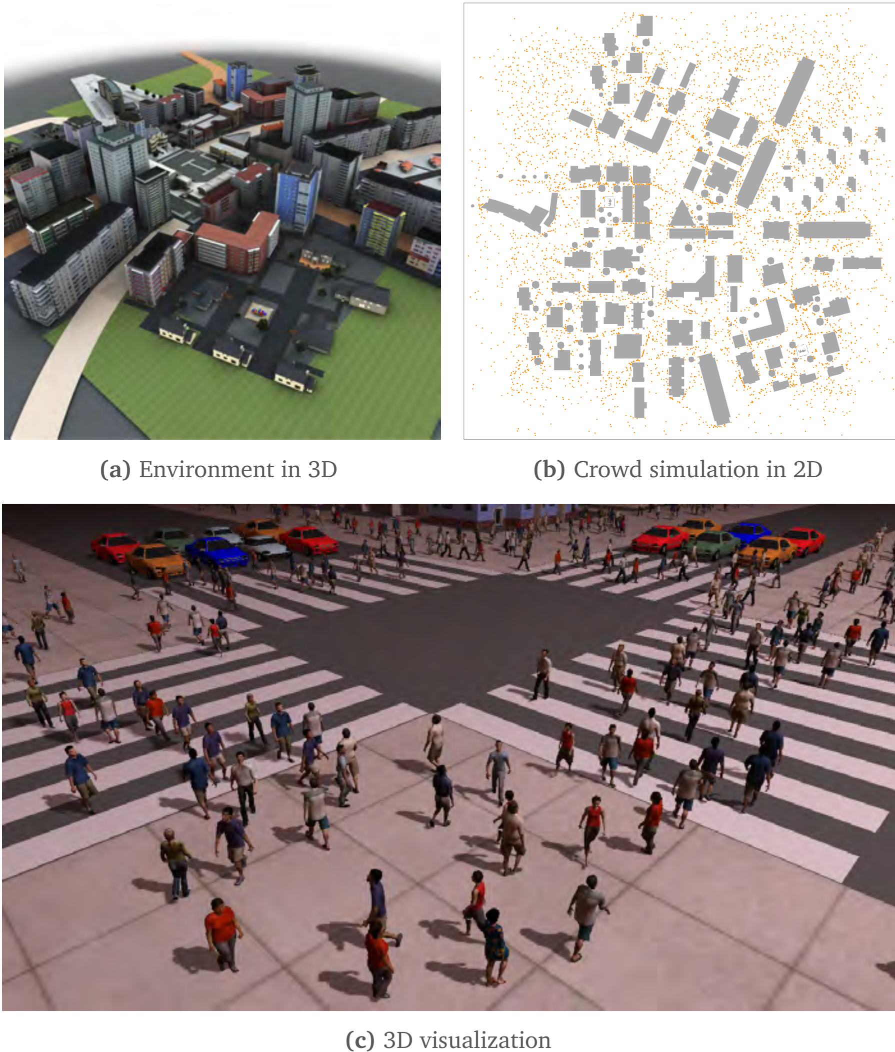
A *navigation mesh* is a representation of the walkable parts of a virtual environment for the purpose of efficient navigation. It can serve as a basis for real-time path planning and crowd simulation. A navigation mesh can be created by hand (i.e. by letting a designer 'draw' the regions in which characters can walk), but ideally, it should be extracted *automatically* from a 2D or 3D virtual environment.

In this thesis, we investigate how to create and use navigation meshes for what we call *complex scenarios*. One factor that can make a simulation complex is the *environment*. If a virtual environment is two-dimensional (or if it can be simplified to a 2D representation), and if no obstacles are added or removed during the simulation, then a navigation mesh is relatively easy to construct, and many solutions for path planning and crowd simulation are available. Figure 1.1 shows an example of a crowd simulation in a 3D city that can be simplified to 2D.

However, modern applications may feature environments that cannot be represented in 2D. For instance, the 2D simplification from Figure 1.1b would no longer be sufficient if we wanted to include the interior of a multi-story building. The automatic construction of efficient navigation meshes for such environments has not been researched as much. The full 3D model of an environment is typically too detailed for the purpose of navigation. Therefore, in this thesis, we will introduce the concepts of *walkable environments* (WEs) and *multi-layered environments* (MLEs) that represent the surfaces on which characters can walk. We will analyze the properties of WEs and MLEs, and we will present a navigation mesh that can represent an MLE.

Furthermore, in many applications, obstacles may appear or disappear during the simulation, such that routes may become available or unavailable over time. For instance, imagine a vehicle that enters the environment to block a road, or a fence that is added or removed. A navigation mesh should be able to reflect these dynamic changes. The navigation mesh presented in this thesis supports real-time insertions and deletions of dynamic obstacles. These dynamic changes also affect the way in which characters plan their paths.

The second complicating factor that we study is the *crowd* itself. We are interested in applications in which each character in the crowd can have its own size, walking speed, goal location, and personal preferences. A crowd simulation system and its underlying navigation mesh should support these differences between characters. Furthermore, the characters in the crowd are expected to plan paths that can be followed despite the presence of other characters. This can be particularly challenging if there are many characters in the simulation, i.e. if the *crowd density* is high.



(a) Environment in 3D

(b) Crowd simulation in 2D

(c) 3D visualization

Figure 1.1: Example of a crowd simulation in a virtual environment. **(a)** The original 3D model of the environment. **(b)** For crowd simulation purposes, this environment can be simplified to a 2D representation with polygonal obstacles. Characters in the simulation are shown as orange disks; we have enlarged these disks for clarity. This figure was generated using our crowd simulation framework from Chapter 10. **(c)** The simulated characters can be visualized as animated 3D models in the original environment [65].

We emphasize that this thesis focuses on the *geometric* aspects of real-time crowd simulations. Next to these geometric aspects, there are at least two other important simulation components: *high-level planning*, which concerns the high-level decision-making of each character, and *animation*, which enriches the characters with sophisticated 3D animations (such as in Figure 1.1c).

In most chapters of this thesis, we consider high-level planning and animation to be out of scope: we assume that each character has already chosen a particular goal position (but not yet a path towards this goal), and we treat and visualize characters as disks or cylinders that move along surfaces. In Chapter 10, however, we will describe the complete set of tasks involved in a crowd simulation, and we will explain how these components fit together into a generic crowd simulation framework.

1.2 Thesis Outline per Chapter

This thesis consists of four parts.

Part I introduces concepts that are relevant for the remaining chapters.

- Chapter 2 describes related work on path planning, navigation meshes, and crowd simulation. It explains why navigation meshes are our concept of choice for simulating crowds of heterogeneous characters.
- Chapter 3 provides background knowledge on topics that will re-occur in various other chapters. It focuses on Voronoi diagrams (which are crucial for our navigation mesh in Part II) and A* search (which we will frequently use and adapt in Part III).

Part II revolves around *navigation meshes*: data structures that represent the walkable areas of a virtual environment for the purpose of path planning and crowd simulation. We introduce the Explicit Corridor Map (ECM) navigation mesh and its extensions and operations, and we conduct a comparative study between the ECM and other state-of-the-art navigation meshes.

- Chapter 4 formally introduces the Explicit Corridor Map for 2D environments. The definitions and geometric operations of the ECM will be used in other parts of the thesis as well. We describe and compare multiple implementations of the ECM construction algorithm, and we measure the efficiency of various geometric operations in the ECM. We show that the ECM supports efficient path planning for disk-shaped characters of any radius.
- In Chapter 5, we present the novel problem domain of *multi-layered environments* (MLEs), which consist of multiple connected layers such that each layer can be handled in 2D. An MLE is a useful simplification of a real-world 3D environment for path planning and crowd simulation purposes. Examples of such environments include multi-story buildings, train stations, and

cities with tunnels and bridges. We give an algorithm that computes the ECM of an MLE, we prove that this algorithm is correct, we describe our implementation, and we use it to conduct various experiments related to the multi-layered ECM. We show that most operations from the 2D ECM (Chapter 4) apply to the multi-layered extension as well.

- Chapter 6 extends the ECM to *dynamic environments* in which polygonal obstacles can appear or disappear at run-time. Practical examples include a vehicle that is blocking an alley, or a fence or wall that is removed to create a new passage. We show how to efficiently update the ECM in response to such dynamic events. This enables path planning and crowd simulation in dynamic environments.

- In Chapter 7, we perform the first *comparative study* of navigation meshes. Using both a theoretical analysis and quantitative metrics, we compare various state-of-the-art navigation meshes, including the ECM. The goal of this chapter is not to expose which navigation mesh is 'the best' for a particular environment, but to propose a way to objectively measure the quality of a navigation mesh, and to set a new standard for experimental research in this area. We use our results to identify the most interesting directions for future work.

Part III presents novel path planning and crowd simulation algorithms for navigation meshes. While our implementations of these algorithms are based on the ECM, we describe the algorithms in an abstract way, such that they can be applied to other navigation meshes as well.

- In Chapter 8, we consider *re-planning* of paths in dynamic navigation meshes, such as the dynamic ECM from Chapter 6. When a dynamic event has occurred, the path of a character may have become incorrect or suboptimal. We present an algorithm that lets a character efficiently re-plan a new optimal path. The algorithm is more memory-friendly than existing re-planning algorithms, which are typically designed for single characters or for other categories of motion planning. Therefore, our algorithm can be used for real-time crowd simulation in dynamic environments.

- Chapter 9 shows how the real-world concept of *crowd density* can be included in a navigation mesh. Characters in a crowd simulation can use this information to plan density-aware paths. We show that this leads to more variety among characters, and that it can improve the flow of a crowd. This behavior is *emergent*: a crowd-wide effect follows from the individual choices of each character.

- Chapter 10 proposes a generic multi-level *framework* that summarizes how crowd simulation software can be structured. This framework subdivides the complex problem of crowd simulation into more manageable subproblems.

We describe our own implementation of such a framework; this implementation is based on the ECM and has been used for all experiments in this thesis. We also give more insight in the software's architecture and performance. Our ECM framework can simulate tens of thousands of heterogeneous characters in real-time. It has been successfully used to perform simulations of various real-life events, such as the crowd flow during the *Grand Départ* of the Tour de France in Utrecht in 2015.

Finally, **Part IV** concludes the thesis.

- Chapter 11 summarizes the most important conclusions and contributions from each chapter.
- In Chapter 12, we discuss the limitations of the work in this thesis, and we identify multiple important topics for future work on navigation meshes and crowd simulation.

1.3 Contributions

As described in the previous section, the chapters of Parts II and III each have their own contributions in different aspects. To summarize this further, the main overall contributions of this thesis are the following:

- We define the domains of walkable environments (WEs) and multi-layered environments (MLEs), which represent a 3D environment for path planning and crowd simulation. (Chapter 5)
- We show how to compute the medial axis and the ECM for an MLE. As such, we present the first navigation mesh that supports path planning for disks of any size in a multi-layered environment. (Chapter 5)
- We show how the ECM can be updated in real-time when obstacles are added or removed. (Chapter 6)
- We introduce definitions and metrics that set a new standard for analyzing and comparing navigation meshes in 2D and 3D. Our comparative study can be used to focus future research. (Chapter 7)
- We present an algorithm that enables efficient re-planning of paths in dynamically changing environments. The algorithm is memory-friendly and suitable for real-time crowd simulation. (Chapter 8)
- We use crowd density information to let the individual route choices of each character improve the overall crowd simulation. (Chapter 9)
- We describe a generic framework that subdivides crowd simulation into modular components. We present an implementation of this framework that

combines many topics from this thesis. The implementation can simulate large crowds of characters with individual sizes and properties in real-time. (Chapter 10)

Implementations of our data structures and algorithms form an important component of this thesis. Many chapters contain a section with implementation details and experiments to show that we can obtain efficient, robust, or visually pleasing results. In short, we show that our concepts *and* implementations can be used to simulate navigation for characters and crowds in complex virtual environments in real-time.

We note in advance that all experiments in this thesis have been performed using the same hardware: a Windows 7 PC with a 3.20 GHz Intel i7-3930K CPU, an NVIDIA GeForce GTX 680 GPU, and 16 GB of RAM. All source code has been compiled in Visual Studio 2013. This ensures consistency between the results of all chapters.

2 Related Work

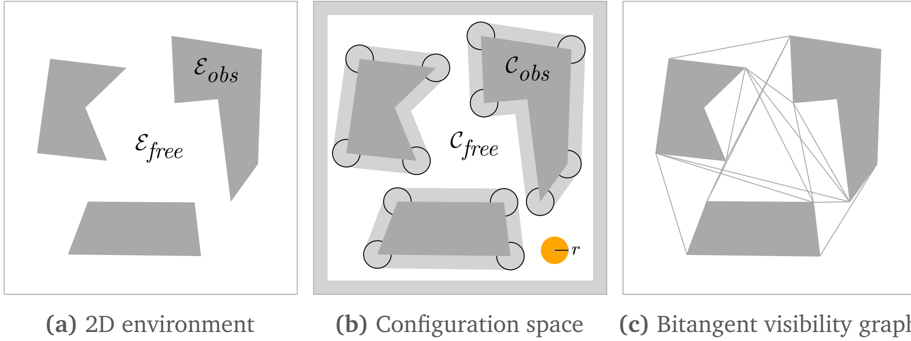
The main chapters of this thesis are all related to path planning, navigation meshes, or crowd simulation. Therefore, we now provide an overview of the related work in these areas. We focus on the work that is related to multiple chapters of the thesis. Any related work that is highly specific to a particular subtopic will be treated in the appropriate chapter (e.g. dynamic environments in Chapter 6, dynamic re-planning in Chapter 8, and crowd density in Chapter 9). Also, Chapter 3 will provide more details on Voronoi diagrams and A* search, two concepts on which various chapters of this thesis are strongly based.

2.1 Motion and Path Planning

The problem of traditional *motion planning* originates from robotics research. In motion planning, a robot needs to compute a collision-free trajectory from one *configuration* to another [91, 92]. For instance, for a robot arm with rotating joints, a configuration can be represented by a set of joint angles. The number of *degrees of freedom* for the robot determines the complexity of a configuration. The *configuration space* $\mathcal{C}$ is the set of all configurations [100]; it can be subdivided into collision-free configurations in which the robot does not touch any obstacles ($\mathcal{C}_{free}$), and colliding configurations in which it does ($\mathcal{C}_{obs}$). The configuration space is often different from the *workspace* $\mathcal{W}$ in which the problem is embedded (which is typically $\mathbb{R}^2$ or $\mathbb{R}^3$).

High-dimensional configuration spaces, such as for a detailed humanoid robot that moves through a 3D workspace, are typically too complex to represent in an exact manner. Instead, a common solution for motion planning is to represent the configuration space in a *sampling-based* manner. The idea behind this approach is to sample the configuration space and build a *roadmap* that connects the samples that are collision-free, until the start and goal configuration have been connected. This roadmap is a simplified representation of $\mathcal{C}_{free}$. Famous examples of such techniques are Probabilistic Roadmaps (PRMs) [79] and Rapidly-Exploring Random Trees (RRTs) [88]. Although parts of a roadmap could be re-used in multiple queries, RRTs are typically designed for 'single-query' problems for a particular start and goal configuration.

In crowd simulations, the virtual environment (i.e. the workspace $\mathcal{W}$) is often three-dimensional, such as in Figure 1.1a. However, characters are constrained to walkable surfaces. In this thesis, we will refer to these walkable surfaces as



(a) 2D environment **(b)** Configuration space **(c)** Bitangent visibility graph

Figure 2.1: (a) A simple 2D environment. Polygonal obstacles and the boundary form the obstacle space $\mathcal{E}_{obs}$ (shown in gray). The remainder is the free space $\mathcal{E}_{free}$ (shown in white). **(b)** For a disk (shown in orange) with radius r, the configuration space can be computed by a *Minkowski sum* that inflates all obstacles by a disk with radius r. The dark gray and light gray areas form $\mathcal{C}_{obs}$. **(c)** The bitangent visibility graph of the original obstacles is shown in black. This graph can be used to plan shortest paths for a point character.

the *free space* $\mathcal{E}_{free}$. In many environments, the surfaces of $\mathcal{E}_{free}$ can be projected onto a common ground plane P without causing overlap. This projection of $\mathcal{E}_{free}$ onto P is a 2D subset of the plane with polygonal obstacles, or (equivalently) a 2D polygon with holes. We will refer to these obstacles or holes as the *obstacle space* $\mathcal{E}_{obs}$. A simple example of $\mathcal{E}_{free}$ and $\mathcal{E}_{obs}$ is shown in Figure 2.1a. If height differences along surfaces can be ignored during path planning, then characters essentially plan their paths in this projected version of $\mathcal{E}_{free}$.

For now, we assume that $\mathcal{E}_{free}$ has been projected onto P, i.e. that it is two-dimensional. In Section 2.3, we will extend the discussion to *multi-layered* environments.

To allow simulations of large crowds, an individual character is often represented by a disk. For a disk-shaped character of radius r, the configuration space $\mathcal{C}$ can be obtained by computing a *Minkowski sum* that inflates the obstacles in $\mathcal{E}_{free}$ by r. This is illustrated in Figure 2.1b. By using Minkowski sums, planning a path for the character in $\mathcal{W}$ can be reduced to planning a path for a *point* in $\mathcal{C}$ [91, 92].

Our discussion will now focus on path planning for points; each concept can be extended to disks of a particular radius by applying a Minkowski sum first. However, note that the configuration space $\mathcal{C}$ is different for each distinct character radius. In this thesis, we will instead present a navigation mesh that can be used to plan paths for characters of different sizes.

To plan a *shortest* path for a point in a 2D configuration space $\mathcal{C}$, a shortest-path map [53] describes how all points in $\mathcal{C}_{free}$ can be reached from a particular query point. A visibility graph [35] can be used to plan shortest paths between arbitrary pairs of points in $\mathcal{C}_{free}$. The visibility graph contains an edge for each pair of

obstacle vertices that are mutually visible, i.e. each pair of obstacle vertices (v, w) for which the line segment $\overline{vw}$ does not intersect any other obstacles.

A *bitangent* visibility graph is a visibility graph from which all edges that are not needed for shortest-path queries have been removed. Figure 2.1c shows an example. For a configuration space with n obstacle vertices, both versions of the visibility graph have $\mathcal{O}(n^2)$ edges and can be computed in $\mathcal{O}(n^2)$ time. The Visibility-Voronoi Complex [161] is an extension that implicitly encodes the visibility graph of $\mathcal{E}_{free}$ for all disk sizes. It can be constructed in $\mathcal{O}(n^2 \log n)$ time; it can generate the visibility graph for a particular disk radius in $\mathcal{O}(n \log n)$ time.

In a real-time crowd simulation, it is important that characters can plan their paths efficiently. For such applications, a visibility graph is too complex; instead, it is common to use a less complex graph that may not always yield the shortest path. Such a graph for path planning on surfaces is sometimes called a *waypoint graph*.

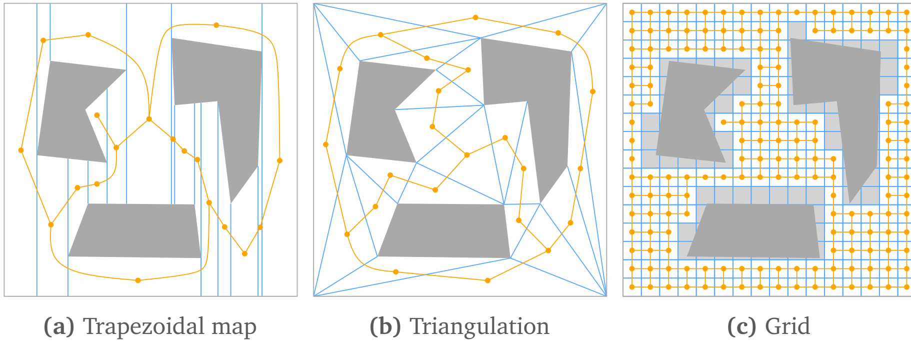
However, purely graph-based approaches are generally not ideal for crowd simulation: characters would need to follow the edges of the graph exactly (which leads to unnatural motion and more collisions between characters), or they would have to perform expensive geometric tests to check how they can deviate from an edge. This deviation is necessary for allowing e.g. collision avoidance and coordination in groups [151], which are crucial elements for simulating crowds. This motivates the need for different representations of 2D environments.

2.2 Navigation Meshes in 2D Environments

There are several ways to automatically subdivide the free space $\mathcal{E}_{free}$ into connected polygonal areas. Snook [137] and Tozour [156] were among the first to use the term *navigation mesh* for such a subdivision, which has now become a common term in the area of path planning and crowd simulation.

Part II of this thesis revolves entirely around navigation meshes. Chapters 4 to 6 present the Explicit Corridor Map (ECM) navigation mesh and its geometric operations and extensions. Chapter 7 compares the ECM to other state-of-the-art navigation meshes.

2D spatial subdivisions are well-described in various books on motion planning [91, 92], but they are also studied in computational geometry [5, 7] because they have many other applications besides path planning. The term ‘navigation mesh’ essentially denotes a spatial subdivision that is used for navigation purposes. One example of a 2D spatial subdivision is the trapezoidal decomposition (or *trapezoidal map*), which subdivides $\mathcal{E}_{free}$ into vertical slabs at obstacle vertices (Figure 2.2a). Another example is a *triangulation*, which subdivides $\mathcal{E}_{free}$ into triangles such that all triangle vertices are also obstacle vertices (Figure 2.2b). The constrained Delaunay triangulation is a specific triangulation in which the triangles have special geometric properties [5].



(a) Trapezoidal map (b) Triangulation (c) Grid

Figure 2.2: Examples of 2D spatial subdivisions. The boundaries of regions are shown in light blue. The dual graph is shown in orange. In the right image, the grid cells that are (partly) covered by an obstacle are shown in light gray.

The *dual graph* of a navigation mesh contains a vertex for each polygonal region and an edge for each pair of connected regions. The dual graph is also displayed in the examples of Figure 2.1. Characters can use a search algorithm such as Dijkstra's algorithm [26] or A* [45] to find an optimal path through this dual graph. This path corresponds to a sequence of polygonal regions to move through, such that characters can use the available space to locally adjust their movements during the simulation. As such, navigation meshes are more flexible for crowd simulation than graphs.

Although all 2D subdivisions can serve as navigation meshes, researchers have also focused on creating navigation meshes with particular *advantages* for the purpose of path planning or crowd simulation [33, 43, 67, 112, 156]. Such advantages can include fast and robust construction algorithms, or a small number of regions, which implies a small dual graph and therefore efficient path planning.

Some navigation meshes support path planning for disks of *any radius*, in contrast to the 'Minkowski sum' approach that assumes a particular radius beforehand. Two notable examples are the Explicit Corridor Map (ECM) [33], on which the majority of this thesis is based, and the Local Clearance Triangulation (LCT) by Kallmann [67]. In Chapter 7, we will compare the ECM and the LCT theoretically and experimentally. Oliva and Pelechano have investigated path planning for disks in other navigation meshes [113].

An alternative option is to construct a *grid* that subdivides the environment into regular cells, such as in Figure 2.2c. Grids are a popular choice for path planning: they are easy to implement and well-studied by researchers (see e.g. [15, 32, 85, 95, 139]). However, because a grid only *approximates* the geometry of $\mathcal{E}_{free}$, grids tend to have resolution problems: a coarse grid (with few cells) does not capture the environment's details, whereas a fine grid (with many cells) quickly becomes too costly to store and query. To account for this, there are ways to use an 'adaptive'

grid resolution that varies throughout the environment [32]. Similarly, a *quadtree* subdivides $\mathcal{E}_{free}$ into squares that get gradually smaller wherever more detail is required, up to a desired level of precision [7]. Still, a grid-like representation of $\mathcal{E}_{free}$ is never perfect if the obstacles are not axis-aligned.

2.3 Navigation Meshes in 3D Environments

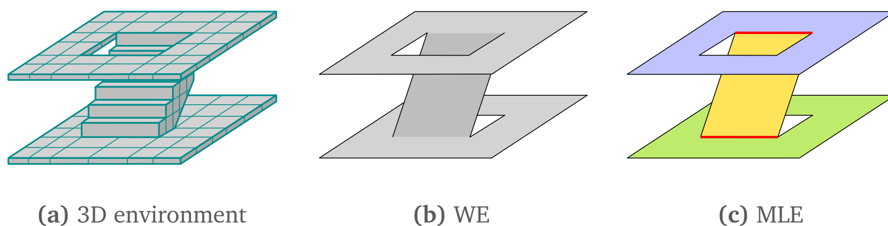
Up until now, we have only discussed virtual environments for which the free space $\mathcal{E}_{free}$ can be projected onto a common plane. Creating a navigation mesh is considerably more complex if the environment *cannot* be simplified to 2D. For such environments, the navigation mesh is sometimes constructed by hand (e.g. by level designers, in the case of computer games), but this process is time-consuming and subject to human error.

Therefore, increasingly more research is being dedicated to the automatic creation of navigation meshes for *arbitrary 3D environments*. A 3D environment can contain various types of geometry such as floors, ceilings, walls, and staircases. Figure 2.3a shows an example. To construct a navigation mesh for such an environment, it is common to first extract the *walkable surfaces* from the 3D geometry through a filtering process, while assuming that the environment has a consistent *direction of gravity*. The result of this filtering process is the free space $\mathcal{E}_{free}$ of the environment. We will also refer to it as a *walkable environment* (WE); an example of a WE is illustrated in Figure 2.3b. We will define WEs more formally in Chapter 5.

Many navigation mesh techniques obtain an approximation of $\mathcal{E}_{free}$ by discretizing the 3D environment into traversable and non-traversable cubes, or *voxels*. An early example by Pettré [121] supports arbitrary character sizes and represents $\mathcal{E}_{free}$ by using overlapping disks. The Recast Navigation toolkit by Mononen [102] is robust and very popular in the computer games industry [158]; it includes many parameters, such as a fixed character radius that is subtracted from $\mathcal{E}_{free}$ during the construction. The NEOGEN method by Oliva and Pelechano [114] uses voxels to subdivide $\mathcal{E}_{free}$ into 2D components; it then uses an exact algorithm for each component. We will explain these methods in more detail in Chapter 7, in which we will conduct a comparative study of navigation meshes.

Voxel-based methods can handle arbitrary 3D geometry: the discretization into voxels automatically resolves issues caused by e.g. intersecting polygons. However, a disadvantage is that these methods are less scalable to physically large environments: such environments require many voxels to obtain sufficient precision, which affects the method's construction time and memory usage. Therefore, *exact* alternatives to 3D filtering are also being investigated [123].

For many applications, it is useful to subdivide the walkable environment into *layers* such that each layer can be treated as a 2D problem. We will refer to such



(a) 3D environment (b) WE (c) MLE

Figure 2.3: **(a)** A 3D environment is a collection of polygons in 3D. **(b)** A walkable environment (WE) is a set of polygons along which characters can walk. We also refer to this as the free space $\mathcal{E}_{free}$. **(c)** A multi-layered environment (MLE) is a subdivision of the WE into 2D layers. Connections between layers are shown as red line segments.

a subdivision as a *multi-layered environment* (MLE). An example is displayed in Figure 2.3c. An MLE representation enables the use of *exact* navigation mesh algorithms: we can apply a 2D algorithm for each layer and then connect the layers appropriately. In Chapter 5, we define MLEs more formally and we extend the Explicit Corridor Map to MLEs.

There are several ways to obtain an MLE from a WE. Oliva and Pelechano use voxelization in their NEOGEN method [114], Deusdado et al. have used rendering techniques that assume certain properties such as axis-alignment [24], and Whiting et al. have shown how to extract layers from a CAD drawing [162]. For an arbitrary WE represented by a triangle mesh, Hillebrand has proven that obtaining an optimal MLE (with a minimum number of connections) is NP-hard in the number of triangles [54], but he has shown that good results can be obtained using heuristics [55].

Thus, there are currently two main categories of navigation mesh construction algorithms for 3D environments: *voxel-based* methods that discretize the environment and that can handle arbitrary 3D geometry, and *exact* methods that require the environment to be pre-processed into a 2D or multi-layered description of the free space. In Chapter 7, we will compare navigation meshes of both categories.

Researchers have also investigated navigation meshes for other types of geometry or character movement. An environment can be subdivided into 3D *volumes* to encode height differences and variable vertical clearance [42, 90]. Alternatively, one could perform crowd simulations on arbitrary surfaces with no consistent direction of gravity [10, 127]. However, in that domain, the algorithms are less mature, and concepts such as path planning and collision avoidance are more computationally expensive. Other methods allow characters to *jump* between surfaces by either checking for jumping possibilities on the fly [98] or annotating a navigation mesh with jump links beforehand [16].

In this thesis, we will not look into such extensions. We will focus only on the *walkable space* of 2D environments and multi-layered environments.

2.4 Crowd Simulation

Navigation meshes are useful for simulating *crowds* of virtual characters with individual properties and goals. Crowd simulation is a large research field with many components including path planning, collision avoidance between characters, visualization, animation, and the evaluation of realism. Several books and articles exist that give good overviews of this field [3, 27, 69, 72, 119, 146, 163]. We will give a short summary of the most relevant topics for this thesis.

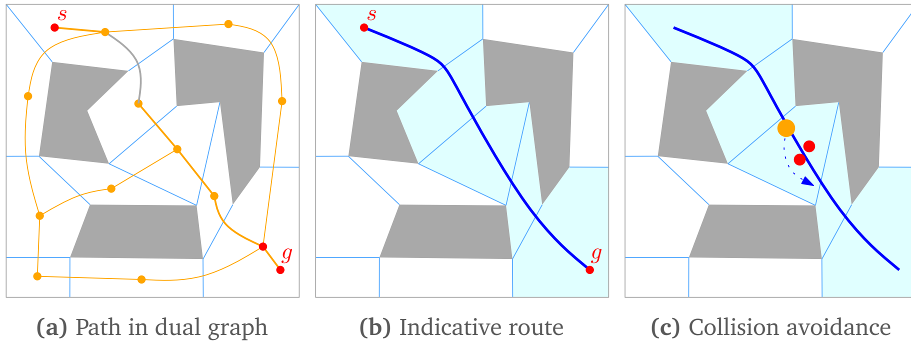
2.4.1 Navigation Meshes: Global and Local Planning

As explained, to plan a path in a navigation mesh, a character first plans a path through the dual graph of the mesh (Figure 2.4a). Within the corresponding sequence of regions, sometimes referred to as a *corridor*, the character can compute an *indicative route*: a rough indication of the character's desired trajectory (Figure 2.4b). In this thesis, we use the term 'path' for a sequence of edges in the graph, and the term 'route' or 'indicative route' for an actual curve through the navigation mesh. We refer to the process of computing a path and an indicative route as *global planning*.

During the simulation, an indicative route can be traversed smoothly in real-time [62, 74]. While traversing this route, a character can *locally* avoid collisions with other characters (Figure 2.4c). Developing intelligent collision-avoidance algorithms is an active research topic; most of these algorithms either use forces [46, 118, 126] or velocity selection [9, 76, 105]. In particular, characters are usually *not* modelled as obstacles in the navigation mesh because this would require many complex update operations in each simulation step. Next to collision avoidance, other local algorithms can be used to model small *social groups* of characters [77, 87].

Local behavior can also be modelled by using a grid. For example, Narain et al. [108] have used a grid to model dense crowds that compress and decompress fluently. A related concept is a *cellular automaton*: a grid representation of the environment in which a character makes local decisions based on its surrounding grid cells [99, 166]. However, grid-based rules may result in sudden changes in behavior when characters move to different cells. Moreover, as discussed earlier, grid representations do not scale well to large or detailed environments.

A disadvantage of treating global and local planning separately is that an indicative route may be difficult to follow in practice due to local obstacles. For example, clusters of characters might be blocking the way. This tends to happen more often if the crowd is large and dense. To improve crowd coordination at high densities, one can use information about the crowd's distribution in the global planning phase [73, 75] or at a local level [36]. Chapter 9 will present an algorithm that can plan a density-aware global path in a navigation mesh, and it will show how to use this concept in a crowd simulation.



(a) Path in dual graph **(b)** Indicative route **(c)** Collision avoidance

Figure 2.4: Path planning in a navigation mesh. **(a)** The dual graph of the navigation mesh is shown in orange and black. Given two query points s and g (shown in red), we compute a path (shown in black) through this dual graph. **(b)** The path corresponds to a sequence of navigation mesh regions (highlighted in blue). Within these regions, we can compute an indicative route (the blue curve). **(c)** During the simulation, the character (orange disk) follows the indicative route while using the corridor to avoid other characters (red disks).

In Chapter 10, we will present a generic crowd simulation framework in which these components can be combined, and we will describe our own implementation of such a framework that uses the Explicit Corridor Map navigation mesh. The *Menge* crowd simulation framework by Curtis et al. is based on similar principles [21].

2.4.2 Other Methods

A different category of crowd simulation algorithms aims to unify the global and local planning levels by defining a *potential field*: a grid representation of the environment that stores the optimal walking direction in each cell. These directions are updated in real-time in response to the crowd's movement [117, 157].

Potential fields can efficiently model *flows* of dense crowds in which many characters share the same goals and properties. Unlike in the approaches that separate global and local planning, characters will always move in an optimal direction instead of following their individual paths. However, this is less appropriate whenever the behavior of individual characters becomes important. For instance, each goal region typically requires its own potential field. For simulating large *heterogeneous* crowds in which each character has different properties, potential fields are not scalable, and navigation meshes with local collision avoidance are preferred.

Other research focuses more on the artificial intelligence of individual characters [58, 116, 132, 165]. In their models, characters can make *high-level plans* (e.g. 'go to the train station, buy a train ticket, and enter the train') based on semantic information in the environment.

In Chapter 10, we will treat high-level planning and character animation as separate levels in a generic five-level crowd simulation framework. As mentioned in Chapter 1, this thesis does not focus on these aspects of path planning and crowd simulation. We will therefore not discuss more related work on animation or high-level planning.

3 Preliminaries

In this chapter, we briefly discuss fundamental concepts that will be used throughout this thesis. We describe the Voronoi diagram, the medial axis, and the A* search algorithm. These descriptions should provide readers with the background knowledge required for the remaining chapters.

3.1 Voronoi Diagrams and the Medial Axis

The *Voronoi diagram* (VD) is a fundamental geometric data structure with many applications. We describe the VD in this chapter because it forms the basis of our Explicit Corridor Map (ECM) navigation mesh, which is central to many chapters in Part II of the thesis. In particular, our algorithms for the multi-layered ECM (Chapter 5) and for dynamic updates (Chapter 6) require insight in the VD.

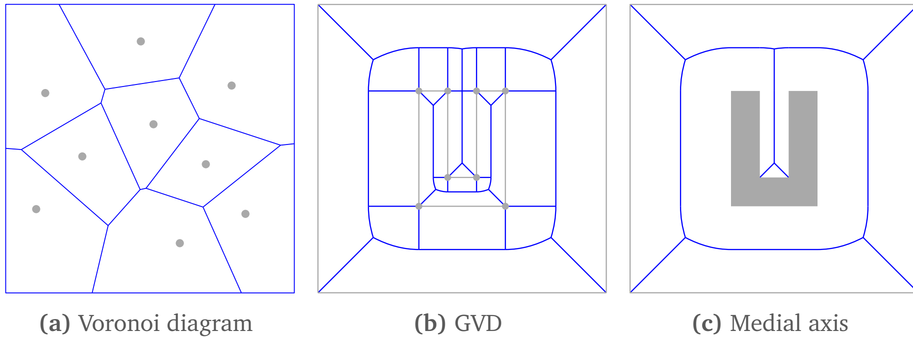
Excellent overviews of the Voronoi diagram exist, such as the books by Okabe et al. [111] and Aurenhammer et al. [5]. Most of the results mentioned in this section can also be found in these books.

For a planar set of point sites, the VD is a subdivision of the plane into cells such that all points in a cell have the same nearest site. An example of a VD is shown in Figure 3.1a. The Voronoi diagram induces a graph. Its *edges* are parts of bisectors: line segments or half-lines on which every point is equidistant to two sites. These bisectors meet at *vertices* that are equidistant to at least three sites.

The dual graph of a Voronoi diagram has a vertex for each site and an edge for each pair of sites that share a Voronoi edge. This graph is known as the Delaunay triangulation (DT); in the design and analysis of algorithms, the VD and the DT are often interchangeable [7].

The VD can be extended to handle line segments and polygons as sites. This version is sometimes called the *generalized Voronoi diagram* or GVD [33, 94]. The edges of a GVD consist of line segments and parabolic arcs, and degree-2 vertices occur when a bisector changes its shape, i.e. when the closest point on a nearest site changes between a vertex and an edge of that site. For a set of interior-disjoint line segments with n distinct endpoints, the GVD has $\mathcal{O}(n)$ edges and vertices. An example is shown in Figure 3.1b.

Unfortunately, there are multiple definitions of the GVD, differing mostly in how they handle site vertices shared by multiple sites. The term 'generalized Voronoi diagram' is also used for other generalizations of the VD [111].



(a) Voronoi diagram (b) GVD (c) Medial axis

Figure 3.1: (a) The Voronoi diagram (shown in blue) of a set of point sites (shown in gray), clipped to a bounding box. Each edge in the Voronoi diagram is a line segment. **(b)** The generalized Voronoi diagram (GVD) for line segment sites can also contain parabolic arcs as edges. In this figure, the edges of the bounding box are sites as well. **(c)** The *medial axis* of a polygon with holes (or an environment with obstacles) is a subset of the edges of the line segment GVD. It is the basis for the ECM navigation mesh used throughout this thesis.

There are multiple ways to compute a (generalized) Voronoi diagram in $\mathcal{O}(n \log n)$ time, including a plane sweep algorithm by Fortune [30], a divide-and-conquer approach by Shamos and Hoey [131], and (randomized) incremental construction by Green and Sibson [39]. In practice, it is challenging to compute the Voronoi diagram in a *numerically robust* way, especially when using imprecise floating-point numbers. Various researchers have discussed this problem of numerical robustness (e.g. [49, 59, 143]). Popular modern implementations of GVD construction algorithms include Vroni [48] and components of Boost [14] and CGAL [17]. The Boost implementation is based on Fortune's plane sweep algorithm; Vroni and CGAL use incremental construction. Alternatively, an approximation of the GVD can be computed using graphics hardware [57, 142].

A data structure closely related to the (generalized) Voronoi diagram is the *medial axis* (MA). Several definitions of the MA exist; most of them are based on 2D polygons with or without holes [12, 20, 93, 124, 164]. Informally, the medial axis of a polygon P is a subset of the GVD of P's boundary segments. Definitions of the MA mainly differ in their choice of which edges to prune from the GVD.

In Section 4.2.2, we will give our own definition of the medial axis to avoid confusion throughout the rest of the thesis. The example in Figure 3.1c is based on this definition as well.

3.2 A* Search

Throughout various chapters of this thesis, we will refer to the *A* search algorithm* [45] that is often used to find an optimal path in a graph. It can be seen as a

variant of Dijkstra's shortest-path algorithm [26] that is extended to steer the search towards a particular goal.

3.2.1 Overview

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph in which each edge in $\mathcal{E}$ is denoted by a pair of vertices (V, W). Let $c(V, W)$ be the *traversal cost* of an edge (V, W). We assume that all edge costs are non-negative, to ensure that optimal paths are well-defined. The A* algorithm can also be applied to *directed* graphs, but we will focus on undirected graphs for simplicity.

A* finds a path through a graph from a start vertex $S \in \mathcal{V}$ to a goal vertex $G \in \mathcal{V}$ by performing *best-first search*. Starting at S, the algorithm iteratively 'expands' the vertex V for which the sum $g(V) + h(V)$ is lowest. Here, $g(V)$ is the *total cost* of the best discovered path from S to V so far, and $h(V)$ is a *heuristic* that estimates the remaining cost of the optimal path from V to G.

To find the most promising vertex at all times, the vertices that are candidates for expansion are stored in an *open list*, sorted by their values of $g + h$. Initially, this list contains only S, but whenever a vertex is expanded, its adjacent vertices will be added to the open list as well.

If the function h is *admissible* (i.e. if it never overestimates the optimal path cost for any vertex), then A* computes an optimal path. If h is also *consistent* (i.e. if $h(G) = 0$ and the decrease in h is never larger than the increase in g), then the best path from S to a vertex V is known by the time V is expanded. In that case, vertices never need to be expanded more than once, which allows us to store expanded vertices in a *closed list* to speed up the algorithm. All consistent heuristics are admissible, but not all admissible heuristics are consistent.

3.2.2 Pseudocode

For completeness, we give the pseudocode of A* in Algorithm 3.1. This will be particularly useful for Chapters 8 and 9 in which variants of the A* algorithm will be presented. In Algorithm 3.1, all elements related to the *closed list* are shown in *gray* to indicate that the use of a closed list is optional.

In this version of A*, each vertex V has a *parent* pointer that denotes the vertex preceding V on the best path to V found so far. As soon as the goal vertex G is reached, we trace the final path back to the start vertex S by iteratively following these parent pointers. This approach only works if there is at most one edge between any pair of vertices, i.e. if a vertex pair (V, W) denotes a unique edge. If the graph does not meet this constraint, there are at least three simple solutions:

(a) For each vertex pair (V, W), only use the edge between V and W that has the lowest cost.

(b) Extend the parent pointers such that they include information about the specific edge that has been chosen.

(c) Extend the graph: for each pair of vertices (V, W) with two edges between V and W, add an extra vertex W' at the position of W, connect one of the two edges to W' instead of W, and add a zero-length edge (W, W'). This solution is used in our implementation of the Explicit Corridor Map.

Algorithm 3.1: A*(S, G)

```
1:  g(S) ← 0, S.parent ← NULL
2:  OPEN ← {S}, CLOSED ← ∅
3:  while OPEN ≠ ∅
4:      V ← argmin_{V'∈OPEN} {g(V') + h(V')}
5:      Remove V from OPEN
6:      Add V to CLOSED
7:      if V = G
8:          return the path from S to G via parent pointers
9:      for each edge (V, W)
10:         if W ∈ CLOSED
11:             continue
12:         if g(V) + c(V, W) < g(W)
13:             g(W) ← g(V) + c(V, W)
14:             W.parent ← V
15:             Insert or update W in OPEN
16: return NULL
```

3.2.3 A* in Grids and Navigation Meshes

The A* algorithm is often explained using grids as examples because grids are easy to understand and to implement [95]. Also, many optimizations and variants of A* have been designed specifically for grids [15, 32], or they have been explained in terms of grids in their original publications [85, 86, 97]. A* therefore tends to be mistaken for a grid-based path planning algorithm, but it is defined for graphs in general.

For navigation meshes, the graph that is used for path planning is embedded in $\mathbb{R}^3$. Therefore, its edge costs are often distance-based, e.g. the cost of an edge is equal to its curve length. Furthermore, because all regions of the navigation mesh are sufficiently flat to walk on, it is common to use the curve length of the edge *projected* onto the ground plane.

Likewise, the heuristic function h is often implemented as the 2D Euclidean distance to the goal; it is easy to show that this is a consistent and admissible function. This ensures that the A* algorithm computes a *shortest* path through the graph.

Alternatively, one can use other types of non-negative edge costs to obtain paths that are optimal based on other criteria. For instance, in Chapter 9 we will map *crowd density* information onto the edges to let characters prefer routes that are less crowded.

PART II

Navigation Meshes

4 The Explicit Corridor Map in 2D Environments

In this chapter, we describe the *Explicit Corridor Map (ECM)* navigation mesh for 2D environments, and we show how the ECM can be used for flexible and efficient path planning and crowd simulation.

This chapter is based on the following publications:

- W. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. The Explicit Corridor Map: A medial axis-based navigation mesh for multi-layered environments. arXiv:1701.05141, 2017. In submission to a journal. [148]
- W. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. The Explicit Corridor Map: Using the medial axis for real-time path planning and crowd simulation. In *International Computational Geometry Multimedia Exposition*, pages 70:1–70:5, 2016. [147]
- W. van Toll, N. Jaklin, and R. Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN / ICT.OPEN (ASCI track)*, 2015. [151]

4.1 Introduction

In Part I of this thesis, we explained that path planning and crowd simulation are important research topics with many useful applications. A *navigation mesh* efficiently subdivides the walkable space of a virtual environment into polygonal cells. As such, it is more flexible for crowd simulation than a graph, and it is a more efficient representation of the environment than a grid.

This chapter presents the *Explicit Corridor Map* (ECM), a navigation mesh that forms the basis of many other chapters in this thesis. The ECM is the *medial axis* of a 2D environment annotated with nearest-obstacle information. Similarly to the medial axis, the ECM has a storage complexity of $\mathcal{O}(n)$ and it can be constructed in $\mathcal{O}(n \log n)$ time for an environment with n obstacle vertices.

The ECM can be used to plan paths for disk-shaped characters of any radius, which is typically not possible when using various other subdivisions into polygons. It also supports various operations that are important for crowd simulation, such as finding the nearest static obstacle to a query point. Furthermore, other chapters will show that the ECM is well-defined for multi-layered 3D environments (Chapter 5),

that it supports real-time dynamic updates (Chapter 6), and that it can be used to simulate large crowds of heterogeneous characters in real-time (Chapter 10).

The ECM was first introduced by Geraerts [33]. Compared to this first publication, we give more formal definitions of a 2D environment, the medial axis, and the ECM. We also describe a range of geometric operations on the ECM that are useful for path planning and crowd simulation. Furthermore, we thoroughly discuss and compare three different implementations of the ECM.

The remainder of this chapter is structured as follows:

- Section 4.2 provides definitions of a 2D environment, its medial axis, and its Explicit Corridor Map.
- In Section 4.3, we describe various operations on the ECM that are useful for path planning and crowd simulation.
- Section 4.4 explains how we have implemented the ECM using various Voronoi diagram libraries.
- In Section 4.5, we use our implementation to efficiently compute the ECM and perform geometric operations in a range of environments.
- Section 4.6 concludes the chapter and discusses the advantages and limitations of the ECM.

4.2 Definitions

In this section, we give a formal definition of two-dimensional virtual environments and of the ECM navigation mesh. We also analyze the asymptotic size and construction time of the ECM.

4.2.1 2D Environment

We define a *2D environment* $\mathcal{E}$ as a bounded subset of the two-dimensional plane, with closed polygonal obstacles. The *obstacle space* $\mathcal{E}_{obs}$ is the union of all obstacles, including the boundary of the environment. The complement of $\mathcal{E}_{obs}$ is the *free space* $\mathcal{E}_{free}$. An example of a 2D environment is shown in Figure 4.1a.

Let n be the number of vertices required to define $\mathcal{E}_{obs}$ using interior-disjoint simple polygons, line segments, and points. We call n the *complexity* of $\mathcal{E}$.

4.2.2 Medial Axis

In Section 3.1, we have briefly described the Voronoi diagram and the medial axis. There are multiple definitions of the medial axis, each with slightly different details. We therefore give our own definition; it is comparable to the definitions by Preparata and Lee [93, 124] but applied specifically to 2D environments.

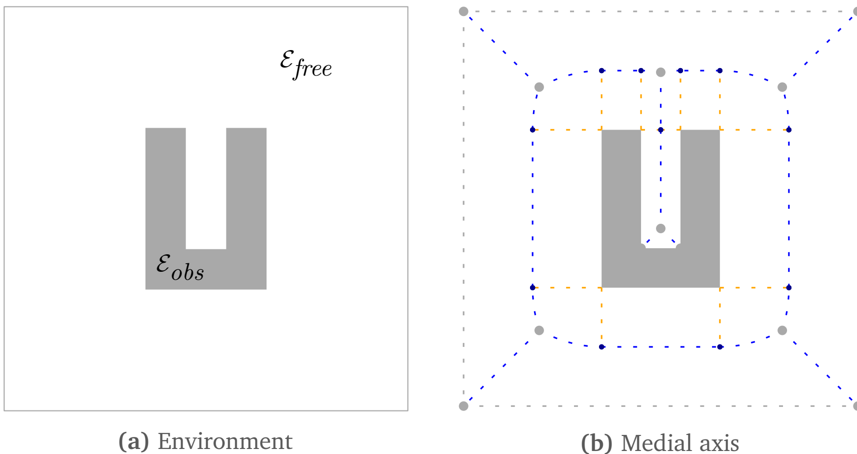
Definition 4.1 (Medial axis, 2D). *For a 2D environment $\mathcal{E}$, let $ma(\mathcal{E})$ be the set of all points in $\mathcal{E}_{free}$ that have at least two distinct equidistant nearest points in $\mathcal{E}_{obs}$, in terms of 2D Euclidean distance. The medial axis $MA(\mathcal{E})$ is the topological closure of $ma(\mathcal{E})$.*

Figure 4.1b shows the medial axis of an example environment. Since the medial axis is a pruned Voronoi diagram, it forms a plane graph (a planar graph embedded in 2D). The term 'closure' ensures that degree-1 vertices (at concave obstacle corners) are also part of the medial axis.

The main difference to the GVD is that a point on the medial axis requires two *distinct points* as nearest obstacles; it does not matter from which 'sites' these points originate. Therefore, in Figure 4.1b, the medial axis does not run into the convex corners of the U-shaped obstacle, whereas such edges typically do appear in a GVD of line segment sites.

Each medial axis arc A is the bisector of two *generators*: the endpoints or segments of $\mathcal{E}_{obs}$ that are nearest to A. If one generator is a line segment and the other is a point, then A is a parabolic arc; otherwise, A is a line segment.

In this chapter, we refer to all vertices of degree 1, 3, or higher as *true vertices*. We refer to degree-2 vertices as *semi-vertices* because the medial axis only changes its shape at these points. Observe from Figure 4.1b that a semi-vertex occurs when the medial axis crosses a normal vector at a convex obstacle corner. We define an *edge* as a sequence of medial axis arcs between two true vertices.



(a) Environment

(b) Medial axis

Figure 4.1: A simple 2D environment. **(a)** The obstacle space $\mathcal{E}_{obs}$ (shown in gray) consists of line segments and polygons. Its complement is the free space $\mathcal{E}_{free}$. **(b)** The medial axis (shown in blue) is a graph through $\mathcal{E}_{free}$. True vertices are shown as large dots. Semi-vertices (small dots) occur when a bisector's generator changes. This is indicated by dashed orange line segments, which are not part of the graph.

4.2.3 Explicit Corridor Map

The Explicit Corridor Map is a graph representation of the medial axis annotated with nearest-obstacle information. It describes each medial axis arc and its surrounding free space in an efficient manner. As such, it is a compact navigation mesh that can be used to find paths for characters of any radius.

Definition 4.2 (Explicit Corridor Map, 2D)**.** *For a 2D environment $\mathcal{E}$ with obstacles, the Explicit Corridor Map ECM($\mathcal{E}$) is an extended representation of the medial axis MA($\mathcal{E}$) as an undirected graph $G = (V, E)$.*

- *V is the set of true vertices of the medial axis.*
- *E is the set of edges of the medial axis.*
- *Each edge $e_{ij} \in E$ represents the medial axis arcs between two true vertices $v_i, v_j \in V$. It is represented by a sequence of $n' \geq 2$ bending points*[1] *$bp_0, \ldots, bp_{n'-1}$ where $bp_0 = v_i$, $bp_{n'-1} = v_j$, and $bp_1, \ldots, bp_{n'-2}$ are the remaining semi-vertices along the edge.*
- *Each bending point is a medial axis vertex annotated with nearest-obstacle information. A bending point bp_k on an edge stores its two nearest obstacle points l_k and r_k on the left and right side of the edge.*

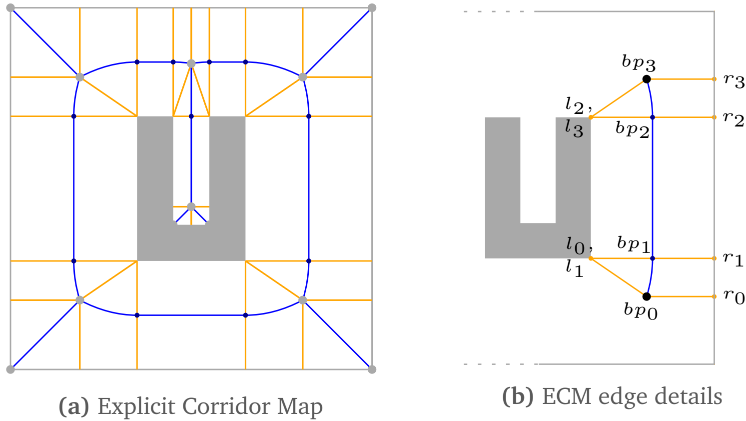
Figure 4.2a shows the ECM of our example environment. Since the ECM is an undirected graph, any edge e_{ij} could also be described as an edge e_{ji}, with the list of bending points reversed and all left and right obstacle points swapped. Furthermore, a true vertex occurs as the first or last bending point for each of its incident edges. Each such bending point has its own sense of left and right; together, they store all nearest obstacle points for the true vertex. Thus, it is sufficient to store only two obstacle points for each bending point. We also emphasize that the orange line segments in Figure 4.2a are *not* graph edges; they merely denote the relation between bending points and their nearest obstacles. Figure 4.2b shows the details of an ECM edge.

Annotating the medial axis with nearest-obstacle information has many advantages. One advantage is that the *clearance* (the distance to the nearest obstacle) is known at each bending point. This enables path planning for characters of any radius; that is, we do not have to inflate the obstacles using Minkowski sums for a particular radius. Another advantage is that these annotations subdivide $\mathcal{E}_{free}$ into non-overlapping polygonal cells. Section 4.3 will show that these cells are useful for point location, path planning, and crowd simulation.

4.2.4 Complexity of the Explicit Corridor Map

Theorem 4.1. *The ECM of a 2D environment with complexity n can be computed in $\mathcal{O}(n \log n)$ time and requires $\mathcal{O}(n)$ space.*

[1] Geraerts originally referred to bending points as *event points*. [33]



(a) Explicit Corridor Map

(b) ECM edge details

Figure 4.2: The Explicit Corridor Map of a 2D environment. **(a)** The ECM is a medial axis with nearest-obstacle annotations, shown as orange line segments between vertices and their nearest obstacle points. These segments are not edges in the graph. **(b)** Details of an ECM edge with four bending points. Each bending point bp_i stores its position p_i and its nearest obstacle points l_i and r_i.

Proof. The generalized Voronoi diagram (GVD) of non-crossing line segments with n distinct endpoints has $\mathcal{O}(n)$ vertices and arcs and can be computed in $\mathcal{O}(n \log n)$ time [5, 7]. The medial axis (MA) has the same asymptotic size because it is a pruned GVD. It can be obtained from the GVD without increasing the overall asymptotic running time [20, 83].

The ECM converts vertices to bending points by adding nearest-obstacle annotations. These can easily be added during the construction algorithm in constant time per bending point. After all, a medial axis arc cannot be computed without knowing its generators, which are exactly the nearest obstacle points in the ECM.

What remains to be analyzed is the total number of bending points. Each MA vertex of degree 1 or 2 occurs as a bending point exactly once. A vertex of degree $d \geq 3$ occurs as a bending point d times: once for each edge that contains this vertex as an endpoint. Since the sum of all vertex degrees is $\mathcal{O}(n)$, there are $\mathcal{O}(n)$ bending points in total, each of which requires $\mathcal{O}(1)$ storage.

Thus, the ECM adds a linear amount of information to the medial axis in linear time. The total construction time remains $\mathcal{O}(n \log n)$ and the storage size remains $\mathcal{O}(n)$. □

4.3 Operations and Applications

This section defines a number of operations on the ECM and explains how they can be used for path planning. Chapter 10 will provide more information on how these concepts fit into a generic crowd simulation framework.

4.3.1 Point Location

The ECM event points and their nearest-obstacle annotations partition the free space $\mathcal{E}_{free}$ into $\mathcal{O}(n)$ non-overlapping polygonal cells. The *ECM cell* for two subsequent bending points bp_i and bp_{i+1} on an ECM edge is the polygon bounded by the counterclockwise sequence of points $[p_i, r_i, r_{i+1}, p_{i+1}, l_{i+1}, l_i]$. Note that some points on this boundary can coincide.

We can perform a *point-location query* to find out in which ECM cell a query point is located. There are several data structures that can answer point-location queries in $\mathcal{O}(\log n)$ time and that require $\mathcal{O}(n)$ space [7, 136]. Once the cell containing a query point p has been determined, the *nearest obstacle point* $np(p)$ to p is guaranteed to lie on the boundary of this cell. This point can be computed in constant time because each cell has constant complexity. Thus, in a crowd simulation application, we can easily determine how far each character is removed from the nearest boundary. This is a special advantage of ECM cells; arbitrary subdivisions into cells do not have this property in general.

4.3.2 Retraction

Points in the free space can be *retracted* onto the medial axis. In robot motion planning, the term 'retraction' is used for a function that maps points in $\mathcal{E}_{free}$ onto the medial axis [164], as well as for complete planning methods based on this principle [110]. We use the following definition:

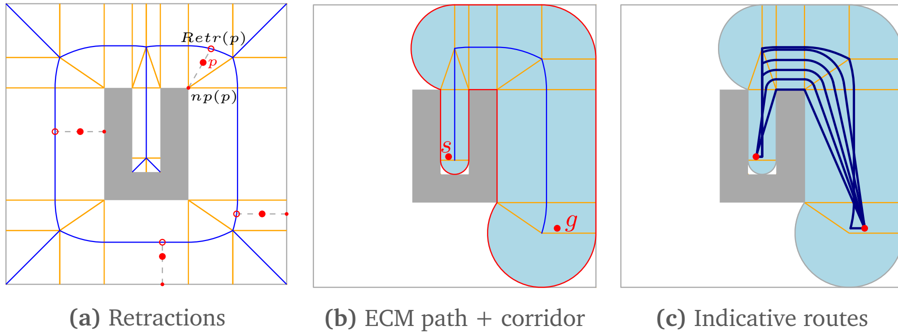
Definition 4.3 (Retraction). *For any point p in the free space $\mathcal{E}_{free}$, the* retraction *Retr(p) is a unique projection of p onto the medial axis.*

1. *If p lies on the medial axis, then Retr$(p) = p$.*
2. *If p does not lie on the medial axis, let l be the half-line that starts at $np(p)$ and passes through p. Retr(p) is the first intersection of l with the medial axis.*

Figure 4.3a shows examples of retractions. Observe that *Retr*(p) always lies in the same ECM cell as p because the half-line l is defined such that the nearest obstacle does not change along l before intersecting the medial axis. Therefore, a retraction is easy to compute using the ECM: it can be found in $\mathcal{O}(\log n)$ time by using a point-location query followed by constant-time geometric operations. Mapping points of $\mathcal{E}_{free}$ onto the medial axis is a useful operation for path planning.

4.3.3 Computing ECM Paths

To plan a path for a disk-shaped character in the ECM, we first find a path along the medial axis that has sufficient clearance. This is equivalent to the *retraction method* for motion planning [110]: given a start position s and a goal position g in $\mathcal{E}_{free}$, we compute their retractions, and then we compute an optimal path on the medial axis from *Retr*(s) to *Retr*(g) using the A* search algorithm. This search is efficient because the medial axis is a sparse graph compared to e.g. a grid.



(a) Retractions **(b)** ECM path + corridor **(c)** Indicative routes

Figure 4.3: **(a)** Examples of query points (shown in red), their nearest obstacle points (small dots), and their retractions (circles). **(b)** Given two positions s and g, the retraction method is used to compute a path from s to g along the medial axis. A corridor describes the free space around this path. **(c)** Within the corridor, we can compute various types of indicative routes, e.g. with an amount of preferred clearance.

The clearance information stored in each ECM bending point allows us to precompute the *minimum clearance* along each edge. The search can then skip edges for which the clearance is too low for our disk to pass through. This yields the shortest path along the medial axis with the required clearance.

The free space around a medial axis path can be described using a *corridor*, which is the sequence of ECM cells along the path combined with the maximum-clearance disks at its ECM vertices [33]. Figure 4.3b shows an example.

4.3.4 Computing Indicative Routes

An ECM path can be converted into an *indicative route* for the character to follow. Various types of indicative routes can be obtained $\mathcal{O}(m)$ time, where m is the number of ECM cells along the path.

For instance, we can use a funnel algorithm to obtain the *shortest path* within a corridor, while keeping a preferred distance to obstacles whenever possible [33]. Examples are displayed in Figure 4.3c. In our implementation, indicative routes are piecewise linear curves, and the circular arcs of Figure 4.3c are approximated. It is also easy to compute indicative routes that stay on the left or right side of the free space (or any interpolation of these extremes), by choosing an appropriate intermediate point on each of the orange line segments in the corridor. Varying the 'side preference' among characters is a convenient way to obtain diversity in the crowd.

4.3.5 Computing Visibility Polygons

The ECM can also be used to compute the *visibility polygon* $V(p)$ of a query point $p \in \mathcal{E}_{free}$, i.e. the set of all points in $\mathcal{E}_{free}$ that are visible from p. This is useful in

crowd simulations: it allows us to model what a character can see at a particular point in time. In Chapter 8, we will use it to let characters re-plan their paths when they see a dynamic obstacle.

There are many ways to compute visibility polygons in general [35], but an efficient local algorithm can be used when $\mathcal{E}_{free}$ has been subdivided into cells whose vertices lie on obstacles [129]. The ECM fulfills this condition, as do various other navigation meshes [67, 102, 114]. We now briefly describe this local algorithm. Chapter 5 will show that it also applies to *multi-layered* environments.

The visibility algorithm is outlined in Figure 4.4a. To compute $V(p)$, we start in the ECM cell that contains the query point p; this cell is entirely visible from p. From there, we move to adjacent cells while tracing the boundary of $V(p)$. Each ECM cell that we visit has a nearest obstacle (a point or a line segment) on the left and right side. The parts of these obstacles that are visible from p are added to the boundary of $V(p)$. These visibility checks can be performed in constant time by keeping track of the angular range of points that p can still see since the last cell that we visited. At each ECM vertex, the ECM 'splits' into multiple edges, each with their own nearest obstacles on the left and right side. Therefore, at a vertex, the angular range of visible points is split into multiple parts (one for each edge), and we recursively compute the parts of $V(p)$ in these sub-ranges. A subroutine stops when the range of visible points becomes empty. This way, we eventually trace the boundary of $V(p)$.

The complexity of this algorithm depends on the total number of visits to ECM cells. A cell can be visited multiple times if the algorithm can reach it by passing an obstacle on the left *and* on the right. In an unlucky environment, $\mathcal{O}(n)$ ECM cells may be visited $\mathcal{O}(n)$ times, which implies a worst-case running time of $\mathcal{O}(n^2)$. However, in many cases, a cell will be visited only a constant number of times. More importantly, the algorithm is *local*: by starting at a visible ECM cell and discovering adjacent visible cells, we do not visit cells that are not visible. Therefore, visibility queries are very fast in practice, as we will show in Section 4.5.3.

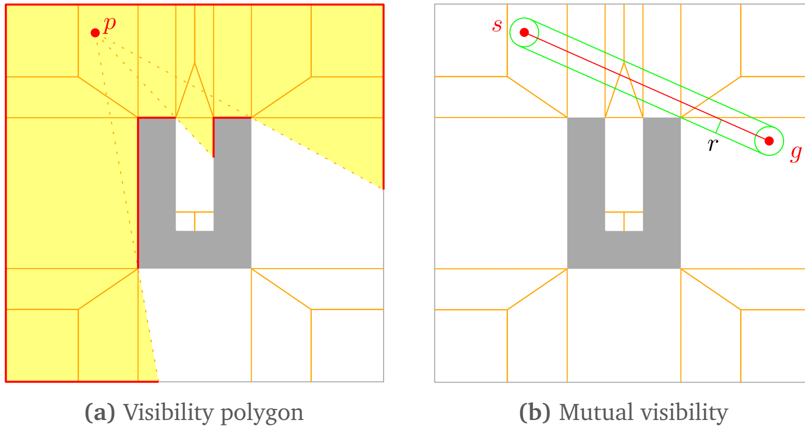
4.3.6 Checking for Mutual Visibility with Clearance

We can also use the ECM to detect whether two query points s and g are mutually visible, i.e. whether the line segment $\overline{sg}$ intersects any obstacles. To compute this, we start at the ECM cell that contains s and then trace $\overline{sg}$ while moving from cell to cell. If g is reached without intersecting a cell boundary that corresponds to an obstacle, then s and g are mutually visible. This technique can also be applied to other navigation meshes.

An advantage of using the ECM is that this query can be extended to compute whether $\overline{sg}$ is collision-free for a *disk*. If we know where $\overline{sg}$ enters and exits a particular cell, we can compute the minimum distance from $\overline{sg}$ to the obstacles of this cell in constant time. For all cells combined, this yields the overall minimum distance of $\overline{sg}$ to obstacles. (This does not work for arbitrary navigation meshes,

in which the cell boundaries do not necessarily represent *nearest* obstacles.)

If the overall minimum distance is d, we know that disks of radius $\leq d$ can move from s to g in a straight line without intersecting any obstacles, and that disks of radius $> d$ cannot. Figure 4.4b shows an example. Jaklin et al. [62] use this technique for a path following method, MIRAN, which needs to know which points on an indicative route can be reached in a straight line by a disk-shaped character.



(a) Visibility polygon

(b) Mutual visibility

Figure 4.4: Visibility queries in the ECM. For clarity, the boundaries of ECM cells are shown, but the medial axis is not. **(a)** We can compute the visibility polygon $V(p)$ of a point p. The boundary parts of $V(p)$ generated by obstacles are shown in black. **(b)** We can compute whether two points s and g are mutually visible, and we can find the maximum value r such that the line segment $\overline{sg}$ (shown in red) is collision-free for a disk of radius r.

4.4 Implementation

Our implementation of the Explicit Corridor Map is the basis of a general crowd simulation framework. This framework will be the topic of Chapter 10. The software was written in C++ in Visual Studio 2013.

In this section, we describe how our implementation computes the ECM. This is useful to discuss because the Voronoi diagram can be constructed in various ways.

4.4.1 Computing the ECM using Voronoi Diagram Libraries

To compute the ECM, we have integrated two different libraries for computing Voronoi diagrams: Vroni [48] and a package of Boost [14]. Both libraries can compute a Voronoi diagram in $\mathcal{O}(n \log n)$ time: Vroni uses randomized incremental construction [49], and Boost uses a plane sweep algorithm [30].

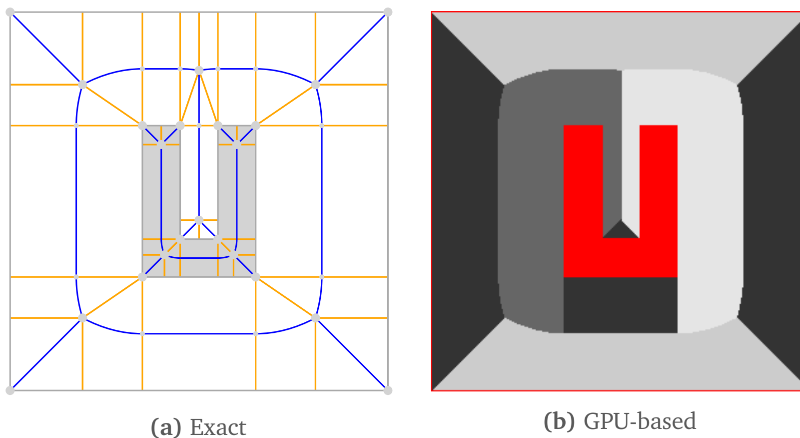
Since the Boost Voronoi library requires integer coordinates as input, we multiply all coordinates by $10{,}000$ and round them to the nearest integer. For

convenience, we use these rounded coordinates in Vroni as well. We use meters as units, so this scaling implies that we represent all coordinates within a precision of 0.1 millimeters.

Both Vroni and Boost assume that the input sites are interior-disjoint line segments. In practice, environments are often drawn by hand and may contain overlapping geometry. Therefore, before computing the ECM, we use another component of Boost to convert obstacles to interior-disjoint segments, using the scaled integer coordinates described earlier. After computing the ECM of these line segments, we remove all graph components that lie inside the original obstacles. These steps will be included in our time measurements in Section 4.5. Figure 4.5a outlines the implementation for an example environment.

4.4.2 Computing the ECM by Rendering

It is also possible to compute an *approximation* of the medial axis and ECM by using the graphics card [33], based on the work of Hoff et al. [57]. This algorithm requires a subdivision of $\mathcal{E}_{obs}$ into convex obstacles. It lets each obstacle generate a 3D shape (a *distance mesh*) in a unique color, and it then renders all shapes using an orthographic top view. The borders between pixels of different colors roughly correspond to points of the Voronoi diagram. The accuracy of this approximation depends of the size of the image that is rendered. An example of a rendered result is shown in Figure 4.5b.



(a) Exact **(b)** GPU-based

Figure 4.5: Two ways to generate the ECM. **(a)** External libraries such as Boost and Vroni can compute the Voronoi diagram of a set of line segments. When converting this to an ECM, we need to remove graph components that lie inside the original obstacles. **(b)** Alternatively, the Voronoi diagram can be approximated using rendering techniques.

Over time, this graphics-based approach has proven to be less practical than exact implementations for various reasons:

- The graphics card imposes a maximum rendering resolution. To allow higher resolutions, we have developed an algorithm [13] that subdivides an image into tiles that are small enough to render. The ECM is computed for each tile separately, and the results are then merged into a single graph. This works well, but the extra operations add to the complexity of the algorithm.
- The running time scales quickly as the resolution increases. To reduce this effect, the GPU can be used to perform many operations in parallel [13], but the overall resolution still has a large impact on the running time.
- It is generally not clear which resolution is required to achieve sufficiently accurate results. Different environments may require different resolutions, depending on their level of detail. This unpredictable precision is particularly problematic in *multi-layered environments* (MLEs), which we will discuss in Chapter 5. Constructing the ECM of an MLE requires a high level of precision that cannot be predicted in advance.
- The algorithm is not robust against special cases such as obstacles that share a vertex. Such cases have unpredictable effects on the rendered image; it is difficult to recognize and solve all possible effects.
- Each machine and graphics card may render the image in a slightly different way, which makes results hard to reproduce.

Due to these disadvantages, the implementations based on Boost and Vroni are preferred. For more details of the GPU-based method, we refer the interested reader to the appropriate publications [13, 33].

4.4.3 Point-Location Data Structure

In theory, point-location queries can be answered in $\mathcal{O}(\log n)$ time [7, 136]. In our implementation, we use a different approach: we create a grid with cells of 10×10 meters, in which each grid cell c_g stores a reference to all ECM cells for which the axis-aligned bounding box overlaps with c_g. To perform a point-location query for a point q, we find the grid cell c_q that contains q in constant time, and we iterate over all ECM cells stored in c_q until we find the ECM cell that actually contains q. Although this approach leads to slower queries if a grid cell has many associated ECM cells, the implementation has proven to be fast in our test environments.

We use a grid rather than a theoretically optimal data structure because it is easier to implement robustly, and because it is easier to update when ECM cells are added or removed during a simulation (which will be the case in Chapter 6).

4.5 Experiments and Results

This section assesses the performance of our ECM implementations in a range of virtual environments. These environments are shown in Figure 4.6. More details

of the environments can be found in Table 4.1.

Military is a simple environment with a small number of obstacles. *City* is a more complex virtual city. *Zelda* is an environment from a computer game. *Zelda2x2*, *Zelda4x4*, and *Zelda8x8* are adapted versions of *Zelda* that have been duplicated in a 2×2, 4×4, and 8×8 grid pattern.

We have chosen these particular environments because they reflect a range of complexities. Chapter 5 will show various *multi-layered* environments. Chapter 7 will contain more 2D and multi-layered examples.

4.5.1 Computing the ECM

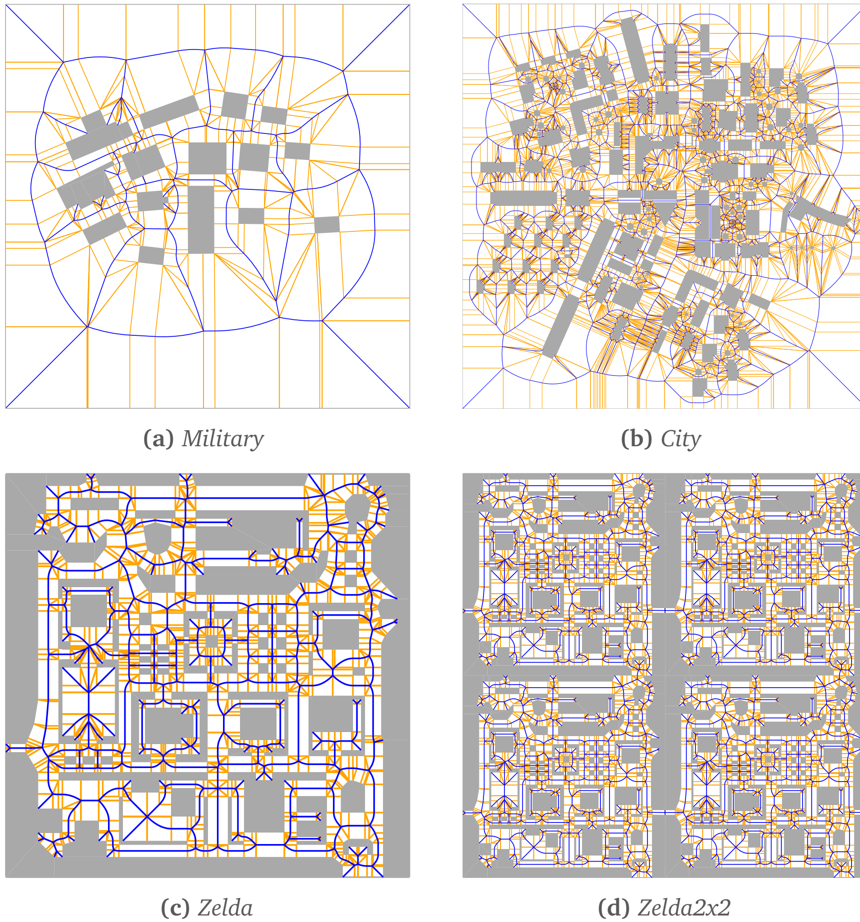
We first computed the ECM for all test environments. For each environment, Table 4.1 shows the complexity of the ECM (i.e. the total number of vertices, edges, and bending points), as well as the construction time for each of the three implementations (Vroni, Boost, and GPU). We used a single CPU core in all implementations.

For the GPU-based method, we only use the standard graphics-based implementation [33] and not the version with tiling and parallel computations [13]. We experimented with two different rendering resolutions: 5 pixels per meter (ppm) and 20 ppm. For example, for an environment of 100×100 meters large, a resolution of 5 ppm corresponds to an image of 500×500 pixels. A resolution of 5 ppm is usually sufficient to obtain a reasonably precise ECM; 20 ppm results in a more detailed ECM at the cost of slower construction. For this experiment, choosing a fixed precision is more logical than choosing a fixed image size because not all environments are equally large.

The complexity of the ECM is slightly different for each implementation. The GPU version typically yields less complex graphs because it does not recognize all of the environment's details. Furthermore, Vroni and Boost handle degenerate cases such as degree-4 vertices differently. In our remaining experiments, we will use the ECMs that were computed using Boost; therefore, the complexities reported in Table 4.1 were taken from the Boost version.

Table 4.2 shows that our Vroni-based implementation was faster than the Boost-based implementation in all environments. For the most complex 2D environment, *Zelda8x8*, computing the ECM took just under 1 second when using Vroni. Hence, even complex ECMs can be computed quickly. This allows the navigation mesh to be generated interactively (e.g. when loading a game level, or when used in a tool for designing environments).

At a resolution of 5 pixels per meter, the GPU-based implementation was always slower than Vroni and usually slower than Boost. At a high resolution of 20 ppm, the implementation was the slowest in all environments. *Lowering* the resolution can improve the running times in exchange for a less accurate ECM, but we will not explore this further. As mentioned before, the Vroni and Boost implementations have many important advantages over the GPU-based version.



(a) *Military* **(b)** *City* **(c)** *Zelda* **(d)** *Zelda2x2*

Figure 4.6: The test environments and their ECMs. Obstacles are shown in gray, the medial axis is shown in blue, and nearest-obstacle annotations are shown in orange. *Zelda4x4* and *Zelda8x8* are not shown because they are very large and structurally similar to *Zelda2x2*.

Environment	**Geometry**		**ECM complexity**		
	#Obstacle vertices	Size (m)	#Vertices	#Edges	#BPs
Military	108	200 × 200	56	71	288
City	2102	500 × 500	1442	1621	6306
Zelda	564	100 × 100	296	351	1258
Zelda2x2	2304	200 × 200	1184	1408	5082
Zelda4x4	9180	400 × 400	4720	5624	20329
Zelda8x8	36684	800 × 800	18848	22480	81365

Table 4.1: Details of the test environments and their ECMs. The *Geometry* columns show the number of obstacle vertices and the physical width and height of the environment (in meters). The *ECM complexity* columns show the number of vertices, edges, and bending points (BPs) in the ECM computed using Boost.

4.5.2 Computing Paths and Indicative Routes

In each environment, we computed indicative routes between $10{,}000$ pairs of random start and goal points. A random point was chosen by uniformly sampling in the environment's bounding box until an obstacle-free point was found. For each query pair (s, g), we computed the shortest path between *Retr*(s) and *Retr*(g) on the medial axis using A* search, with the 2D Euclidean distance to *Retr*(g) as a heuristic. We then converted this path to a short indicative route with a preferred distance of 0.5 m to obstacles, using the algorithm described by Geraerts [33]. Figure 4.7a shows examples of indicative routes in the *City* environment.

The second and third columns of Table 4.3 show the average running times of these path planning queries per environment. The 'Path only' column denotes the time (in milliseconds) for performing A* search to obtain a path on the medial axis. The 'Path + IR' column denotes the total time for computing the medial axis path and the indicative route.

The running time depends heavily on the complexity of the resulting path; this explains the high standard deviations. It can be seen that queries require only a few milliseconds on average in the most complex environments. Thus, the ECM allows real-time path planning for large crowds of characters with individual goals.

In complex environments such as *Zelda8x8*, the advantage of a sparse graph over a grid becomes clear. A high grid resolution would be required to capture all details, so the storage requirements would be higher, and path planning would be slower.

4.5.3 Computing Visibility Polygons

Next, we computed the visibility polygons of $10{,}000$ random query points in each environment. The running times are reported in the fourth column of Table 4.3.

As explained in Section 4.3.5, the performance of a visibility query depends on the number of ECM cells visited during the query. Thus, the running time depends on the environment's complexity around a query point. This explains why the algorithm does not become slower for the larger variants of *Zelda*. In most environments, a query took around 0.05 milliseconds on average. In the *City* environment, the average running time was higher (0.15 ms) because this environment features large open spaces surrounded by many obstacles. Still, the algorithm is clearly fast enough for real-time visibility computations during a simulation. Figure 4.7b shows examples of visibility polygons in the *City* environment.

4.6 Conclusions and Future Work

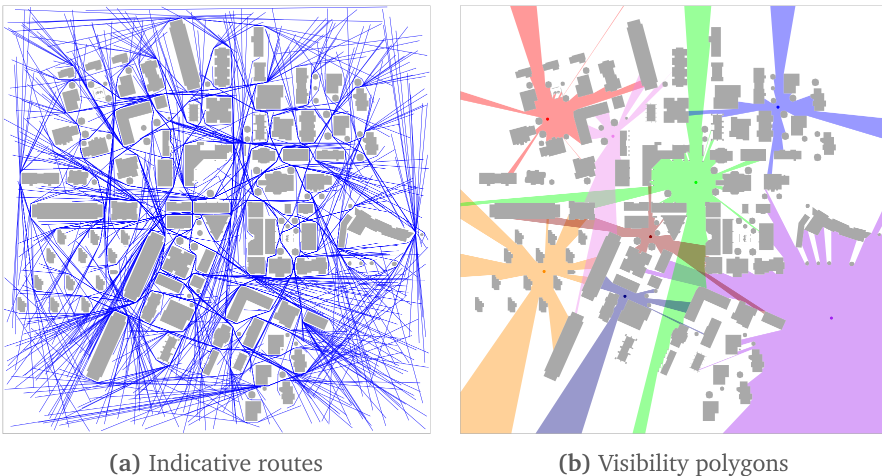
In this chapter, we have formally defined the Explicit Corridor Map (ECM) [33] as a navigation mesh based on the 2D medial axis. The ECM enables path planning for disk-shaped characters of any radius. It supports efficient geometric operations such as retractions, nearest-obstacle queries, visibility queries, and the computation

Environment	ECM time (ms)			
	Vroni	Boost	GPU (5)	GPU (20)
Military	3.5 [0.1]	6.7 [0.1]	37.3 [2.0]	407.2 [35.7]
City	70.9 [0.3]	130.0 [0.7]	300.5 [4.3]	3221.5 [32.0]
Zelda	14.9 [0.1]	23.0 [0.2]	25.5 [0.8]	124.0 [2.7]
Zelda2x2	59.2 [0.5]	92.0 [0.6]	81.4 [2.4]	541.3 [19.6]
Zelda4x4	235.4 [2.4]	367.7 [1.6]	384.7 [9.7]	3110.5 [18.0]
Zelda8x8	997.4 [5.4]	1529.1 [3.8]	3000.0 [17.1]	29304.1 [32.6]

Table 4.2: Construction times of the ECM. The *ECM time* columns show the ECM construction time for each implementation. 'GPU (5)' and 'GPU (20)' refer to the GPU-based implementation with a precision of 5 and 20 pixels per meter, respectively. All times are in milliseconds and have been averaged over 10 runs. Standard deviations are shown between square brackets.

Environment	Path only	Path + IR	Visibility
Military	0.006 [0.004]	0.06 [0.04]	0.04 [0.02]
City	0.08 [0.07]	0.30 [0.19]	0.15 [0.06]
Zelda	0.02 [0.02]	0.12 [0.06]	0.05 [0.02]
Zelda2x2	0.07 [0.05]	0.27 [0.13]	0.05 [0.03]
Zelda4x4	0.27 [0.23]	0.64 [0.39]	0.07 [0.03]
Zelda8x8	1.18 [1.13]	1.93 [1.50]	0.05 [0.02]

Table 4.3: Results of the experiments for path planning and visibility queries, as described in Section 4.5.2 and Section 4.5.3, respectively. All times are in milliseconds, averaged over 10,000 random queries. Standard deviations are shown between square brackets.



(a) Indicative routes **(b)** Visibility polygons

Figure 4.7: Examples of using the ECM in the *City* environment. **(a)** 500 short indicative routes (shown in blue) with a preferred clearance to obstacles. **(b)** Visibility polygons. Query points are shown in black; their visibility polygons are shown in different colors.

of short paths with preferred clearance to obstacles. For a 2D environment with n obstacle vertices, the ECM has size $\mathcal{O}(n)$ and can be computed in $\mathcal{O}(n \log n)$ time.

We have implemented the ECM using two robust Voronoi diagram implementations (Vroni [48] and Boost [14]) and our own GPU-based implementation. The Vroni and Boost implementations have proven to be more reliable and more efficient than the GPU-based one. Our software can compute the ECM efficiently and robustly for large 2D environments. This software forms a solid basis for many chapters of this thesis. Chapter 10 will show how the ECM and its implementation can be used for real-time crowd simulation.

A potential disadvantage of the ECM is that the Voronoi diagram is a relatively complicated data structure. One might argue that the ECM is a less intuitive representation of the walkable space than e.g. a grid [95] or a triangulation [66]. For the same reason, the best way to implement the ECM construction algorithm is to use an existing Voronoi diagram library; the ECM cannot easily be implemented from scratch without relying on external software. Howevever, in exchange for its somewhat complicated nature, the ECM offers a number of advantages over other navigation meshes, as explained in this chapter.

A drawback of navigation meshes in general is that the shortest path in the path planning graph (in our case: the medial axis) is not necessarily homotopic to the shortest path in the entire environment. Therefore, even a shortened path as described in Section 4.3.4 can be longer than the overall shortest path. For future work, we want to analyze path lengths in the ECM and investigate how to improve them. One option is to combine the ECM with a visibility graph; this combination would be comparable to the Visibility-Voronoi Complex [161]. A visibility graph yields shortest paths but has a size of $\mathcal{O}(n^2)$ [35], which is more complex than a navigation mesh.

In the following two chapters, we will extend the ECM to handle multi-layered environments (Chapter 5) and dynamic obstacles (Chapter 6). Next, Chapter 7 will thoroughly compare the ECM to other state-of-the-art navigation meshes.

5 The Explicit Corridor Map in Multi-Layered Environments

In this chapter, we extend the Explicit Corridor Map to a new class of environments, *multi-layered environments* (MLEs), which consist of connected planar components. We extend the medial axis and the ECM to MLEs, we give a construction algorithm, and we test our implementation on a wide range of environments.

This chapter is based on the following publications:

- W. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. The Explicit Corridor Map: A medial axis-based navigation mesh for multi-layered environments. arXiv:1701.05141, 2017. In submission to a journal. [148]
- W.G. van Toll, A.F. Cook IV, and R. Geraerts. Navigation meshes for realistic multi-layered environments. In *Proceedings of the 24th IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3526–3532, 2011. [153]

5.1 Introduction

The Explicit Corridor Map (ECM) from Chapter 4 is a navigation mesh that enables real-time path planning and crowd simulation for disk-shaped characters of any radius. In this chapter, we extend the ECM to 3D environments in which characters are constrained to walkable surfaces, under the assumption that there is a consistent direction of gravity throughout the environment. Section 2.3 has explained that these types of environments have become increasingly popular domains for navigation meshes. Real-world examples include multi-story buildings, train stations, and sports stadiums.

The complete 3D model of an environment is typically too detailed for the purpose of navigation. Therefore, we will introduce a concept called a *walkable environment* (WE) that represents the *free space* of the original 3D environment. It is often useful to subdivide the WE into planar *layers* connected by line segments called *connections*. We will refer to such a subdivision as a *multi-layered environment* (MLE).

We define the medial axis and the ECM for multi-layered environments, based on projected distances on the ground plane. For an MLE with n obstacle vertices and k connections, the MA has size $\mathcal{O}(n)$. We present an algorithm that constructs the MA and ECM in $\mathcal{O}(n \log n \log k)$ time.

The multi-layered ECM has the same properties and applications as in 2D. We perform the experiments from Chapter 4 in a range of MLEs to show that the

ECM supports real-time path planning and crowd simulation in large multi-layered environments. Our formalization of WEs and MLEs will also be used in Chapter 7.

The focus of this chapter lies on the theoretical properties of WEs and MLEs and their medial axes. The remainder of this chapter is structured as follows:

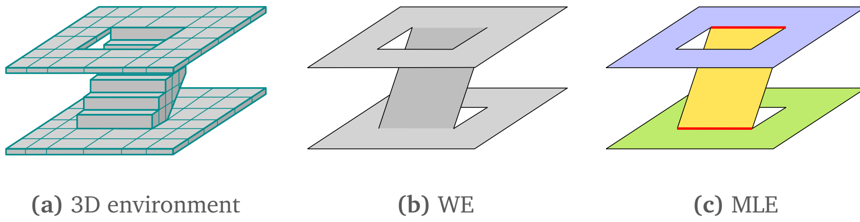
- Section 5.2 defines walkable environments and multi-layered environments.
- Section 5.3 defines the medial axis and the ECM for WEs and MLEs based on a projected distance function.
- In Sections 5.4 to 5.6, we present our construction algorithm for the multi-layered medial axis and ECM.
- Section 5.7 proves that the medial axis of an MLE with n obstacle vertices and k connections can be computed in $\mathcal{O}(n \log n \log k)$ time.
- Section 5.8 outlines our implementation of the multi-layered ECM.
- In Section 5.9, we apply the same experiments as in Chapter 4 to a range of large multi-layered environments.
- Section 5.10 concludes the chapter and suggests directions for future work.

5.2 Definitions of Environments

In this section, we define the types of environments *embedded in 3D* for which we want to construct a navigation mesh: *walkable environments* and *multi-layered environments*. Our main assumption is that there is a consistent direction of gravity $\vec{g}$ throughout the environment. For example, we support multi-story buildings, but not arbitrary 3D surfaces such as spherical planets or Möbius strips. As explained in Chapter 2, this is a common assumption for many navigation meshes [102, 114, 121], and we consider other 3D surfaces [10, 127] to be outside the scope of this thesis.

Let a *3D environment* be a collection of polygons in $\mathbb{R}^3$. These polygons may include floors, ceilings, walls, or any other type of geometry. Figure 5.1a shows a simple example of a 3D environment.

The free space $\mathcal{E}_{free}$ of a 3DE is determined by various parameters that describe on which surfaces a character may walk. Examples include the maximum slope with respect to the gravity direction $\vec{g}$, the maximum height difference between nearby polygons (e.g. the maximum step height of a staircase), and the required vertical distance between a floor and a ceiling.



(a) 3D environment (b) WE (c) MLE

Figure 5.1: A simple 3D environment for which we want to compute a navigation mesh. This figure is equal to Figure 2.3, but we repeat it for convenience. **(a)** The original environment is a collection of polygons in 3D. **(b)** A walkable environment (WE) is a set of polygons along which characters can walk. **(c)** A multi-layered environment (MLE) is a subdivision of the WE into 2D layers. Connections between layers are shown in red.

5.2.1 Walkable Environment

We define a *walkable environment* (WE) as a set of interior-disjoint polygonal surfaces in $\mathbb{R}^3$ on which characters can walk. Thus, a WE is a clean representation of the free space $\mathcal{E}_{free}$ of a 3DE, based on the filtering parameters mentioned earlier. Figure 5.1b shows a simple example of a walkable environment.

A WE can be obtained from a 3DE by filtering out unwalkable parts, e.g. surfaces that are too steep and surfaces along which the ceiling is too low for characters to pass under. Such a filtering process typically also merges polygons that are nearly adjacent; for example, staircases are converted into ramps. As mentioned in Section 2.3, it is common to use voxel-based techniques for this process [24, 102, 114, 121], but alternative methods are also in development [123].

Characters can move from one polygon onto another if these polygons are connected in 3D. The *free space* $\mathcal{E}_{free}$ of a WE is simply the entire set of surfaces. Unlike in the 2D environments from Chapter 4, the obstacle space $\mathcal{E}_{obs}$ of a WE is not intuitively defined, but we will sometimes refer to points on the boundary of $\mathcal{E}_{free}$ as 'obstacle points'.

The WE may consist of multiple connected components: for example, consider two islands with no bridge connecting them. In topological terms, each component is an orientable 2-manifold (a *surface*) with a boundary. This intuitively means that the WE has a 'top' and 'bottom' side, and any point on the bottom side cannot be reached from a point on the top side without intersecting a boundary. Geometrically, we are only interested in the top side, i.e. the floors and not the ceilings. The WE is also what we call *direction-consistent*: slopes are allowed, but there is a single gravity direction for the entire environment. All polygons in the WE have a maximum slope with respect to the *ground plane* P, which is the plane perpendicular to the gravity direction $\vec{g}$. This leads to the following definition:

Definition 5.1 (Walkable environment)**.** *A walkable environment (WE) is a set of interior-disjoint polygons in* $\mathbb{R}^3$ *on which characters can walk. Topologically,*

each connected component of a WE is an orientable 2-manifold with a boundary. Geometrically, the WE is direction-consistent: there exists a horizontal ground plane P *below the WE such that for any non-boundary point* q*, the infinitesimal neighborhood* $\sigma(q)$ *of* q *does not overlap itself when projected vertically down onto* P*.*

It is important to note that the entire WE *can be self-overlapping* when projected onto the ground plane P, i.e. it is not guaranteed that all surfaces are visible from a single top view. This is the main difference to 2D environments, and it strongly influences the construction of navigation meshes: an algorithm for 2D environments cannot easily be applied to WEs in general.

5.2.2 Multi-Layered Environment

A *multi-layered environment* (MLE) is a subdivision of a WE into planar *layers*. Such a subdivision is useful for many purposes, including visualization (each layer can be drawn in 2D), identification (all surface points can be uniquely specified using a 2D position and a layer ID), and the construction of navigation meshes (as the next section will show). Although a single layer does not need to have a particular meaning, a typical example of a layer is one floor of a building.

The layers of an MLE are connected by line segments which we call *connections*. Intuitively, they are the ‘cuts’ that were introduced during the subdivision into layers, and they are the edges along which the layers can be ‘glued together’ to obtain the original WE. Formally, we define an MLE as follows:

Definition 5.2 (Multi-layered environment). *A multi-layered environment (MLE) is a walkable environment (WE) that has been subdivided into* l *planar layers,* $\mathcal{L} = \{L_0, \ldots, L_{l-1}\}$*, using a set* $\mathcal{C} = \{C_0, \ldots, C_{k-1}\}$ *of* k *connections.*

Each layer $L_i \in \mathcal{L}$ *is a set of walkable surfaces that are non-overlapping when projected onto the ground plane* P*. The free space* $\mathcal{E}_{\mathit{free},i}$ *of* L_i *is the union of all polygons in* L_i*. Combining the free space of all layers yields the free space* $\mathcal{E}_{\mathit{free}}$ *of the original WE.*

Each connection $C_q \in \mathcal{C}$ *is a line segment with the following properties:*

- *It lies on the shared boundary of two layers* L_i *and* L_j *(*$i \neq j$*), thus connecting the walkable polygons of these layers.*
- *Its endpoints lie on existing boundary vertices of* $\mathcal{E}_{\mathit{free}}$*, so its endpoints are impassable obstacles.*
- *Its interior is not intersected by any obstacles or by other connections.*

Figure 5.1c shows an example of a multi-layered environment. Note that the MLE is still *embedded in 3D*, but that each individual layer *can* be projected onto P without self-overlap, if desired. Therefore, the projection of a layer L_i onto P is essentially a 2D environment with obstacles as described in Chapter 4. The boundary vertices of these obstacles are also boundary vertices of $\mathcal{E}_{\mathit{free}}$.

Two layers L_i and L_j may be connected through multiple connections at different positions; for example, imagine a bridge that connects to the same layer at both ends. Also, a subdivision into layers is usually not unique: any subdivision that meets the requirements described above is acceptable. As described in Section 2.3 of this thesis, there are several approaches to obtaining such a subdivision [24, 55, 114, 162].

5.2.3 Complexity of a Multi-Layered Environment

The complexity of an MLE is given by the number of connections k and the number of obstacle vertices n in all layers combined. Let n_i be the number of obstacle vertices in a layer L_i. We define n as $\sum_{i=0}^{l-1} n_i$. Note that a vertex occurs in multiple layers if it is an endpoint of a connection. The following lemma bounds the number of connections.

Lemma 5.1. *For any multi-layered environment with l layers and n obstacle vertices, the number of connections k is $\mathcal{O}(n)$.*

Proof. Let n_i be the number of obstacle vertices in a layer L_i. By definition, $n = \sum_{i=0}^{l-1} n_i$. In each layer L_i, the number of connections is bounded by the maximum number of non-intersecting line segments that can be drawn between its n_i vertices. Euler's formula for planar graphs implies that this is $\mathcal{O}(n_i)$. Therefore, the total number of connections is $\mathcal{O}(\sum_{i=0}^{l-1} n_i) = \mathcal{O}(n)$. □

5.3 Definitions of the Medial Axis and ECM

In this section, we define the medial axis and the ECM for walkable and multi-layered environments. Because our definitions do not require a subdivision into layers, they apply to both WEs and MLEs.

5.3.1 Projected Distance

To define the medial axis for walkable and multi-layered environments, we need a notion of distance and path length. We will use the direction-consistency of the WE to define *projected* distances in which height differences are ignored. We acknowledge that this is not the same as the 3D distance on a surface. However, projections are useful and very common for navigation meshes in direction-consistent environments [102, 114, 121].

For two points s and g in a WE or MLE, let $\pi(s, g)$ be a path from s to g through $\mathcal{E}_{free}$ along the walkable surfaces. We define the *projected length* of $\pi(s, g)$ as the curve length of $\pi(s, g)$ when projected vertically onto the ground plane P. This projected path can intersect itself: for instance, consider a path along a spiral staircase.

Let $\pi^*(s, g)$ be a path from s to g with minimal projected length. We define the *projected distance* $d_P(s, g)$ between s and g as the projected length of $\pi^*(s, g)$. That

is, $d_P(s, g)$ ignores any height differences along paths from s to g. Figure 5.2a shows an example of projected distances.

A shortest path $\pi^*(s, g)$ is *unobstructed* if it does not have any bending points around obstacles. The projection of an unobstructed path onto P is a single line segment, so its projected length is simply the 2D Euclidean distance between s and g (when projected onto P). The following properties hold:

Property 5.1 (Straight-line property)**.** *The shortest path $\pi^*(q, n_q)$ from any point $q \in \mathcal{E}_{free}$ to any of its nearest boundary points n_q is unobstructed.*

Property 5.2 (Empty-circle property)**.** *Let q be a point in $\mathcal{E}_{free}$ and let n_q be a nearest boundary point to q, at projected distance $d = d_P(q, n_q)$. For all points $q' \in \mathcal{E}_{free}$ for which $d_P(q, q') \leq d$, the shortest path $\pi^*(q, q')$ is unobstructed. When projected onto P, these points form a disk with radius d.*

5.3.2 Medial Axis

We now define the medial axis based on the function d_P:

Definition 5.3 (Medial axis, multi-layered)**.** *For a walkable or multi-layered environment $\mathcal{E}$ with free space $\mathcal{E}_{free}$, let ma$(\mathcal{E})$ be the set of all points in $\mathcal{E}_{free}$ that have at least two distinct equidistant nearest points on the boundary of $\mathcal{E}_{free}$ with respect to the projected distance function d_P. The medial axis MA$(\mathcal{E})$ is the topological closure of ma$(\mathcal{E})$.*

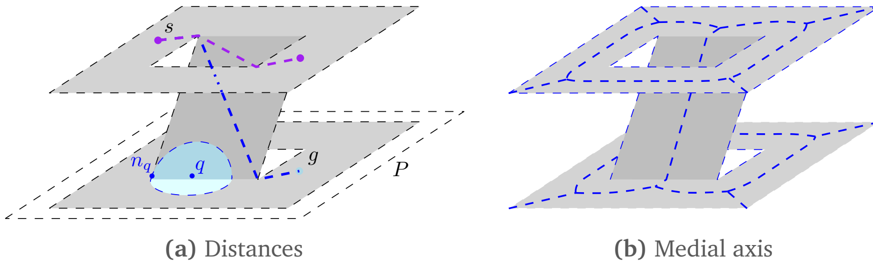
Because the remainder of this chapter is based on the projected distance function, we will often omit the term 'projected' when discussing distances and path lengths.

Figure 5.2b shows the medial axis of an example walkable environment. If an environment $\mathcal{E}$ consists of a single layer L_i, then $MA(\mathcal{E}) = MA(L_i)$. If $\mathcal{E}$ consists of multiple layers, then $MA(\mathcal{E})$ is typically not planar, but intuitively, it is locally similar to a 2D medial axis everywhere due to the straight-line and empty-circle properties. We will use these properties to prove that our construction algorithm for the multi-layered medial axis is correct.

5.3.3 Explicit Corridor Map

The ECM is a graph representation of $MA(\mathcal{E})$ annotated with nearest-obstacle information, exactly as in 2D environments. The nearest obstacles of a medial axis point p may lie in different layers than p itself, but this does not change the ECM's definition. Due to the straight-line property, all ECM annotations are line segments when projected onto P.

All operations and applications of the ECM in 2D (given in Section 4.3) immediately apply to the multi-layered ECM because they are only based on the graph or on the adjacency between ECM cells. The main difference lies in point-location queries: a query point q in an MLE can no longer be specified uniquely as a 2D



(a) Distances **(b)** Medial axis

Figure 5.2: The medial axis of a walkable environment $\mathcal{E}$ is based on path lengths projected onto the ground plane P. **(a)** The shortest path between two points s and g is shown as a bold curve. Its projected length is indicated by the dashed curve. For a non-boundary point $q \in \mathcal{E}_{free}$ with a nearest obstacle point n_q, the set of points in $\mathcal{E}_{free}$ within a distance of $d_P(q, n_q)$ from q is a disk when projected onto P. **(b)** The medial axis $MA(\mathcal{E})$ is drawn on the surfaces of $\mathcal{E}$.

point. Instead, q should be given as a 2D point and a layer ID, or as a 3D point that can be mapped to the appropriate layer. We then use a separate point-location data structure for each layer.

5.4 Construction Algorithm Outline

We now give an outline of our algorithm that computes the medial axis of a multi-layered environment $\mathcal{E}$. The result is also the medial axis of the corresponding *walkable* environment. However, our algorithm makes use of the fact that the *two-dimensional* medial axis is easy to compute. For this reason, we assume that the environment has been partitioned into layers. We acknowledge that this is a necessary pre-processing step that can be solved using other algorithms [54, 55]. Our construction algorithm consists of the following steps:

1. For each individual layer L_i, project L_i onto P and compute its 2D medial axis, while treating all of its connections as *closed* impassable obstacles. This yields exactly the medial axis $MA(L_i)$ according to the projected distance function d_P, but under the assumption that each connection is an obstacle. The result for all layers combined is the medial axis of $\mathcal{E}$ with an extra line segment obstacle for each connection in $\mathcal{C}$. We denote this result by $MA(\mathcal{E}, \mathcal{C})$. The final medial axis will be different because the connections are not supposed to be obstacles.

2. Given $MA(\mathcal{E}, \mathcal{C}')$ with $\mathcal{C}' \subseteq \mathcal{C}$, choose a closed connection $C_q \in \mathcal{C}'$. *Open* the connection by removing its *interior* as an obstacle and repairing the medial axis in its neighborhood. (The endpoints of the connection will remain obstacles because they are on the boundary of $\mathcal{E}_{free}$.) The result is the medial axis of $\mathcal{E}$ in which C_q is no longer an obstacle, i.e. $MA(\mathcal{E}, \mathcal{C}'')$ with

$\mathcal{C}'' = \mathcal{C}' \setminus \{C_q\}$. In Section 5.6, we will describe our algorithm for opening a connection.

3. Repeat step 2 until all connections are open. The result is $MA(\mathcal{E}, \emptyset) = MA(\mathcal{E})$.

In short, we initially treat all connections as closed and then iteratively remove them as obstacles. Opening a connection is essentially the deletion of a line segment Voronoi site [5] but with the extra difficulty that the neighborhood of the deleted site may span multiple layers. We will explain this further in Section 5.6. For now, it is sufficient to know that existing deletion algorithms for Voronoi sites in 2D [1, 25, 81] cannot immediately be applied. Section 5.6 will present an alternative algorithm.

As explained in Chapter 4, adding nearest-obstacle annotations to the medial axis to obtain the ECM is easy because these nearest obstacles are already required to generate medial axis arcs. In the following sections, we will focus on the medial axis only.

5.5 Properties of a Closed Connection

To develop an algorithm for opening a closed connection, we must first study the properties of such a connection. Consider a closed connection between two layers L_i and L_j, as in Figure 5.3a. We will now refer to this connection as C_{ij} to emphasize to which layers it is associated. This notation is not unique because L_i and L_j may be connected via other connections as well. However, in our discussion of opening a single connection chosen by the main algorithm, it is clear to which instance we are referring.

5.5.1 Sides

The connection is currently treated as an impassable obstacle between L_i and L_j. Thus, it occurs as an obstacle for the medial axis on two 'sides'. We define the *side* S_i as the set of all walkable surfaces and boundary points that are currently reachable from C_{ij} by starting in L_i. Likewise, the side S_j consists of all surfaces and obstacle points that can be reached from C_{ij} by starting in L_j. These sides are also annotated in Figure 5.3a.

A side S_i at this point in our algorithm is not necessarily the same as a layer L_i in the environment. The side S_i includes at least the part of L_i that has C_{ij} on its boundary. If other connections are already open, then S_i may contain other layers as well. If sufficiently many connections have been opened such that L_i and L_j are already connected via another route, then S_i and S_j are even equal. However, for our algorithm, it does not matter which layers are already included in S_i or S_j, and it is useful to speak of two different sides of the connection.

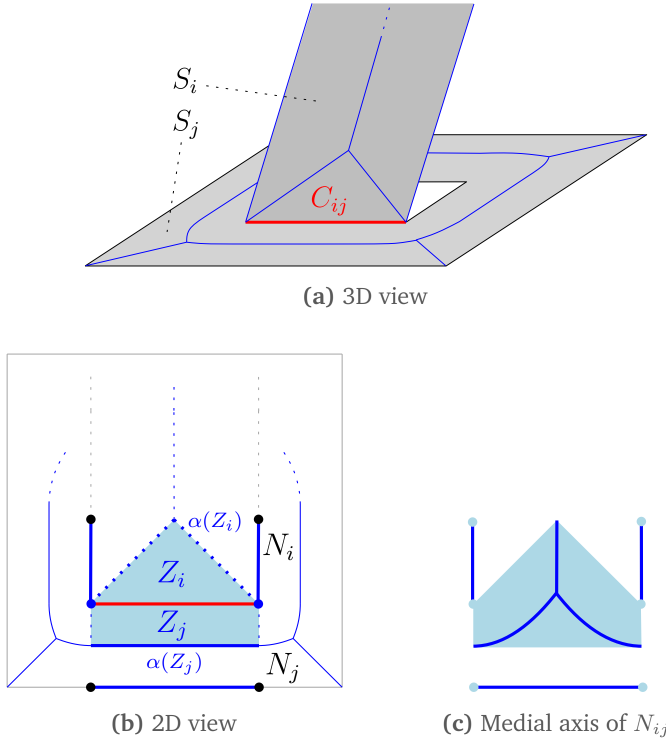


Figure 5.3: Opening a connection C_{ij} in an MLE. **(a)** Initially, C_{ij} is an obstacle on both sides, S_i and S_j. **(b)** 2D top view of the area around C_{ij}. The *influence zone* $Z_{ij} = Z_i \cup Z_j$ is shaded. The obstacle points $N_{ij} = N_i \cup N_j$ that are nearest to Z_{ij} are shown in bold black. **(c)** When opening C_{ij}, the medial axis changes only inside Z_{ij}. This medial axis M_Z is defined by N_{ij}.

5.5.2 Influence Zone

When we open C_{ij}, we effectively remove the *interior* of C_{ij}, denoted by *Int*(C_{ij}), as an obstacle from the environment. We do not remove the endpoints because they will remain obstacles in the WE.

Therefore, we need to determine a new nearest obstacle for all points in the WE that were previously nearest to *Int*(C_{ij}). Let the *influence zone* Z_{ij} be the closure of the set of all points in $\mathcal{E}$ that currently have *Int*(C_{ij}) as a nearest obstacle. Observe from Figure 5.3b that Z_{ij} consists of two parts: one on side S_i and the other on side S_j. (Conceptually, it does not matter if S_i and S_j are already equal.) For convenience, we will refer to these parts as Z_i and Z_j, respectively.

Lemma 5.2. *If the interior of a connection C_{ij} is removed as an obstacle, the medial axis changes only inside the influence zone Z_{ij}.*

Proof. By the definition of Z_{ij}, removing *Int*(C_{ij}) causes the nearest obstacle points

to change only inside (and on the boundary of) Z_{ij}. After all, the other points in $\mathcal{E}_{free}$ were already closer to other obstacles. A consequence is that opening C_{ij} causes the *medial axis* to change only in Z_{ij}. □

Lemma 5.2 implies that $MA(\mathcal{E}, \mathcal{C}')$ (the current medial axis with C_{ij} as an obstacle) and $MA(\mathcal{E}, \mathcal{C}'')$ (the medial axis without C_{ij} as an obstacle, which we want to compute) are equal except in Z_{ij}. It is therefore useful to analyze the shape of Z_{ij}.

Lemma 5.3. *The influence zone Z_{ij} is bounded by the two lines perpendicular to C_{ij} through C_{ij}'s endpoints.*

Proof. (We will refer to these two lines as the *endpoint normals* of C_{ij}.) Consider any point $p \in \mathcal{E}_{free}$ that is *not* between or on the endpoint normals of C_{ij}. Such a point cannot be closest to *Int*(C_{ij}) because it must be closer to an endpoint of C_{ij} or to another obstacle in the environment. Therefore, p cannot be in Z_{ij}. □

Lemma 5.4. *Z_i and Z_j are both bounded by a sequence of medial axis arcs. Both sequences, denoted by $\alpha(Z_i)$ and $\alpha(Z_j)$, are uninterrupted and monotone with respect to the line supporting C_{ij}.*

Proof. We prove the lemma for Z_i; the proof for Z_j is analogous. Z_i is bounded by a set of medial axis arcs $\alpha(Z_i)$ that have a nearest obstacle point on C_{ij}. For every point z on $\alpha(Z_i)$, the nearest point c on C_{ij} can be reached via a line segment $\overline{zc}$ that is perpendicular to C_{ij}. If this were not true, then another obstacle would be in the way and c would not be a nearest obstacle point. Furthermore, c is a nearest obstacle point for all points on $\overline{zc}$ because this nearest obstacle cannot change when moving from z to c. Thus, $\overline{zc}$ lies entirely inside Z_i. Because Z_i consists of infinitely many line segments that all have an endpoint on C_{ij} and that are all perpendicular to C_{ij}, the boundary $\alpha(Z_i)$ is monotone with respect to C_{ij}.

Finally, the definition of an MLE enforces that *Int*(C_{ij}) does not intersect any obstacles. Because of this, every point on *Int*(C_{ij}) has some free space in its neighborhood: for each $c \in$ *Int*(C_{ij}), the line segment $\overline{zc}$ exists and has non-zero length. The endpoints of C_{ij} are the only points where z and c can be equal. This proves that $\alpha(Z_i)$ is a single sequence of arcs. □

These lemmas have the following consequences:

Corollary 5.1. *The boundary of the influence zone Z_{ij} is a single closed loop consisting of $\alpha(Z_i)$, $\alpha(Z_j)$, and the endpoint normals of C_{ij}.*

Corollary 5.2. *The influence zone Z_{ij} can be projected onto the ground plane P without overlap. This projection is a single shape without holes (because it has only one boundary).*

5.5.3 Neighbor Set

Next, we determine which obstacles are required to update the medial axis inside Z_{ij}. On one side S_i of the connection, let N_i be the set of all obstacle points that are nearest to at least one point on $\alpha(Z_i)$, excluding *Int*(C_{ij}) itself. (On the other side S_j, let N_j be defined analogously.) These are the obstacle points that (together with C_{ij}) generate the arcs in $\alpha(Z_i)$. We exclude *Int*(C_{ij}) from N_i because we will be removing this interior as an obstacle. We do explicitly include the *endpoints* of C_{ij} in N_i because these will remain obstacles.

We define the *neighbor set* N_{ij} as the union of N_i and N_j. Informally, this set contains the 'Voronoi neighbors' of the connection. N_{ij} consists of line segments and points on the boundary of $\mathcal{E}_{free}$. These are not necessarily the complete original boundary segments, but only the parts that are actually relevant for Z_{ij}. Note that the neighbors can originate from many different layers and that they can even be (parts of) other connections that are still closed. A neighbor set is illustrated in Figure 5.3b.

We will now prove that N_{ij} contains the obstacle points that define the medial axis in Z_{ij} when *Int*(C_{ij}) is removed. Lemma 5.5 proves that *all* points of N_{ij} are needed; Lemma 5.6 proves that *no other* points are needed.

Lemma 5.5. *When C_{ij} is opened, every obstacle point in N_{ij} is a nearest obstacle for at least one point in Z_{ij}.*

Proof. When the connection is still closed, every point $p \in N_{ij}$ is a nearest obstacle for at least one point z on the *boundary* of Z_{ij}, by definition. When C_{ij} is opened, p will still be nearest to z because opening the connection only exposes z to obstacles that are farther away. Hence, there remains at least one point in Z_{ij} (namely z) for which p is a nearest obstacle. This means that all points of N_{ij} are required. □

Lemma 5.6. *When C_{ij} is opened, N_{ij} contains all possible nearest obstacle points for any point in Z_{ij}.*

Proof. We prove the lemma for Z_i; the proof for Z_j is analogous. For any point $p \in Z_i$, the nearest obstacle points currently lie in N_i or on C_{ij}, by definition. Removing the interior of C_{ij} cannot cause other obstacle points on the same side S_i to suddenly become nearest to p. The only remaining option is that an obstacle on the *other* side S_j becomes nearest to p. Such an obstacle must definitely lie in N_j: by definition, all other obstacle points of S_j were already not closest to C_{ij} itself, so they cannot be closest to a point *beyond* C_{ij}. Therefore, all possible nearest obstacle points for Z_i are included in N_i and N_j. □

5.6 Opening a Connection

To open a closed connection C_{ij}, we now know that we only need to update the medial axis inside the influence zone Z_{ij}. Thus, to compute *MA*$(\mathcal{E}, \mathcal{C}'')$, it is

sufficient to *only* compute $MA(\mathcal{E}, \mathcal{C}'') \cap Z_{ij}$ and then replace $MA(\mathcal{E}, \mathcal{C}') \cap Z_{ij}$ by it. We will refer to the new medial axis part, $MA(\mathcal{E}, \mathcal{C}'') \cap Z_{ij}$, as M_Z for convenience.

Thus, our goal is to compute M_Z. Lemmas 5.5 and 5.6 guarantee that M_Z is defined by the obstacles in the neighbor set N_{ij}. An example of M_Z is shown in Figure 5.3c.

Lemma 5.7. *The medial axis M_Z is a tree that can be projected onto P without overlap.*

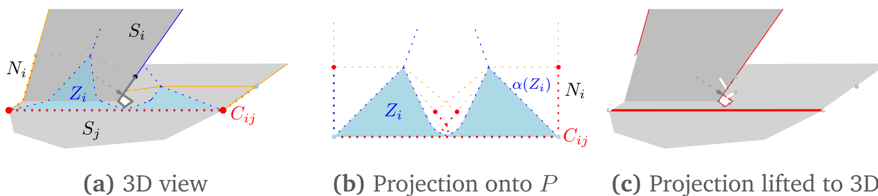
Proof. M_Z can only contain cycles if Z_{ij} contains holes; otherwise, there are no obstacles to circumnavigate. Furthermore, M_Z can only consist of multiple connected components if Z_{ij} is consists of multiple disconnected shapes. Corollary 5.2 states that Z_{ij} is a single shape without holes. Therefore, M_Z is a single tree.

Corollary 5.2 also states that Z_{ij} is non-overlapping when projected onto P. Because M_Z lies entirely inside Z_{ij}, it can be projected onto P without overlap as well. □

5.6.1 Preliminary Algorithm

A preliminary approach to computing M_Z is to project all obstacles of N_{ij} onto the ground plane P and then compute the 2D medial axis of this projection [153]. This is equivalent to a deletion of a site from a 2D Voronoi diagram [25]; the algorithm takes $\mathcal{O}(m \log m)$ time where m is the complexity of N_{ij}. Also, the algorithm is relatively easy to implement by using any available library for Voronoi diagrams in 2D. We therefore still use it in our current implementation of the multi-layered ECM (Section 5.8).

However, due to the multi-layered structure of $\mathcal{E}$, this algorithm does not work in all environments. A single common projection onto P may cause an obstacle of N_{ij} to influence parts of Z_{ij} to which it is actually not closest. Figure 5.4 shows an example in which the obstacles N_i on side S_i cannot be treated as a planar set.



(a) 3D view **(b)** Projection onto P **(c)** Projection lifted to 3D

Figure 5.4: Example in which projecting the entire neighbor set onto the ground plane P leads to problems. **(a)** One side S_i of a connection C_{ij} contains a ramp and a flat surface. (In this view, the flat surface is partly occluded by the ramp. The occluded boundary part is shown in dotted gray.) N_i (shown in bold) contains obstacle points from both parts. **(b)** A projection of the same situation onto P. **(c)** If we project all of N_i onto P at the same time, we effectively treat the points of N_i as obstacles in all surfaces. This will yield an incorrect medial axis for Z_i.

5.6.2 Improved Algorithm: Outline

We now propose an improved algorithm that uses projected distances *without* explicitly projecting all of N_i (or N_j) onto P at the same time. Our new approach starts at the boundary of Z_{ij} and traces the medial axis from there, based on the obstacles that are locally nearest. This approach is more complicated than the first algorithm, and it is more difficult to implement robustly because it relies on different Voronoi diagram algorithms. On the other hand, it avoids the problem of Figure 5.4, and it is provably correct.

The new approach is outlined in Figure 5.5. Figure 5.5a shows the current situation with C_{ij} closed, and the other subfigures represent the algorithm for opening C_{ij}.

Let $M_{Z,i}$ be M_Z under the assumption that there are no obstacles in N_j and that Z_j extends to infinity. Thus, $M_{Z,i}$ is defined solely by the obstacles of N_i. By the same arguments as before, the version of Z_{ij} in which Z_j extends to infinity is a simple shape when projected onto P (Corollary 5.2), and $M_{Z,i}$ is a tree that does not overlap in P either (Lemma 5.7). An example is shown in Figure 5.5b.

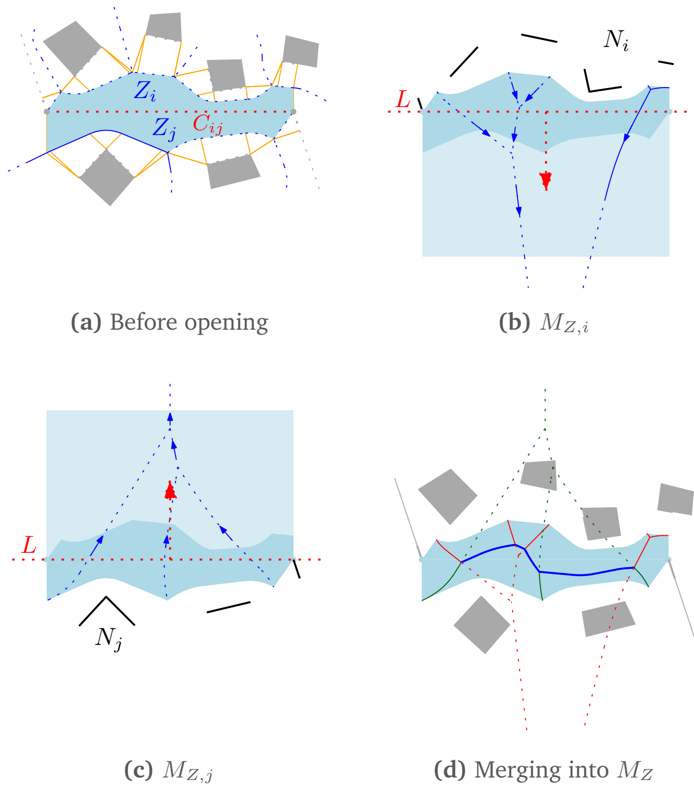
Let $M_{Z,j}$ be defined analogously (Figure 5.5c). We compute $M_{Z,i}$ and $M_{Z,j}$ separately and then merge them to obtain M_Z (Figure 5.5d).

5.6.3 Improved Algorithm: Computing a Single Part

To compute $M_{Z,i}$, we use the *plane sweep algorithm* by Fortune [30], which traces a Voronoi diagram (VD) by moving a horizontal sweep line L downwards. This algorithm is defined for sites in 2D, but we will show how to apply it to our multi-layered problem.

A thorough analysis of Fortune's algorithm has been given by de Berg et al. [7]. We will repeat the most important features. The algorithm maintains an x-monotone 'beach line' consisting of bisector arcs; each arc is defined by the sweep line L and an input site above L. The endpoints of these beach line arcs (which are referred to as 'break points') are the centers of the largest empty disks in the environment that are tangent to L. The VD below the beach line is yet to be determined. As the sweep line moves downwards, the beach line changes, and its break points trace the edges of the VD. There are two types of events: *site* events when L reaches a new site, and *circle* events when L reaches the lowest point of a circle through three sites defining adjacent arcs on the beach line. Each event indicates that a site starts or stops generating a particular arc on the beach line.

We apply Fortune's algorithm to our multi-layered problem by initializing the algorithm in such a way that all *site* events are already handled and all *circle* events can be processed just as in 2D. Assume without loss of generality that C_{ij} is horizontal and that Z_i lies above it. We start with a sweep line at the height of C_{ij} and initialize the beach line as the sequence of arcs $\alpha(Z_i)$. Lemma 5.4 states that this beach line is x-monotone, given that C_{ij} is horizontal. By the same argument, it will remain x-monotone during the sweep. After initialization, we move the



(a) Before opening

(b) $M_{Z,i}$

(c) $M_{Z,j}$

(d) Merging into M_Z

Figure 5.5: We compute the medial axis M_Z in three steps. **(a)** The medial axis when C_{ij} is still closed. **(b)** $M_{Z,i}$ uses only the obstacles of N_i and assumes that Z_j extends to infinity. We compute it using a plane sweep on the ground plane P, starting with the sweep line L at C_{ij}, *without* explicitly projecting all of N_i onto P at the same time. **(c)** Analogously, $M_{Z,j}$ uses only N_j. **(d)** We merge the two parts to obtain M_Z.

sweep line downwards, and the algorithm proceeds exactly as if we were working in 2D.

The essential difference from a 2D problem is that the beach line now represents empty disks in the MLE and not on a single plane. To explain this further, Figure 5.6 shows the initial sweep line situation for the self-overlapping environment of Figure 5.4. An empty disk on the beach line is highlighted in blue. If we would project all of N_i onto P at the same time (as in our old algorithm [153]), then this disk would suddenly contain obstacles from other layers. However, these obstacles are *not nearest obstacles* according to our distance function d_P.

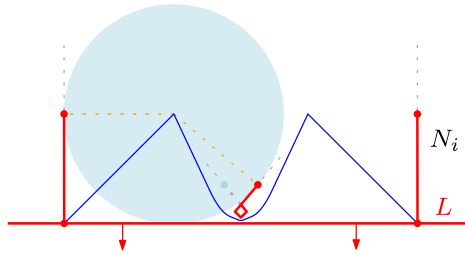


Figure 5.6: The environment from Figure 5.4b when the sweep algorithm begins. A disk on the beach line may contain obstacles from other layers when projected onto P, but these are not *nearest* obstacles.

Each individual event in the sweep algorithm relies only on a point on the beach line and its nearest obstacles. Due to the straight-line and empty-circle properties given in Section 5.3, an event can be projected onto P, and its geometric computations will then work exactly as in 2D: disks are still disks, and paths to nearest obstacles are still straight line segments. The overall algorithm is a combinatorial sequence of events, so it does not require a projection of all events onto P at the same time. Therefore, the algorithm is not affected by the multi-layered structure of N_i. We will now explain further which events occur and how they can be processed.

Site events

When the sweep begins, the endpoints of C_{ij} lie exactly on the sweep line. Both endpoints induce a site event that needs to be processed immediately. The following lemma implies that all other sites lie above C_{ij}, so there are no other site events.

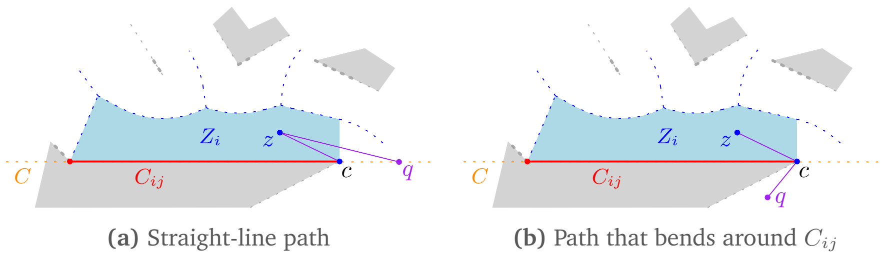
Lemma 5.8. *Each point of N_i either lies above C_{ij} or is an endpoint of C_{ij}.*

Proof. We will prove that any obstacle point on side S_i that does *not* lie above C_{ij} cannot be in N_i. By definition, the interior of C_{ij} is *excluded* from N_i, and the endpoints of C_{ij} are *included*. Thus, we only need to consider the *other* obstacle points of S_i.

Recall that C_{ij} is still a closed obstacle when the set N_i is determined. This means that paths cannot yet go through C_{ij}. Let q be any obstacle point on side S_i that lies below or on the horizontal line C through C_{ij}. Let z be an arbitrary point in Z_i, and let c be the endpoint of C_{ij} that is nearest to z. Note that the shortest path $\pi^*(z, c)$ is unobstructed. We can prove that $\pi^*(z, q)$ is always longer:

- If the line segment $\overline{zq}$ does not intersect C_{ij} when projected onto P, then the shortest path $\pi^*(z, q)$ is at best unobstructed, so it is at least as long as $\overline{zq}$. See Figure 5.7a.
- Otherwise, $\pi^*(z, q)$ must navigate around C_{ij}, so it is at best a sequence of two line segments that bends around c (or the other endpoint of C_{ij}). See Figure 5.7b.

In both cases, $\pi^*(z, q)$ is clearly longer than $\pi^*(z, c)$. Therefore, q cannot be nearest to any point in Z_i, and q cannot occur in N_i. □



(a) Straight-line path **(b)** Path that bends around C_{ij}

Figure 5.7: One side S_i of a horizontal connection C_{ij}. An arbitrary point in $z \in Z_i$ is highlighted. Any obstacle point q below or on the supporting line C of C_{ij} cannot belong to the neighboring obstacles N_i. **(a)** The case in which $\pi^*(z, q)$ does not navigate around C_{ij}. **(b)** The case in which $\pi^*(z, q)$ navigates around C_{ij}. In both cases, an endpoint c of C_{ij} will be closer to z than q is. Therefore, q cannot be in N_i.

Circle events

Initially, the potential circle events can be obtained by inspecting all 3-tuples of adjacent arcs in $\alpha(Z_i)$, just as in 2D. Because we look at adjacent arcs only, we will only trace Voronoi edges of obstacles that are actually Voronoi neighbors in the MLE. Since $M_{Z,i}$ will be a tree and Z_{ij} is planar when projected onto P, only adjacent Voronoi edges traced on the beach line can meet in a Voronoi vertex. Therefore, all circle events are discovered.

The effect of a circle event is the same as in 2D. Two of the empty disks on the beach line merge into one, a site locally disappears as a nearest site, and an arc disappears from the beach line. In terms of the VD, two Voronoi edges merge into a Voronoi vertex, and a new Voronoi edge starts being traced. The disappearance of an arc from the beach line induces two new 3-tuples of adjacent arcs. The circles

tangent to the corresponding 3-tuples of sites may induce new circle events below the sweep line. These new events will be added to the event queue (sorted by y coordinate).

Each individual event (i.e. each change of a nearest site) can be processed exactly as in 2D. Therefore, all events are recognized, and the algorithm correctly computes $M_{Z,i}$. As shown in Figure 5.5b, we end up tracing a tree of medial axis arcs, starting at the leaves and moving towards the root as we sweep downwards. By Lemma 5.8, the endpoints of C_{ij} are the lowest sites, so the final event occurs when the endpoints of C_{ij} become the only remaining nearest obstacles to the sweep line. These endpoints generate an infinite final edge that is perpendicular to C_{ij}.

5.6.4 Improved Algorithm: Merging the Two Parts

We compute $M_{Z,j}$ similarly to $M_{Z,i}$, but by starting with $\alpha(Z_j)$ as the beach line and moving the sweep line upwards instead of downwards. Next, we merge $M_{Z,i}$ and $M_{Z,j}$ to obtain M_Z. We do this by using the merge procedure for Voronoi diagrams from Shamos and Hoey [131]. Figure 5.5d shows an example.

The merge procedure traverses the Voronoi cells of $M_{Z,i}$ and $M_{Z,j}$ simultaneously and builds a new monotone sequence of Voronoi edges between them. Afterwards, it removes the parts of $M_{Z,i}$ and $M_{Z,j}$ that are no longer needed. This algorithm requires $M_{Z,i}$ and $M_{Z,j}$ to be planar; we have shown in Section 5.6.3 that this requirement is fulfilled. The second requirement is that N_i and N_j lie in separate half-planes. Lemma 5.8 implies that this holds: N_i and N_j are separated by C. The only exceptions are the endpoints of C_{ij} at which the merge starts and ends.

In each step of the merge procedure, there is one nearest site (a point or a line segment) $n_i \in N_i$ and one nearest site $n_j \in N_j$, and the new Voronoi edge is the bisector of n_i and n_j. This bisector arc ends when either of the nearest sites changes. Because the medial axes $M_{Z,i}$ and $M_{Z,j}$ are correct, they represent for all points in Z_{ij} the nearest sites from N_i and N_j, respectively. They therefore store all the information required to detect the changes in nearest sites. Furthermore, the computations in each step work exactly as in 2D due to the straight-line and empty-circle properties. Thus, the merge procedure [131] is not affected by the potential multi-layered nature of N_{ij}, and it correctly computes M_Z.

5.6.5 Improved Algorithm: Summary

We now summarize our algorithm for opening a connection. Let $\mathcal{E}$ be a multi-layered environment, and let $MA(\mathcal{E}, \mathcal{C}')$ be the medial axis computed so far, in which a non-empty set of connections $\mathcal{C}' \subseteq \mathcal{C}$ is closed. We open a connection $C_{ij} \in \mathcal{C}'$ to obtain $MA(\mathcal{E}, \mathcal{C}'')$, where $\mathcal{C}'' = C' \setminus \{C_{ij}\}$. Opening C_{ij} works as follows:

1. Remove the arcs from $MA(\mathcal{E},\mathcal{C}')$ that are nearest to the interior of C_{ij}. These are the arcs of $MA(\mathcal{E},\mathcal{C}')$ that bound the influence zone Z_{ij} described in Section 5.5.

2. Compute $M_Z = MA(\mathcal{E},\mathcal{C}'') \cap Z_{ij}$ as described in Sections 5.6.3 and 5.6.4.

3. Insert M_Z into $MA(\mathcal{E},\mathcal{C}')$ to obtain $MA(\mathcal{E},\mathcal{C}'')$.

5.7 Analysis

In this section, we prove the correctness and the worst-case running time of our algorithm, and we give the storage complexity of the multi-layered ECM.

5.7.1 Correctness of the Algorithm

The overall construction algorithm outlined in Section 5.4 first computes the medial axis with all connections as closed obstacles. It then iteratively opens a closed connection, using the algorithm from Section 5.6, until all connections are open. The following theorem states that this algorithm correctly computes the multi-layered medial axis:

Theorem 5.1. *Let $\mathcal{E}$ be an MLE. Computing $MA(\mathcal{E},\mathcal{C})$ and then iteratively opening each connection in $\mathcal{C}$ as described in Section 5.6 yields the medial axis $MA(\mathcal{E})$.*

Proof. Each iteration of this algorithm starts with a correct medial axis $MA(\mathcal{E},\mathcal{C}')$ and computes a correct medial axis $MA(\mathcal{E},\mathcal{C}'')$ in which one more connection has been removed as an obstacle. By induction over the number of iterations, the final result is the correct medial axis $MA(\mathcal{E})$ in which all connections are traversable. □

The connections can be opened in any order without affecting the correctness of the algorithm. However, we will see in the next subsection that opening the connections in a particular order can affect the *running time* of the algorithm.

5.7.2 Running Time of the First Step

The first step of our construction algorithm computes the medial axis of all layers with all connections closed. This can be achieved using a single 2D algorithm because the medial axis consists of separate 2D components that do not yet influence each other. Lemma 5.1 has shown that the number of connections k is linear in the number of obstacle points n in the MLE. Thus, the presence of k connections does not affect the asymptotic complexity, and we essentially compute a 2D medial axis of an input with complexity $\mathcal{O}(n)$. Section 4.2.4 has shown that this can be performed in $\mathcal{O}(n \log n)$ time.

5.7.3 Running Time to Open One Connection

Lemma 5.9. *A single connection C_{ij} can be opened in $\mathcal{O}(m \log m)$ time, where m is the complexity of the neighbor set N_{ij}.*

Proof. Our algorithm for opening a connection starts with two instances of a sweep line algorithm [30]. Both instances take $\mathcal{O}(m \log m)$ time because they involve $\mathcal{O}(m)$ circle events that need to be maintained in sorted order. Next, we perform one $\mathcal{O}(m)$-time merge step of a divide-and-conquer algorithm [131]. Therefore, the total running time is $\mathcal{O}(m \log m)$. □

In many practical scenarios, the connections and obstacles are spread throughout the environment, and m will be constant in most iterations of the algorithm. However, if many obstacles are close to the connection, m can be $\Theta(n)$.

5.7.4 Total Running Time Using a Bad Order

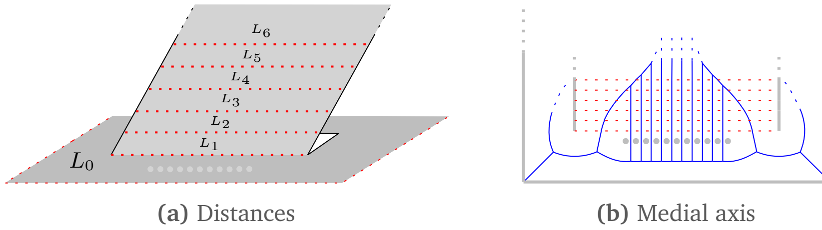
Based on Lemma 5.9, iteratively opening *all* connections takes $\mathcal{O}(\sum_{i=0}^{k-1} m_i \log m_i)$ time in total, where m_i is the neighbor set complexity of the connection that is opened in iteration i. Note that the neighbor sets of closed connections can change during the algorithm because obstacles may turn out to influence other connections when a nearby connection is opened. In other words, a neighbor set's complexity depends on what lies beyond the nearby connections that are already open. This suggests that the total construction time depends on the *order* in which the connections are opened.

In many environments, each obstacle will only affect the algorithm in a constant number of iterations regardless of this order, which means that the total construction time will remain $\mathcal{O}(n \log n)$. However, there are environments in which opening the connections in an 'unlucky' order leads to a worse construction time. The following lemma analyzes this:

Lemma 5.10. *There exists an environment with n obstacle vertices and k connections such that opening the connections in an inefficient order gives $\Theta(k)$ neighbor sets of complexity $\Theta(n)$.*

Proof. Figure 5.8 shows an example in which a ramp has been subdivided into a chain of small layers. All connections are close together, and the bottom connection has a row of $\Theta(n)$ neighboring obstacles on the ground plane. If the connections are opened from the bottom to the top, the first neighbor set has complexity $\Theta(n)$. When the first connection is open, the same obstacles have become neighbors of the second connection, so the second neighbor set will also have complexity $\Theta(n)$. By repeating this argument, we see that this holds for each of the k iterations. □

When using the $\mathcal{O}(m \log m)$-time algorithm from Section 5.6 in each iteration, the total running time for this unfortunate order becomes $\mathcal{O}(kn \log n)$. In our first publication on the multi-layered ECM [153], we reported this as the worst-case running time of our algorithm.



(a) Distances　　**(b)** Medial axis

Figure 5.8: A worst-case example for our incremental algorithm. **(a)** A ramp has been subdivided into a sequence of small layers. The ground floor contains a row of $\Theta(n)$ obstacles. If we open the connections from bottom to top, each connection will have $\Theta(n)$ obstacles in its neighbor set. **(b)** In the final medial axis, many edges will run through multiple connections.

5.7.5 Total Running Time Using a Good Order

To improve upon this result, we now show that we can *always* construct the medial axis in $\mathcal{O}(n \log n \log k)$ time by choosing an 'easy' connection in each iteration. We begin by showing that the medial axis has linear size throughout the entire algorithm.

Lemma 5.11. *For any MLE, the medial axis has size $\mathcal{O}(n)$ in each iteration of our algorithm.*

Proof. In the initial step of the algorithm, we compute a medial axis of $\mathcal{O}(n)$ sites, which has size $\mathcal{O}(n)$. Since opening a connection is analogous to deleting a Voronoi site, the asymptotic complexity of the graph cannot increase during the algorithm. □

Lemma 5.12. *When q connections are still closed, there is at least one connection with a neighbor set complexity of $\mathcal{O}(\frac{n}{q})$.*

Proof. By Lemma 5.11, the medial axis always has $\mathcal{O}(n)$ arcs. At any point in the incremental algorithm, every arc bounds the influence zone of at most two connections, namely one on each side of the arc. Therefore, the combined complexity of all influence zones (or, equivalently, of all neighbor sets) is $\mathcal{O}(n)$. When this $\mathcal{O}(n)$ complexity is shared by q connections, there must be at least one connection with a neighbor set complexity of $\mathcal{O}(\frac{n}{q})$. □

Lemma 5.13. *The k connections can be opened in $\mathcal{O}(n \log n \log k)$ total time.*

Proof. We will repeatedly open the connection with the smallest neighbor set complexity. To achieve this, we first compute the neighbor set complexities of all k closed connections. This can be done in $\mathcal{O}(n)$ time by traversing the medial axis once and incrementing the complexity for a connection C_x whenever an arc has C_x as a nearest obstacle. Next, we sort the connections by complexity in a

balanced binary search tree $\mathcal{T}$. This requires $\mathcal{O}(k \log k)$ time, which is $\mathcal{O}(n \log n)$ because k is $\mathcal{O}(n)$.

Assume for now that we can maintain the sorting order in $\mathcal{T}$ such that we can always get the connection with the smallest complexity. Let C_q be this easiest connection when q connections are closed. By Lemma 5.12, it has a complexity of $\mathcal{O}(\frac{n}{q})$. Using the algorithm of Section 5.6, we can open it in $\mathcal{O}(\frac{n}{q} \log \frac{n}{q})$ time.

Next, we show that $\mathcal{T}$ can indeed be maintained efficiently. The neighbor sets of other closed connections can change due to opening C_q. However, any connection that is affected *must* be one of the neighboring obstacles of C_q. Therefore, the number of neighbor sets that can change is $\mathcal{O}(\frac{n}{q})$. We can update the complexities of these neighbor sets in $\mathcal{O}(\frac{n}{q})$ time: for each added or removed arc with a connection C_x as a nearest obstacle, we increment or decrement the neighbor set complexity for C_x. Afterwards, we update the search tree $\mathcal{T}$ by deleting and re-inserting all complexities that have changed. This takes $\mathcal{O}(\frac{n}{q} \log q)$ time because it requires $\mathcal{O}(\frac{n}{q})$ update operations. Thus, opening C_q and updating $\mathcal{T}$ afterwards takes $\mathcal{O}(\frac{n}{q}(\log \frac{n}{q} + \log q)) = \mathcal{O}(\frac{n}{q} \log n)$ time.

Because $\mathcal{T}$ is maintained correctly, we can always open the connection with the lowest neighbor set complexity. For all k iterations combined, we obtain a running time of $\mathcal{O}(\sum_{q=1}^{k} \frac{n}{q} \log n) = \mathcal{O}(\sum_{q=1}^{k}(n \log n \cdot \frac{1}{q})) = \mathcal{O}(n \log n \sum_{q=1}^{k} \frac{1}{q}) = \mathcal{O}(n \log n \cdot H_k)$. Here, H_k is the kth harmonic number, which is known to be $\Theta(\log k)$. Therefore, the total running time to open all connections is $\mathcal{O}(n \log n \log k)$. □

Combined with the $\mathcal{O}(n \log n)$ running time for the first step (i.e. computing the medial axis of each layer with all connections closed), we obtain the following result:

Theorem 5.2. *The medial axis of a multi-layered environment with n obstacle vertices and k connections can be computed in $\mathcal{O}(n \log n \log k)$ time.*

5.7.6 Running Time In Case of Linear-Time Iterations

Recently, Khramtcova and Papadopoulou [81] have proposed an $\mathcal{O}(m)$-time algorithm for deleting a line segment site s from a 2D Voronoi diagram, where m is the complexity of the Voronoi cell of s. It is therefore likely that a connection with a neighbor set complexity of m can be opened in $\mathcal{O}(m)$ time as well. However, it is unclear whether this algorithm can be applied to the multi-layered neighborhood of a connection. Proving this is a challenging topic for future work.

The next theorem states that an $\mathcal{O}(m)$-time algorithm for opening a connection would improve the overall running time to $\mathcal{O}(n \log n)$. This would be an important result: the number of connections would have no influence on the asymptotic time, and the algorithm would be *optimal* because a lower bound of $\Omega(n \log n)$ already exists for 2D environments [5].

Theorem 5.3. *If a connection with neighbor set complexity m can be opened in $\mathcal{O}(m)$ time, then the medial axis of an MLE can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. To achieve this result, we can no longer afford to keep the connections in sorted order by complexity. Instead, we will iteratively open a connection that is *sufficiently easy*, but not necessarily the easiest. This allows us to open the connections in $\log k$ phases that each take $\mathcal{O}(n)$ time.

By Lemma 5.12, there must be connections in the first $k/2$ iterations with a complexity of $\mathcal{O}(\frac{n}{k})$, $\mathcal{O}(\frac{n}{k-1})$, ..., $\mathcal{O}(\frac{n}{k/2})$. These complexities are all $\leq d \cdot \frac{n}{k}$ for some constant $d > 0$. In practice, d can be quite small because the complexity of the initial medial axis can only increase by a small constant factor.

As in Lemma 5.13, we first compute all neighbor set complexities in $\mathcal{O}(n)$ time. However, instead of storing the connections in a tree, we fill a doubly-linked list $\mathcal{D}$ with all connections that have a neighbor set complexity of at most $N = d \cdot \frac{n}{k}$. We can then choose an arbitrary connection C in $\mathcal{D}$ and open it in $\mathcal{O}(\frac{n}{k})$ time. For all $\mathcal{O}(\frac{n}{k})$ complexities that change, we add the corresponding connection to $\mathcal{D}$ if its complexity becomes sufficiently small, or remove it from $\mathcal{D}$ if the complexity becomes too large. These operations take constant time if we give each connection a pointer to its position in $\mathcal{D}$.

By doing this $k/2$ times, we spend $\mathcal{O}(n)$ time on opening half of all connections. In the next $k/4$ iterations, there must be a connection with a neighbor set complexity of $\leq 2N$. Thus, if we update $\mathcal{D}$ using a new upper bound of $2N$, we can perform $k/4$ more iterations in $\mathcal{O}(n)$ total time. If we repeat this process of doubling N and opening half of the remaining connections, we will eventually have opened all connections in $\log k$ phases of $\mathcal{O}(n)$ time each. This yields a total running time of $\mathcal{O}(n \log k)$. Combined with the $\mathcal{O}(n \log n)$-time first step and the fact that k is $\mathcal{O}(n)$, the theorem follows. □

5.7.7 Complexity of the Medial Axis

Finally, we give the storage complexity of the multi-layered medial axis when all connections have been opened. It follows immediately from Lemma 5.11: the complexity is $\mathcal{O}(n)$ at each point in the algorithm, including at the end.

Theorem 5.4. *The medial axis of a multi-layered environment with n obstacle vertices and k connections has a storage complexity of $\mathcal{O}(n)$.*

5.8 Implementation

We have extended our C++ implementation of the 2D ECM (Section 4.4) to multi-layered environments. For each 2D component of the algorithm, we have included the same Vroni and Boost implementations as in Chapter 4.

In preliminary experiments, the GPU-based implementation turned out to be too imprecise to yield correct results. Opening a connection C_{ij} requires a certain level of precision such that the new medial axis M_Z through C_{ij} can be reliably merged into the main medial axis. In general, the rendering resolution that is

needed to open all connections correctly cannot be predicted. We therefore exclude the GPU-based implementation from our experiments in this chapter.

In some environments, the medial axis may contain edges that run across many layers. We ensure that each edge can be associated with a single layer, mainly for visualization purposes. We do this by splitting each edge wherever it intersects one of the (now opened) connections. Computing these intersections takes extra time, and it can increase the worst-case complexity of the graph to $\mathcal{O}(kn)$ if many edges intersect many connections, such as in Figure 5.8b. We include this post-processing step in the measurements of Section 5.9.

Even after the ECM edges have been split up, a single cell in the multi-layered ECM can span multiple layers because points on the medial axis can have their nearest obstacles in other layers. In other words, a point in a particular layer L_i may belong to an ECM cell whose ECM edge does *not* lie in L_i. This makes point-location queries slightly more involved in practice because the appropriate layer needs to be found. A basic solution is to create a separate data structure that maps these difficult areas of $\mathcal{E}_{free}$ to the layer IDs of the corresponding edges. We will not discuss these details further in this thesis.

We have used many test environments to obtain a robust implementation. For future work, there are two ways in which our implementation can still be improved. First, we currently open the connections in the order in which they are listed in the environment, which is not necessarily optimal. Second, we open the connections using the algorithm from Section 5.6.1, so we cannot yet handle self-overlap near connections such as in Figure 5.4. On the other hand, this allowed us to use existing robust Voronoi diagram libraries such as Vroni [48] and Boost [14] for opening connections. These theoretical issues are not a problem for any of the real-world environments in our test set. Section 5.9 will show that our implementation of the multi-layered ECM construction algorithm is very fast in practice.

5.9 Experiments and Results

This section assesses the performance of our ECM implementations in multi-layered environments. We use only one CPU core in our experiments, except at the end of Section 5.9.2 where we use multi-threading to speed up the algorithm.

5.9.1 Environments

The multi-layered environments of our experiments are described below. More details can be found in Table 5.1.

- *Ramps* (Figures 5.9a to 5.9c) consists of three flat layers connected by four ramps. Each ramp is modelled as a separate layer for simplicity.
- *Ramps2* (Figure 5.9) is a version of *Ramps* in which we have added 56 polygonal obstacles to the flat layers.

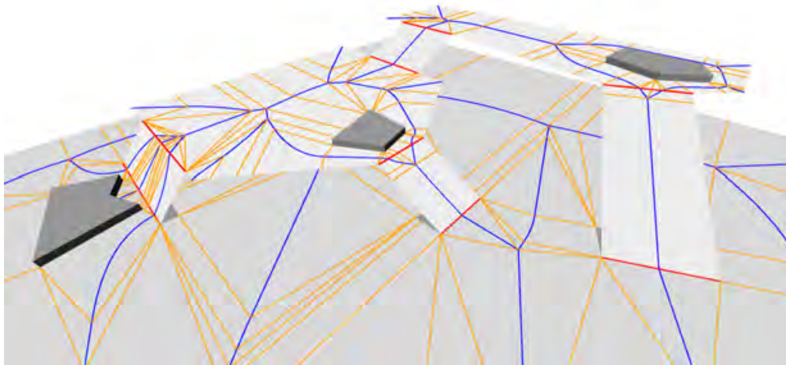
- *Library* (Figure 5.11) is a simplified model of the Utrecht University campus library.
- *Station* (Figures 5.10a and 5.10b) is a model of a train station with one main hall and one layer containing all platforms; these two layers are connected by 32 ramps. Again, each ramp has been modelled as a separate layer.
- *Tower* (Figure 5.12) is a complex multi-story apartment building.
- *Stadium* (Figures 5.10b and 5.10c) is a model of an American football stadium with many staircases and obstacles. Since it has been drawn manually based on real-world data, it contains small gaps that generate disconnected graph components. It also features sequences of nearly-collinear points that generate medial axis edges when the input coordinates have been rounded and scaled. These graph elements seem redundant, but they are correct in our scaled integer coordinate system.
- *BigCity* (Figure 5.13) is a combination of the 2D city environment, six instances of *Tower*, and two instances of *Library*. The towers are highly detailed compared to the rest of the environment. Voxel-based navigation mesh algorithms would require a very high resolution to capture all details.
- *BigCity2x2* consists of four tiled instances of *BigCity*. It measures 1 km^2 and contains 784 connections.

Environment	Geometry		Multi-layered	
	#Obstacle vertices	Size (m)	#Layers	#Connections
Ramps	147	100×100	7	8
Ramps2	422	100×100	7	8
Library	717	60×24	9	8
Station	2242	153×111	34	64
Tower	6058	35×35	17	30
Stadium	12915	280×184	18	82
BigCity	49476	500×500	113	196
BigCity2x2	197884	1000×1000	449	784

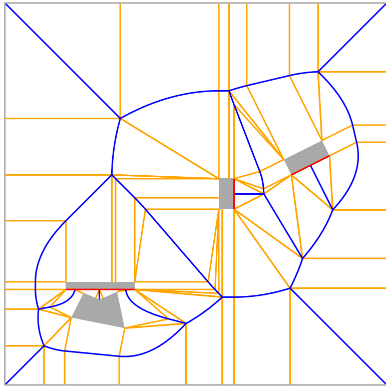
Table 5.1: Details of the test environments. The *Geometry* columns show the number of obstacle vertices and the physical width and height of the environment (in meters). The *Multi-layered* columns show the number of layers and connections of each environment.

5.9.2 Computing the ECM

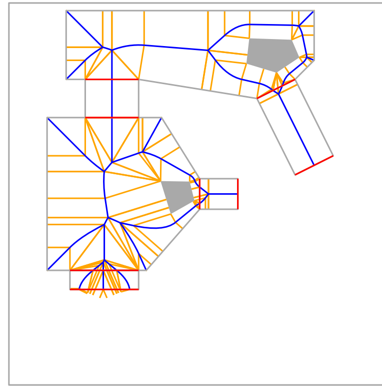
Table 5.2 shows the ECM complexities and construction times for these environments. For comparative purposes, we have also added the results for *Zelda8x8*, the largest 2D environment from Chapter 4. The construction time is still well under a



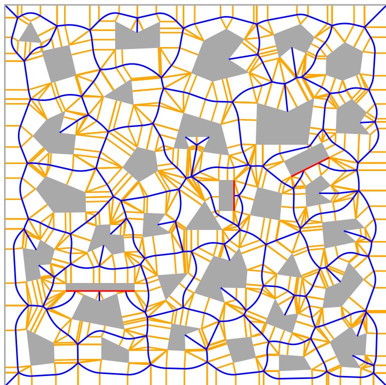
(a) *Ramps* in 3D



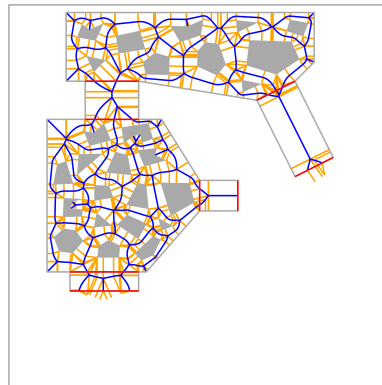
(b) *Ramps*



(c) *Ramps* (continued)

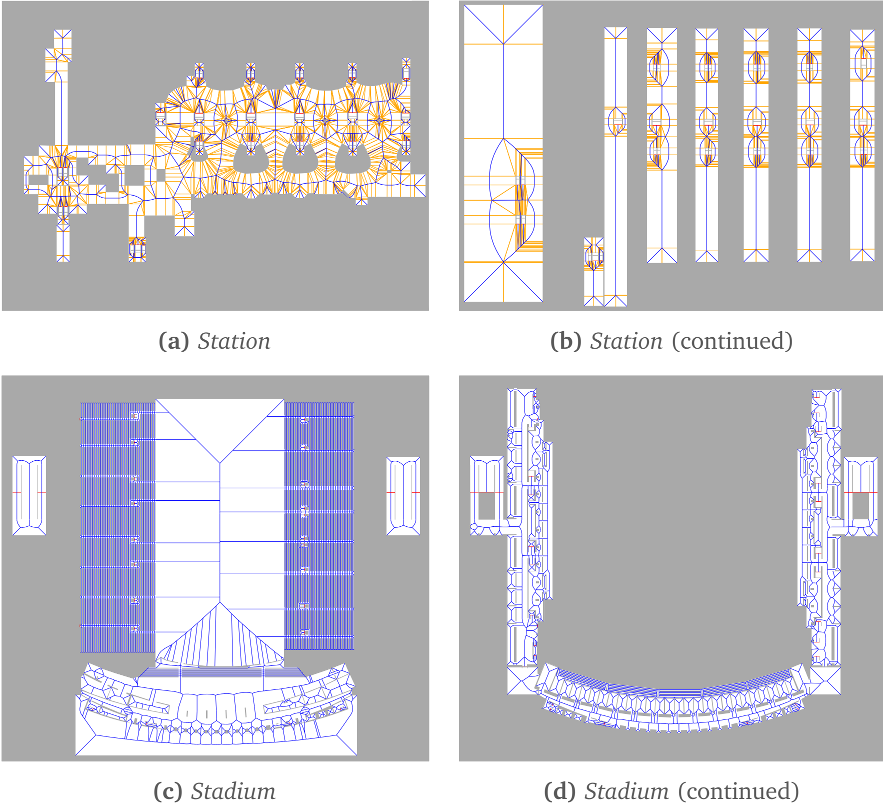


(d) *Ramps2*



(e) *Ramps2* (continued)

Figure 5.9: The *Ramps* and *Ramps2* environments and their ECMs. **(a)** 3D view of *Ramps*. **(b)–(c)** Top views of *Ramps* for the ground floor and the other layers, respectively. **(d)–(e)** Top views of *Ramps2*, a version of Ramps in which many extra obstacles have been added.



(a) *Station*

(b) *Station* (continued)

(c) *Stadium*

(d) *Stadium* (continued)

Figure 5.10: Top views of some of the layers of *Station* and *Stadium*. For *Stadium*, the nearest-obstacle annotations of the ECM have been omitted for clarity.

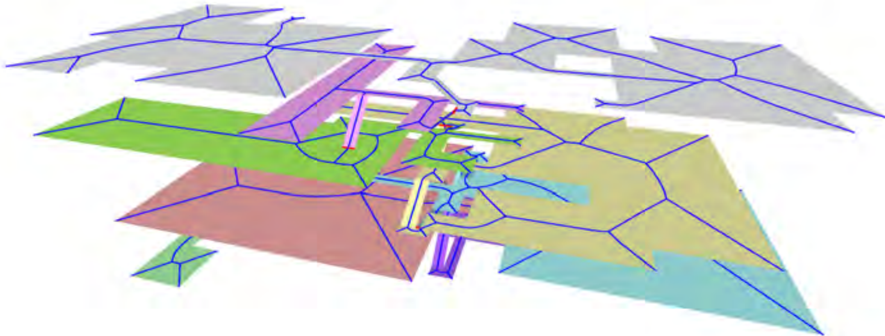


Figure 5.11: 3D view of the *Library* environment and its medial axis. The nearest-obstacle annotations of the ECM have been omitted for clarity.

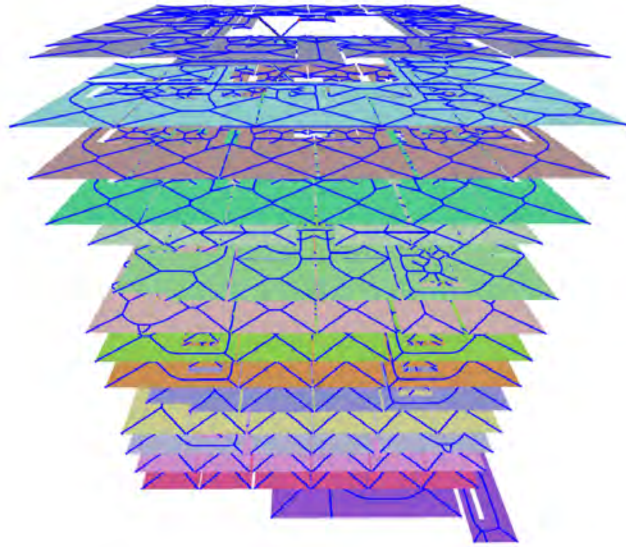


Figure 5.12: 3D view of the *Tower* environment and its medial axis.

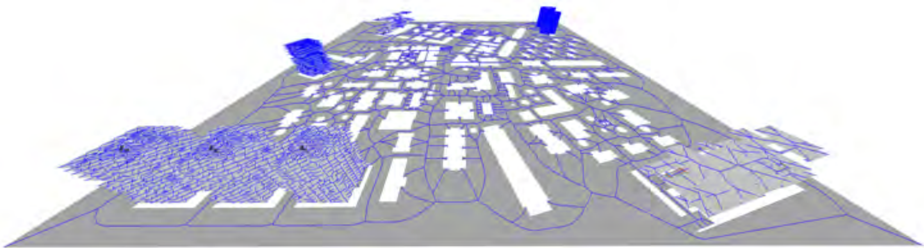


Figure 5.13: 3D view of the *BigCity* environment and its medial axis. For clarity, the nearest-obstacle annotations of the ECM have been omitted. The gray obstacles from the 2D *City* environment in Chapter 4 are now modelled as holes in the walkable space of *BigCity*. *BigCity2x2* is not shown because it is very large and structurally similar to *BigCity*.

second for all MLEs except the two *BigCity* variants. For the largest environment, *BigCity2x2*, the construction takes about 9 seconds when using Vroni.

The Boost implementation for Voronoi diagrams is thread-safe, so we can use *multi-threading* to compute the initial ECMs of all layers in parallel. The running times for this multi-threaded Boost version are also shown in Table 5.2. We used OpenMP with 5 parallel threads and dynamic scheduling. This version performed particularly well in environments with many complex layers; in particular, the ECM of *BigCity2x2* was computed in approximately 4.2 seconds. Standard deviations among running times were higher because the threads were scheduled in an unpredictable way. Still, this implementation shows that multi-threading is a promising addition.

Environment	ECM complexity			ECM time (ms)		
	#Vertices	#Edges	#BPs	Vroni	Boost	Boost (MT)
Ramps	54	61	181	5.8 [0.2]	9.4 [0.3]	7.6 [0.1]
Ramps2	228	290	1118	16.3 [0.4]	29.5 [0.6]	21.2 [0.1]
Library	219	222	599	14.7 [0.3]	20.2 [0.4]	9.2 [0.1]
Station	660	768	2804	68.6 [0.3]	97.3 [0.5]	74.3 [0.3]
Tower	4948	4979	14407	248.8 [2.2]	383.4 [1.3]	110.1 [3.4]
Stadium	6303	7754	26323	442.4 [8.2]	572.2 [1.6]	263.5 [4.7]
Zelda8x8 (2D)	18848	22480	81365	997.4 [5.4]	1529.1 [3.8]	-
BigCity	32264	32652	104002	2168.6 [19.6]	3430.0 [10.9]	925.7 [29.6]
BigCity2x2	129147	130702	416411	8972.5 [32.7]	14287.0 [41.7]	4219.3 [91.9]

Table 5.2: Details of the ECMs for our experiments. The *ECM complexity* columns show the number of vertices, edges, and bending points (BPs) in the ECM computed using Boost. The *ECM time* columns show the ECM construction time for each implementation: Vroni, Boost, and Boost using 5 parallel threads. All times are in milliseconds and have been averaged over 10 runs. Standard deviations are shown between square brackets.

Environment	Path only	Path + IR	Visibility
Ramps	0.005 [0.004]	0.04 [0.02]	0.03 [0.01]
Ramps2	0.02 [0.01]	0.11 [0.05]	0.06 [0.02]
Library	0.02 [0.01]	0.15 [0.08]	0.03 [0.01]
Station	0.05 [0.04]	0.21 [0.13]	0.09 [0.08]
Tower	0.40 [0.30]	0.72 [0.40]	0.06 [0.03]
Stadium	0.45 [0.47]	0.80 [0.64]	0.14 [0.12]
Zelda8x8 (2D)	1.18 [1.13]	1.93 [1.50]	0.05 [0.02]
BigCity	0.87 [1.17]	1.32 [1.46]	0.11 [0.07]
BigCity2x2	3.71 [4.41]	4.56 [5.01]	0.18 [0.12]

Table 5.3: Results of the other experiments. All times are in milliseconds, averaged over 10,000 random queries. Standard deviations are shown between square brackets.

5.9.3 Other Operations

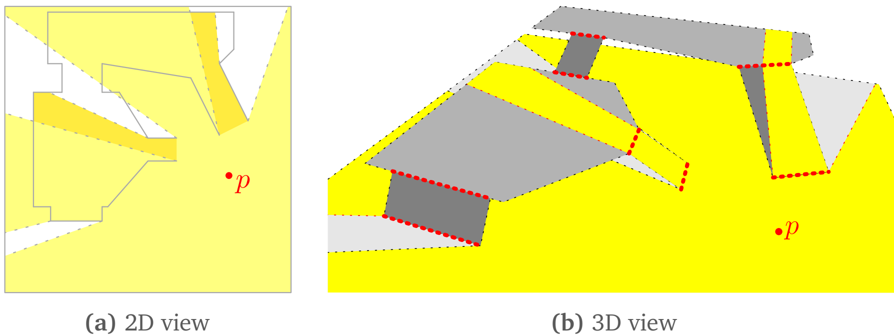
In Chapter 4, we have described various geometric operations and applications of the ECM in 2D: computing paths on the medial axis, computing indicative routes

with clearance, and computing visibility information. For completeness, we have applied the corresponding experiments from Section 4.5 to our new MLEs as well.

When computing a random query point, we first chose a random layer such that the probability of a layer being chosen was proportional to its surface area. We then computed a random point within this layer. All other settings of the experiments have remained the same as in the previous chapter.

The results for path planning are comparable to the results in Chapter 4: path planning is slower in more complex environments, and standard deviations are high because the queries differ in difficulty. In *BigCity2x2*, the average running time for computing an indicative route was highest (4.56 ms) because this environment is more complex than any of the 2D environments from Chapter 4.

The visibility polygon algorithm from Section 4.3.5 can also be applied to MLEs. We do not need to make any changes to the 2D algorithm because it is only based on the adjacency between ECM cells. In an MLE, the algorithm computes a '2.5D' visibility polygon [129] that can span multiple layers while ignoring height differences along surfaces. This is coarse approximation of full 3D visibility, but it is still useful for crowd simulation purposes. An example is illustrated in Figure 5.14.



(a) 2D view **(b)** 3D view

Figure 5.14: Example of a visibility polygon $V(p)$ for a query point p, using a simplified version the *Ramps* environment. **(a)** When projected onto P, $V(p)$ is a non-overlapping polygon, just like in 2D environments. The part of $V(p)$ that lies on the ground plane is shown in a different color than the parts that lie on other layers. **(b)** The same visibility polygon lifted back into 3D.

Visibility queries in MLEs perform comparably to those in 2D environments. These queries are local and less dependent on the overall complexity of an environment. In *BigCity2x2*, the average running time is highest because most query points were on the ground plane (which has the highest surface area), and this ground plane is essentially a more complex version of the *City* environment.

These results indicate that the multi-layered ECM supports the same efficient operations and applications as the 2D ECM.

5.10 Conclusions and Future Work

Modern games and simulations feature environments that cannot be represented in 2D. Examples include multi-story buildings, train stations, or cities with bridges and tunnels. In this chapter, we have extended the Explicit Corridor Map navigation mesh to a new domain of *multi-layered environments*. A walkable environment (WE) is a collection of walkable surfaces in 3D with a consistent direction of gravity. A multi-layered environment (MLE) is a WE that has been subdivided into 2D layers connected by k line segment connections. These formalized concepts are useful for other navigation meshes as well.

We have defined the medial axis and the ECM for WEs and MLEs based on projected distances on the ground plane. We have presented an algorithm that computes the multi-layered medial axis in $\mathcal{O}(n \log n \log k)$ time by initially treating all connections as closed obstacles and then opening them incrementally. Opening a connection is comparable to a deletion of a Voronoi site, but with possible complications such that 2D deletion algorithms cannot always be applied.

Experiments show that our implementation can compute the medial axis and ECM of an MLE efficiently, and that the operations and applications for the ECM described in Chapter 4 apply to MLEs as well. As such, the ECM enables real-time path planning and crowd simulation for disk-shaped characters in 2D and multi-layered environments.

Discussion and future work. Just as in 2D, the complicated nature of the medial axis may be seen as a disadvantage of the ECM compared to e.g. grids or triangulations. For MLEs, the ECM construction algorithm is relatively complex, and the possibility of ECM cells spanning multiple layers may seem counter-intuitive. However, because the multi-layered ECM is based on a continuous medial axis for the entire MLE, it shares the same advantages as the ECM for 2D environments. Therefore, the ECM is a useful basis for crowd simulation in MLEs as well.

We have shown how the multi-layered medial axis construction time can be improved to an optimal $\mathcal{O}(n \log n)$ if a connection bounded by m medial axis arcs can be opened in $\mathcal{O}(m)$ time. Recent results for 2D Voronoi diagrams indicate that such a linear-time algorithm might exist [81], but these results are not immediately applicable to our problem. Thus, finding an $\mathcal{O}(m)$-time algorithm for opening a connection is a challenging topic for future work.

In the future, we intend to create *exact* algorithms for obtaining walkable and multi-layered environments from arbitrary 3D geometry. Up until now, most existing algorithms are based on voxels; these techniques approximate the geometry, do not scale well to large environments, and tend to require a lot of parameter tuning. Exact algorithms would depend only on the actual complexity of the environment (e.g. the number of triangles). On the other hand, exact algorithms also recognize small details such as gaps or overlaps that were accidentally introduced by the environment's designer. It may be difficult to obtain a robust implementation in which these details can still be filtered away.

Finally, it is also interesting to lift other 2D data structures (such as visibility graphs) to the multi-layered domain. We believe that multi-layered environments give rise to an interesting new class of problems for future research.

6 The Explicit Corridor Map in Dynamic Environments

In this chapter, we present algorithms that update the ECM locally when a *dynamic obstacle* is inserted or removed. These operations on the ECM enable efficient crowd simulations in dynamically changing environments.

This chapter is based on the following publication:

- W.G. van Toll, A.F. Cook IV, and R. Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012. [154]

6.1 Introduction

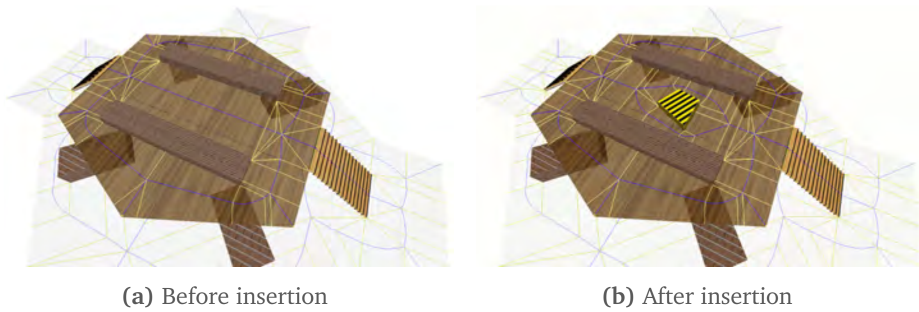
Up until now, this thesis has presented navigation meshes as efficient representations of a 2D or 3D environment for path planning purposes.

In this chapter, we extend the concept to *dynamic environments* in which obstacles can appear, disappear, or move during the simulation. Such an obstacle may have a large impact on the environment. For example, imagine a bridge collapsing, an explosion opening up a new route, or a large vehicle blocking an alley. In such cases, a local collision avoidance method may not be able to guide characters towards their goals: a character may get stuck along its old route, instead of looking for a detour.

The solution to such problems is to *update the navigation mesh* and to let characters *re-plan* their paths in the updated mesh. Recomputing the navigation mesh from scratch is too computationally expensive for real-time performance, especially if the environment is complex. Therefore, we are interested in algorithms that update the mesh *locally*, i.e. only in the areas that actually change.

This chapter shows how to locally update the Explicit Corridor Map (ECM) from Chapters 4 and 5 in response to dynamic obstacles. An example of a dynamic update is shown in Figure 6.1. We focus on *insertions* and *deletions* of obstacles. In practice, it is common to treat *moving* obstacles as locally avoidable entities (just like moving characters) until they become stationary, or to approximate their movement by a sequence of deletions and insertions.

Our algorithms are based on site insertions and deletions in Voronoi diagrams; the algorithms are efficient because they update the ECM only in the areas that change. We acknowledge that the ideas behind these algorithms are not novel,



(a) Before insertion **(b)** After insertion

Figure 6.1: 3D impression of a dynamic ECM update in a multi-layered environment. The dynamic obstacle is shown in yellow and black.

unlike in Chapter 5 in which we explored an entirely new problem domain. Instead, the goal of this chapter is to explain and solve the problem of dynamic updates step by step, with detailed pseudocode, in the context of the ECM.

We will also show that our implementation can update the ECM within milliseconds. This enables path planning and crowd simulation in dynamic environments in which obstacles are added and removed in real-time. Furthermore, in Chapter 8, we will present an algorithm that re-plans paths more efficiently after the navigation mesh has been updated.

Compared to our original publication on the dynamic ECM [154], we express the insertion algorithm and its pseudocode in terms of ECM cells, we discuss extensions and limitations more thoroughly, and we repeat our experiments without requiring GPU-based methods.

The remainder of this chapter is structured as follows:

- Section 6.2 describes related work on dynamic updates in Voronoi diagrams and navigation meshes.
- Section 6.3 presents an algorithm that inserts a point obstacle into the ECM.
- Section 6.4 extends this algorithm to insertions of line segments and convex polygons.
- Section 6.5 shows how to delete an obstacle from the ECM.
- In Section 6.6, we explain how the algorithms can be extended to handle e.g. intersecting obstacles and multi-layered environments.
- In Section 6.7, we perform experiments to show that our implementation can update the ECM within milliseconds.
- Section 6.8 concludes the chapter and discusses potential directions for future work.

6.2 Related Work

For general information on Voronoi diagrams (VDs), we recommend the reader to revisit Section 3.1. In this section, we will focus on dynamic environments and dynamically updated VDs.

6.2.1 Representing Dynamic Environments

Some data structures can handle obstacles that change over time. The *adaptive roadmaps* of Sud et al. [141] contain elastic edges that can change along with the environment. Roos and Noltemeier [128] have studied Voronoi diagrams of moving points. Kallmann and Matarić [70] describe *dynamic roadmaps* that keep track of the obstacles in the environment and constantly update a graph. In general, if the dynamic changes are known in advance (e.g. for moving obstacles with predefined trajectories), a time dimension can be added to the problem space to represent changes in the environment [92].

However, adding an extra dimension makes the problem space more complicated and less suitable for e.g. real-time path planning and crowd simulation. Also, we are interested in environments in which obstacles can be added and removed in arbitrary ways. For these purposes, it is more appropriate to use a navigation mesh that can be updated dynamically. At the time of writing this thesis, some navigation meshes other than the ECM support dynamic updates as well [41, 67, 68].

6.2.2 Updating Voronoi Diagrams

Green and Sibson [39] have developed an incremental algorithm for constructing VDs of point sites. Each iteration of this algorithm adds a single site; thus, such an iteration can be seen as a dynamic update. Since then, others have investigated numerically robust implementations [59] that can succesfully insert up to one million point sites [143]. Held [48, 49] has created robust implementations that also support line segments and circular arcs as sites.

Deletions of point sites from VDs have been studied by e.g. Devillers [25], Mostafavi et al. [103], and Gowda et al. [38]. Khramtcova and Papadopoulou [81] have shown that a *line segment* site can be deleted in $\mathcal{O}(m)$ time where m is the complexity of the segment's Voronoi cell. Concurrently with our work, De Moura Pinto and Dal Sasso Freitas [104] have implemented insertions and deletions for VDs of complex sites. Our results are similar, but more application-oriented.

6.3 Inserting a Point Obstacle

We first describe how to insert a *point* obstacle into a 2D ECM. In Section 6.3, we will extend this algorithm to convex polygon obstacles in 2D. Section 6.6 will study other extensions, e.g. to non-convex polygons and to multi-layered ECMs.

6.3.1 ECM Cells and Cell Relations

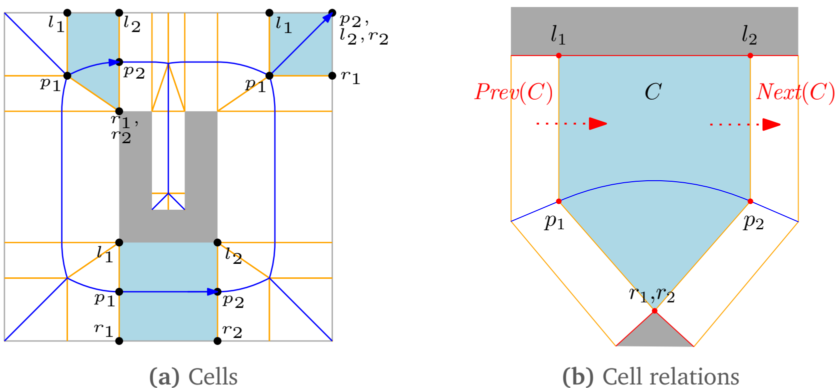
In Chapter 4, we have shown that the ECM subdivides $\mathcal{E}_{free}$ into polygonal *cells*, and that an ECM cell is defined by two subsequent *bending points* on the medial axis. Because our algorithms will be expressed in terms of ECM cells and their adjacency relations, we now define these concepts more precisely.

For an ECM cell C, we will refer to its two bending points as $bp_1(C)$ and $bp_2(C)$. We will omit the argument C whenever it is clear to which cell we are referring. Both bending points bp_i consist of a position p_i and two nearest obstacle points: l_i on the left side and r_i on the right side of the medial axis. Examples are shown in Figure 6.2a. Note that some of the points p_i, l_i, and r_i can coincide.

Because the ECM is an undirected graph, a cell can be specified in two directions, with two different senses of left and right. For each cell C, we define the *twin cell* $Twin(C)$ as the same cell specified in the other direction: $p_1(Twin(C)) = p_2(C)$, $p_2(Twin(C)) = p_1(C)$, $l_1(Twin(C)) = r_2(C)$, $l_2(Twin(C)) = r_1(C)$, $r_1(Twin(C)) = l_2(C)$, and $r_2(Twin(C)) = l_1(C)$. Note that $Twin(Twin(C)) = C$.

The *next cell* $Next(C)$ of an ECM cell C is the cell that continues along the same obstacle as C on the left side. It is the cell that lies immediately to the other side of the line segment $\overline{l_2(C)p_2(C)}$. If $l_2(C) = p_2(C)$, then C ends at a concave obstacle corner (such as in the top right of Figure 6.2a), and $Next(C) = \text{NULL}$. Likewise, the *previous cell* $Prev(C)$ of C is the ECM cell that lies to the other side of $\overline{l_1(C)p_1(C)}$. If $l_1(C) = p_1(C)$, then C begins at a concave obstacle corner, and $Prev(C) = \text{NULL}$.

An example of *Next* and *Prev* is shown in Figure 6.2b. Note that *Next* and *Prev* are dual operations: if $Next(C) \neq \text{NULL}$, then $Prev(Next(C)) = C$. These three operations resemble the traversal operations of a doubly-connected edge list (DCEL) [7]. We assume that they can all be performed in constant time; this is true for our own implementation of the ECM.



(a) Cells

(b) Cell relations

Figure 6.2: ECM cells and their relations. **(a)** Three examples of ECM cells are highlighted in blue. A cell is defined by two bending points, each with a position and a nearest obstacle point to the left and right. **(b)** Example of an ECM cell and its next and previous cells.

6.3.2 Background: Insertion in a Point Voronoi Diagram

Green and Sibson [39] have presented an incremental construction algorithm for Voronoi diagrams of point sites; this algorithm inserts the point sites one by one. Given a site p to insert and an existing Voronoi diagram of point sites, the algorithm iteratively traces the new Voronoi cell of p as follows:

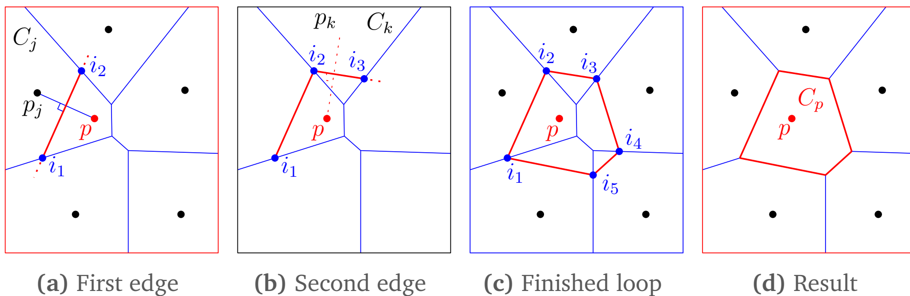
1. Find a Voronoi cell C_j that contains p. This cell identifies a nearest obstacle p_j to p.

2. Calculate the bisector of p and p_j. Let i_1 and i_2 be the two intersection points of this bisector with the boundary of the cell C_j. See Figure 6.3a.

3. The bisector from i_1 to i_2 is the first edge of the new cell C_p for p. At i_2, the bisector runs into an adjacent Voronoi cell, say C_k. This cell identifies a nearest obstacle p_k to the point i_2. Calculate the bisector of p and p_k, and let the intersections of this bisector with the boundary of C_k be i_2 and i_3. See Figure 6.3b.

4. Repeat this process to determine points i_3, i_4, and so on, until a bisector endpoint is found that returns to i_1. The resulting closed loop of bisectors will define the boundary of the new cell C_p. See Figure 6.3c.

5. Delete all old vertices and edges inside C_p to obtain the new Voronoi diagram. See Figure 6.3d.



(a) First edge **(b)** Second edge **(c)** Finished loop **(d)** Result

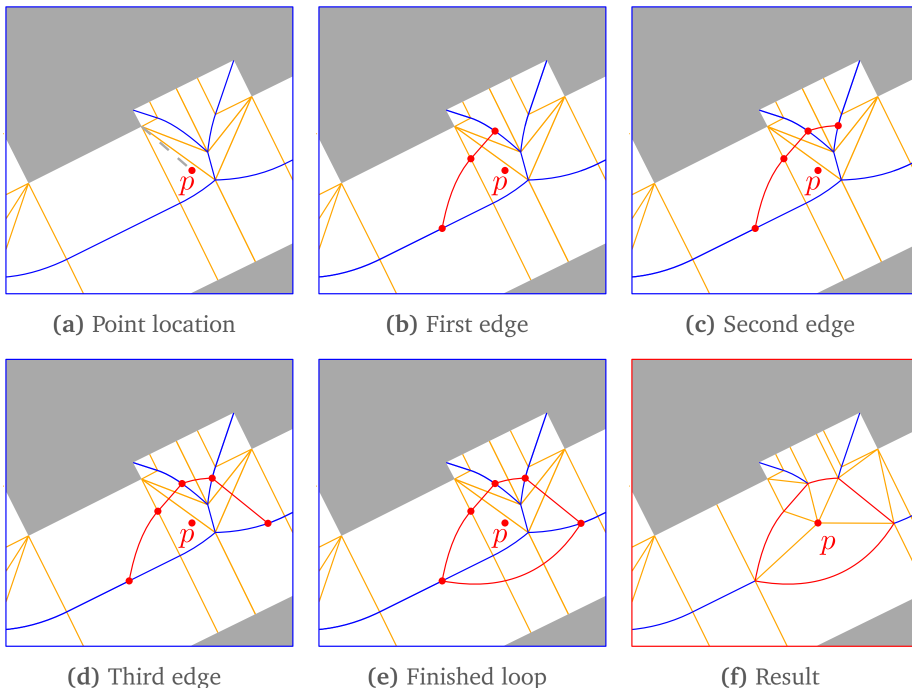
Figure 6.3: Inserting a point site p into a Voronoi diagram, as described by Green and Sibson [39]. The old Voronoi edges are shown in blue; the new edges are shown in red. The original algorithm built the new cell in counterclockwise order; we show a clockwise order to emphasize the analogy to our ECM algorithm.

In this algorithm, there is only one type of event: the intersection of a new bisector with an old Voronoi edge. However, if the neighboring Voronoi sites can be *line segments and polygons*, the algorithm becomes more complicated: bisectors can be parabolic arcs as well as line segments, and extra events occur when a bisector changes its shape.

6.3.3 Algorithm

Our algorithm for inserting a point obstacle p into the ECM is conceptually similar to the algorithm by Green and Sibson, but it uses the ECM cells to conveniently recognize all events. We trace the new ECM edges around p in clockwise order, always keeping p on the right side and the other nearest obstacle on the left side. A summary of the algorithm is shown in Figure 6.4. The full pseudocode is given in Algorithm 6.1. We will now describe the algorithm in detail.

We start by finding the cell C_0 that contains p. We use either C_0 or *Twin*(C_0) to ensure that the nearest obstacle point of C_0 to p lies on the *left* side. Next, we trace the first ECM edge in two directions (Lines 5–8). We then iteratively trace the next clockwise edge until the loop of edges around p is closed (Lines 9–12).



(a) Point location **(b)** First edge **(c)** Second edge

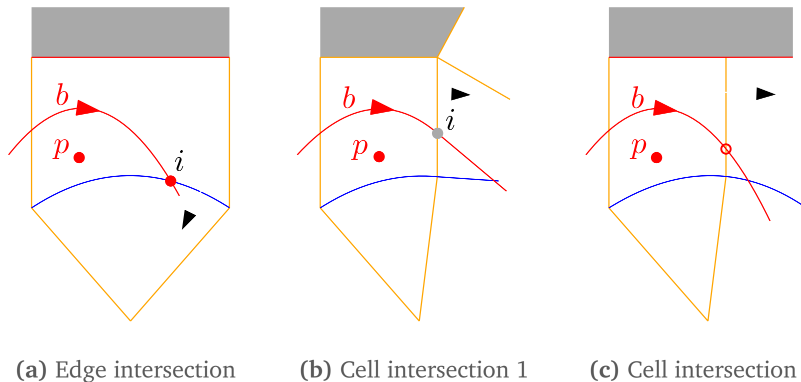
(d) Third edge **(e)** Finished loop **(f)** Result

Figure 6.4: Inserting a point obstacle p into the ECM. New edges are shown in red. We build the edges around p one by one, in clockwise order. Bending points (red dots) occur when an old ECM edge is intersected, or when the nearest obstacle changes between cells.

The subroutine for tracing a single new ECM edge is given in Algorithm 6.3. We follow the new edge until it intersects an old ECM edge. This is analogous to tracing a new Voronoi edge until it intersects the boundary of a Voronoi cell. Algorithm 6.3 iteratively computes the bisector b of p and the current nearest obstacle on the left side. In each iteration, the bisector b should exit the current ECM cell C at some point. There are two options:

(a) The bisector b intersects the existing ECM edge; see Figure 6.5a. If this happens, the currently traced ECM edge is finished: we add a last bending point to mark this event, and we return the final list of event points. In the next iteration, the obstacle on the *right* side of C will be the new closest obstacle. Therefore, the next call to TRACENEWEDGE-POINT will start with *Twin*(C) as its first cell. This case is handled in Lines 7–12 of Algorithm 6.3.

(b) The bisector b intersects the cell boundary and moves into the next ECM cell. If this happens, the currently traced edge is not yet finished, and the algorithm will go to the next cell. However, if this move causes the left obstacle to *change its shape* (i.e. from a line segment into a point, or vice versa), the bisector b will change its shape as well, and we need to add a bending point, as in Figure 6.5b. If the left obstacle does *not* change its shape, we simply proceed without adding a bending point, as in Figure 6.5c. We use a function ISRELEVANT(bp) to determine (in constant time) whether the left obstacle changes at bending point bp. The entire case is handled in Lines 13–23 of Algorithm 6.3.



(a) Edge intersection **(b)** Cell intersection 1 **(c)** Cell intersection 2

Figure 6.5: An iteration of TRACENEWEDGE-POINT. The bisector b of p and the left obstacle is shown in red. **(a)** If b intersects the existing ECM edge at a point i, the new edge ends at i. **(b)** If b intersects the cell boundary *and* the shape of the left obstacle changes, a bending point will appear at the intersection point i. **(c)** If b intersects the cell boundary, but the left obstacle does *not* change its shape, we simply move to the next cell.

Note that case (a) computes an intersection between two lines, a line and a parabola, or two parabolas. The first two options are easy to solve algebraically; the third option is more difficult in general. However, in our case, both bisectors will always have one nearest obstacle in common (namely the left obstacle), so the two parabolas will always have either the same focus or the same directrix. This allows us to compute these intersections algebraically as well.

The overall insertion algorithm ends when a new edge stops at the same position

Algorithm 6.1: INSERTPOINT(ECM, p)

1: *edges* $\leftarrow$ an empty list of edges

{Determine the first cell.}

2: $C_0 \leftarrow$ POINTLOCATION(ECM, p)
3: **if** p is closer to $\overline{r_1(C_0)r_2(C_0)}$ than to $\overline{l_1(C_0)l_2(C_0)}$
4: $\quad C_0 \leftarrow Twin(C_0)$

{Trace the first edge.}

5: $(edge_l, C_{prev}) \leftarrow$ TRACENEWEDGE-POINT($C_0, p,$ **false**, NULL)
6: $(edge_r, C_{next}) \leftarrow$ TRACENEWEDGE-POINT($C_0, p,$ **true**, NULL)
7: $edge_0 \leftarrow$ the concatenation of $edge_l$ and $edge_r$
8: Add $edge_0$ to *edges*

{Trace the other edges.}

9: **while** $C_{next} \neq C_0$
10: $\quad i_{start} \leftarrow$ the position of the last bending point that was added
11: $\quad (edge, C_{next}) \leftarrow$ TRACENEWEDGE-POINT($C_{next}, p,$ **true**, i_{start})
12: $\quad$ Add *edge* to *edges*

{Update the ECM.}

13: Add all elements from *edges* to the ECM
14: Remove all ECM edges inside the new edge loop

Algorithm 6.2: BISECTOR(l_1, l_2, r_1, r_2)

1: **if** $l_1 = l_2$ **and** $r_1 = r_2$
2: $\quad$ **return** the line bisector of l_1 and l_2
3: **if** $l_1 = l_2$ **and** $r_1 \neq r_2$
4: $\quad$ **return** the parabolic bisector of l_1 and $\overline{r_1 r_2}$
5: **if** $l_1 \neq l_2$ **and** $r_1 = r_2$
6: $\quad$ **return** the parabolic bisector of r_1 and $\overline{l_1 l_2}$
7: **return** the line bisector of $\overline{l_1 l_2}$ and $\overline{r_1 r_2}$

Algorithm 6.3: TRACENEWEDGE-POINT($C, p, \mathit{forward}, i_{start}$)

Input: A starting ECM cell C, the new point obstacle p, a flag *forward* (true or false), and the starting point i_{start} of the edge (optional).

Output: A sequence of new bending points describing the new ECM edge in forward or backward direction; plus the ECM cell in which the next iteration should start.

1: *result* $\leftarrow$ an empty list of bending points
2: **if** $i_{start} \neq$ NULL
3: Create a bending point at i_{start} and add it to *result*

4: **while true**
5: $b_{new} \leftarrow$ BISECTOR($l_1(C), l_2(C), p, p$)
6: $b_{old} \leftarrow$ BISECTOR($l_1(C), l_2(C), r_1(C), r_2(C)$)

{Check if b_{new} intersects the existing ECM edge.}
7: $i_{edge} \leftarrow$ the intersection of b_{new} and b_{old} in the correct direction
8: **if** $i_{edge} \neq$ NULL **and** i_{edge} lies inside C
9: Create a bending point at i_{edge} and add it to *result*
{Return the newly traced edge.}
10: **if** *forward* = **false**
11: *result*.REVERSE()
12: **return** (*result*, *Twin*(C))

{Otherwise, b_{new} must intersect the cell boundary.}
13: **if** *forward* = **true**
14: $j \leftarrow 2$
15: **else**
16: $j \leftarrow 1$
17: **if** ISRELEVANT($bp_j(C)$)
18: $i_{cell} \leftarrow$ the intersection of b_{new} and $\overline{p_j(C) l_j(C)}$
19: Create a bending point at i_{cell} and add it to *result*
20: **if** *forward* = **true**
21: $C \leftarrow$ *Next*(C)
22: **else**
23: $C \leftarrow$ *Prev*(C)

where the first edge started. Then, the loop of edges around p is finished. Just as in the algorithm by Green and Sibson, we remove the ECM edges and vertices inside this new loop, and we add the new loop to the graph (Lines 13–14).

After this dynamic update of the ECM, the next step is to update the point-location data structure such that future point-location queries will yield correct results. The details of this step are highly dependent of the chosen data structure and implementation; we will not explain it further here. In our experiments in Section 6.7, we *will* include this step in our running time measurements.

6.3.4 Complexity

The initial point location query for p takes $\mathcal{O}(\log n)$ time. The remainder of the running time depends on how much of the ECM is affected by the update. In the new ECM, let $m_i \in \mathcal{O}(n)$ be the number of cells traversed while constructing the new edges around p. All operations in a single cell take constant time; hence, the traversal takes $\Theta(m_i)$ time in total. Furthermore, when the new edges have been created, there are $\Theta(m_i)$ old edges and vertices to be removed.

Thus, inserting a point obstacle p into an ECM requires $\mathcal{O}(\log n + m_i)$ time, where m_i is the number of visited ECM cells. In practice, m_i is often much smaller than n, and a local insertion is very fast. In the worst case, large parts of the ECM are affected and m_i can be $\Theta(n)$, which gives a running time of $\Theta(n)$. This is still asymptotically faster than reconstructing the entire ECM (in $\mathcal{O}(n \log n)$ time).

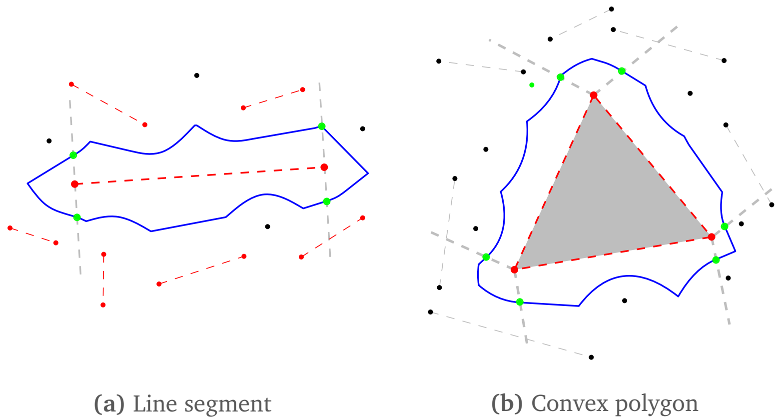
Note that this analysis does not yet include the post-processing step in which the point-location data structure is updated. Depending on the chosen data structure, the asymptotic complexity of this step may be higher than the complexity of the insertion itself. Dynamic point-location data structures form a separate research topic [4, 7, 19]. In general, there is a trade-off between the complexity of a query and the complexity of an update.

Our own implementation dynamically updates the grid-based point-location structure described in Section 4.4. The experiments in Section 6.7 will show that this implementation is fast in practice.

6.4 Inserting a Line Segment or Convex Polygon

We now extend the algorithm from Section 6.3 to handle the insertion of a line segment or *convex* polygon O that does *not* intersect the boundary of $\mathcal{E}_{free}$. Section 6.6 will explain how this algorithm can be extended to non-convex obstacles and intersecting geometry.

The main difference to adding a *point* obstacle is that O itself now generates bending points as well: in the updated ECM, the new edges around O will contain bending points when they cross the outward normals through the endpoints of O. This is illustrated in Figure 6.6.



(a) Line segment **(b)** Convex polygon

Figure 6.6: The medial axis edges around a line segment site and a convex polygon site. Compared to point sites, these sites generate extra bending points wherever an edge crosses an outward normal. These extra bending points are shown in green.

In theory, a line segment site can be inserted into a Voronoi diagram by first inserting the endpoints of the segment and then its interior [49]. This technique is used in multiple software libraries for computing Voronoi diagrams of line segments [48, 78]. A polygon could be inserted by inserting the edges of the polygon one by one and then (optionally) removing the Voronoi diagram in the polygon's interior. In this section, we instead explain how to insert a line segment or convex polygon using a single algorithm that extends InsertPoint.

6.4.1 Algorithm

Algorithms 6.4 to 6.7 give the pseudocode for inserting a line segment or polygon into the ECM. The parts that are equal to InsertPoint are shown in gray; the differences are shown in black.

The main difference is due to the bending points generated by O: the subroutine TraceNewEdge-Polygon (Algorithm 6.5) should now consider a third possible event in each iteration. Just like in TraceNewEdge-Point, the current bisector b can intersect a medial axis edge or a cell boundary: in these cases, the nearest obstacle on the *left* side changes. The new third option is that b intersects a surface normal through a vertex of O: in this case, the nearest site on the *right* side changes. To compute this intersection (if it exists) in constant time per iteration, the algorithm needs to keep track of the part of O that is generating the current bisector. This part is always either a vertex or an edge of O, and it changes whenever the third type of event is triggered.

The other significant difference to InsertPoint is the *initialization step* of the algorithm, in which we need to find a cell and bisector to start with. For a point

site p, we found the starting cell C_0 by performing a point-location query for p, and we knew for sure that p would generate a medial axis arc inside C_0. For a polygon site O, the situation is more complicated. A first idea would be to perform a point-location query for an arbitrary vertex p of O. However, p does not necessarily generate medial axis arcs in the ECM cell that is found: other points of O may be closer to the existing geometry than p is. Therefore, an alternative and non-trivial solution is needed.

Let p^* be a point on the boundary of O that has (of all boundary points of O) the smallest distance to *any* existing obstacle in the environment. Note that p^* is not necessarily a *vertex* of O; it can lie anywhere on the boundary of O. Let $n^* = np(p^*)$ be the nearest obstacle point to p^* according to the ECM. As shown in Figure 6.7, the disk with diameter $\overline{p^*n^*}$ must be empty (i.e. obstacle-free): otherwise, either n^* would not be nearest to p^*, or there would be another point of O that is closer to other obstacles. Therefore, the center c^* of this disk will have two distinct equidistant nearest obstacle points after O has been inserted. By definition of the medial axis, c^* will lie on the medial axis after the insertion. This makes it a valid starting point for our algorithm.

Thus, to start the dynamic insertion, we need to find p^* and its nearest obstacle point n^*. As explained in Chapter 4, for any point $q \in \mathcal{E}_{free}$, the nearest obstacle point $np(q)$ to q lies on the boundary of the ECM cell that contains q. We can therefore find p^* and n^* by tracing the boundary of O and checking all ECM cells that are visited during this traversal. Whenever we move to a different cell, we compute the distance from O to the corresponding nearest obstacle.

This is almost equivalent to performing a mutual visibility query with clearance (Section 4.3.6) for each edge of O. The difference is that we are now not interested in the *value* of the lowest clearance along these edges, but in the *point* at which this lowest clearance occurs. This is the point p^*.

We will not provide the pseudocode for this initialization step; instead, Algorithm 6.4 summarizes it as FINDSTARTCELL. This procedure returns the ECM cell C_0 in which p^* and n^* are located. It also returns the edge or vertex of O that contains p^*, given by two points p_1 and p_2. If p^* is a vertex of O, then $p_1 = p_2 = p^*$. Otherwise, p_1 and p_2 are the endpoints of the edge that contains p^*.

6.4.2 Complexity

The algorithm INSERTPOLYGON runs in $\mathcal{O}(m_i + n' + \log n)$ time, where m_i is the number of visited ECM cells and n' is the number of vertices of the polygon that is being added. After all, each of the n' vertices generates two bending points, as can be seen in Figure 6.6b. This leads to $\Theta(n')$ more events in total. In each iteration within TRACENEWEDGE-POLYGON, it takes constant time to check if such an event occurs, and constant time to handle the event if it does occur.

Finally, we expect that n' will typically be a small number in practical applications, i.e. that the new obstacle will have a relatively simple shape.

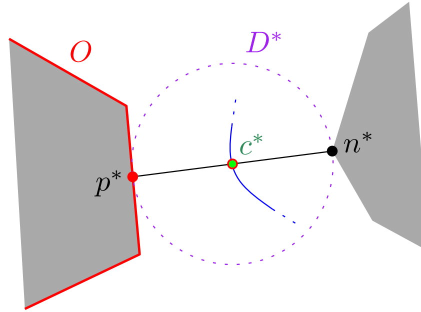


Figure 6.7: The starting point for inserting a dynamic obstacle O. We find the point p^* on the boundary of O (shown in red), and its nearest obstacle point n^* (shown in black), as described in the text. The disk D^* (purple) must be empty. Therefore, its center c^* (green) must lie on a new medial axis edge (blue).

Algorithm 6.4: INSERTPOLYGON(ECM, O)

1: $edges \leftarrow$ an empty list of edges
2: $(C_0, p_1, p_2) \leftarrow$ FINDSTARTCELL(ECM, O)

{Trace the first edge.}
3: $(edge_l, C_{prev}, p'_1, p'_2) \leftarrow$ TRACENEWEDGE-POLYGON($C_0, O, p_1, p_2,$ **false**, NULL)
4: $(edge_r, C_{next}, p_1, p_2) \leftarrow$ TRACENEWEDGE-POLYGON($C_0, O, p_1, p_2,$ **true**, NULL)
5: $edge_0 \leftarrow$ the concatenation of $edge_l$ and $edge_r$
6: Add $edge_0$ to $edges$

{Trace the other edges.}
7: **while** $C_{next} \neq C_0$
8: $\quad i_{start} \leftarrow$ the position of the last bending point that was added
9: $\quad (edge, C_{next}, p_1, p_2) \leftarrow$ TRACENEWEDGE-POLYGON($C_{next}, O, p_1, p_2,$ **true**, i_{start})
10: $\quad$ Add $edge$ to $edges$

{Update the ECM.}
11: Add all elements from $edges$ to the ECM
12: Remove all ECM edges inside the new edge loop
13: Update the point location data structure for future queries

Algorithm 6.5: TRACENEWEDGE-POLYGON(C, O, p_1, p_2, *forward*, i_{start})

Input: A starting ECM cell C, the obstacle O to insert, the two endpoints p_1 and p_2 of O that generate the current bisector, a flag *forward* (true or false), and the starting point i_{start} of the edge (optional).

Output: A sequence of new bending points describing the new ECM edge in forward or backward direction; plus the ECM cell in which the next iteration should start; plus the part of P that generates the bisector at the end of this algorithm.

```
 1: result ← an empty list of bending points
 2: if i_start ≠ NULL
 3:     Create a bending point at i_start and add it to result

 4: while true
 5:     b_new ← BISECTOR(l1(C), l2(C), p1, p2)
 6:     b_old ← BISECTOR(l1(C), l2(C), r1(C), r2(C))

        {Check if b_new intersects a normal of P; check if it occurs before i_edge}
 7:     i_normal ← the intersection of b and NEXTNORMAL(O, p1, p2, forward)
 8:     if i_normal ≠ NULL and i_normal lies inside C and i_normal lies to the left of b_old
 9:         Create a bending point at i_normal and add it to result
10:         (p1, p2) ← NEXTSITE(O, p1, p2, forward)
11:         continue

        {Check if b_new intersects the existing ECM edge.}
12:     i_edge ← the intersection of b_new and b_old in the correct direction
13:     if i_edge ≠ NULL and i_edge lies inside C
14:         Create a bending point at i_edge and add it to result
            {Return the newly traced edge.}
15:         if forward = false
16:             result.REVERSE()
17:         return (result, Twin(C), p1, p2)

        {Otherwise, b_new must intersect the cell boundary.}
18:     Identical to Lines 13–23 of TRACENEWEDGE-POINT (Algorithm 6.3)
```

Algorithm 6.6: NEXTNORMAL(O, p_1, p_2, *forward*)

```
1:  if p₁ = p₂
2:      o ← p₁
3:      if forward = true
4:          dir ← the second clockwise normal of O at p₁
5:      else
6:          dir ← the first clockwise normal of O at p₁
7:  else
8:      dir ← the left normal of p₁p₂ (segment)
9:      if forward = true
10:         o ← p₁
11:     else
12:         o ← p₂
13: return the half-line with origin o and direction dir
```

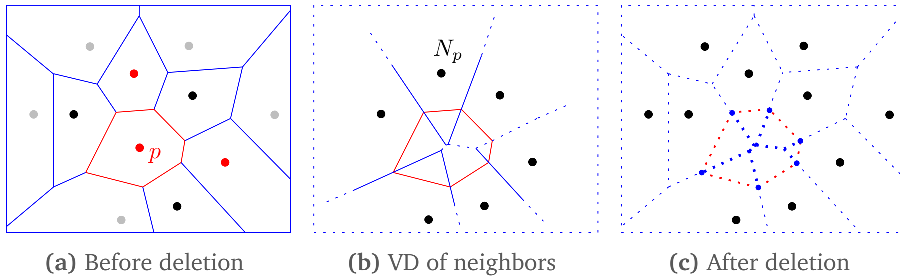
1: **if** $p_1 = p_2$
2: $\quad o \leftarrow p_1$
3: $\quad$ **if** *forward* = **true**
4: $\quad\quad$ *dir* $\leftarrow$ the second clockwise normal of O at p_1
5: $\quad$ **else**
6: $\quad\quad$ *dir* $\leftarrow$ the first clockwise normal of O at p_1
7: **else**
8: $\quad$ *dir* $\leftarrow$ the left normal of $\overline{p_1 p_2}$
9: $\quad$ **if** *forward* = **true**
10: $\quad\quad o \leftarrow p_1$
11: $\quad$ **else**
12: $\quad\quad o \leftarrow p_2$
13: **return** the half-line with origin o and direction *dir*

Algorithm 6.7: NEXTSITE($O, p_1, p_2, forward$)

1: **if** $p_1 = p_2$
2: $\quad$ **if** *forward* = **true**
3: $\quad\quad p_3 \leftarrow$ the vertex of O after p_2 (clockwise)
4: $\quad\quad$ **return** (p_2, p_3)
5: $\quad$ **else**
6: $\quad\quad p_0 \leftarrow$ the vertex of O after p_1 (counter-clockwise)
7: $\quad\quad$ **return** (p_0, p_1)
8: **else**
9: $\quad$ **if** *forward* = **true**
10: $\quad\quad$ **return** (p_2, p_2)
11: $\quad$ **else**
12: $\quad\quad$ **return** (p_1, p_1)

6.5 Deleting an Obstacle

We now describe how to *delete* an obstacle O from the ECM. For Voronoi diagrams (VDs) of point sites, deletions of sites are well-studied [5, 25, 103, 111]. In summary, a point site p can be deleted as follows: find the set of sites N_p that generate a Voronoi edge with p (i.e. the Voronoi neighbors of p), compute the VD of N_p from scratch, and replace the old Voronoi cell of p by edges from this new VD. This is illustrated in Figure 6.8.



(a) Before deletion **(b)** VD of neighbors **(c)** After deletion

Figure 6.8: Deleting a point site p from a Voronoi diagram, as described by Devillers [25], among others. **(a)** Locate the set N_p of neighboring sites of p, which are shown in black. **(b)** Compute the Voronoi diagram (VD) of N_p. **(c)** Remove the Voronoi edges around p, and merge the VD of N_p into the main graph.

The same concept applies to deletions of points, line segments, and convex polygons from the ECM. Let O be the obstacle that needs to be removed, and assume that it does not intersect other obstacles, such that there is a closed loop of ECM edges around O. Let Z_O be the area enclosed by this loop of edges. We only need to update the ECM inside (and on the boundary of) Z_O because the nearest obstacles cannot have changed anywhere else. Furthermore, to compute the correct new ECM inside Z_O, we only need the obstacle parts that currently generate an ECM edge with O. All other obstacles are too far away to suddenly become a nearest obstacle in Z_O. In Chapter 5, we have used (and have proved) similar arguments for opening a connection in the multi-layered ECM. The remainder of the deletion algorithm is exactly the same as the algorithm for deleting a point from a VD.

Let m_d be the total number of ECM cells on the edges around O. The set of neighboring obstacle parts N_O has size $\mathcal{O}(m_d)$; it can be found in $\mathcal{O}(m_d)$ time by traversing the ECM edges around O and collecting the obstacle points and line segments that lie on the other side of these edges. Computing the ECM of N_O takes $\mathcal{O}(m_d \log m_d)$ time. Combined with an initial point-location query, the complete algorithm runs in $\mathcal{O}(\log n + m_d \log m_d)$ time. If O has a pointer to one of its surrounding ECM edges, then the factor $\mathcal{O}(\log n)$ can be removed because a point-location query is no longer required.

Others have shown that deletions can theoretically be performed in $\mathcal{O}(m_d)$ time for Voronoi diagrams of points [1] or line segments [81]. Their key observation is that the neighboring sites are already logically ordered around the site to delete; this extra information allows the VD of these neighbors to be computed more efficiently. Therefore, it is likely that the running time of our deletion algorithm can be improved to $\mathcal{O}(m_d)$ time as well. We will not explore this further in this thesis.

6.6 Extensions

In this section, we discuss how the algorithms from the previous sections can be extended to handle other types of obstacles and environments. This will be a high-level discussion in which we sketch possible solutions.

6.6.1 Non-Convex Obstacles

A non-convex polygonal obstacle O is not surrounded by a single loop of ECM edges. Instead, there are extra ECM edges that run into the concave corners of O. Therefore, such an obstacle cannot be inserted by using the algorithm from Section 6.4. Conceptually, the easiest solution is to subdivide O into convex parts and then inserting these convex parts one by one. In practice, this may be difficult to implement robustly because the parts of O will have edges in common, which leads to degenerate cases at which obstacle edges coincide.

Deleting a non-convex polygon O can be achieved by using the algorithm from Section 6.5. There is one difference, though: points on the boundary of O can now be Voronoi neighbors. Thus, when collecting the neighboring obstacles of O, we need to ignore the obstacle points that belong to O itself. The remainder of the deletion algorithm is conceptually the same as for convex polygons.

6.6.2 Intersecting Geometry

If we want to insert a line segment or polygon obstacle O that intersects the boundary of $\mathcal{E}_{free}$, then the new ECM edges around this obstacle will not form a single loop. When tracing these new edges, the algorithm should pause when the boundary of O first intersects an obstacle. This a fourth type of event that can be added to the TRACENEWEDGE-POLYGON routine. Next, the algorithm should follow the boundary of O until it enters $\mathcal{E}_{free}$ again. At that point, a new edge begins.

Ideally, *deleting* such an obstacle should only generate ECM edges in the areas that were obstacle-free before O was added. However, the ECM does not preserve such 'history' information: it knows only which areas are currently walkable. If the deletion algorithm is based purely on the ECM, it cannot know if there are any other obstacles 'below' O that may become relevant again. The result is that the entire surface of O will be added to $\mathcal{E}_{free}$.

One way to preserve this information is to also maintain the ECM in the *interior* of each obstacle. This way, the intersection points of all obstacle boundaries are also represented in the ECM. A similar approach has been used successfully for the Local Clearance Triangulation by Kallmann [67]. To know which obstacle continues where at an intersection point, the ECM bending points should store references to specific obstacles rather than just obstacle coordinates.

However, this approach complicates the navigation mesh with extra information about obstacles and about areas that characters cannot reach. Furthermore, it cannot trivially be extended to *multi-layered* environments because the obstacle space of an MLE is defined per layer, and the obstacles of different layers are not logically connected. In Section 6.8, we will suggest alternative solutions for future work.

6.6.3 Multi-Layered Environments

Our insertion algorithm also works in the multi-layered environments (MLEs) from Chapter 5 because the algorithm is based on individual ECM cells and their relations only. Thus, it does not matter that the overall MLE can overlap itself when projected onto the ground plane. The dynamic obstacle can even overlap with a connection and lie in multiple layers; this does not affect the algorithm, as long as a starting point for the insertion can be found.

As stated in Chapter 5, if we want to ensure that each ECM edge belongs to a single layer, we need to split edges where they intersect a connection. We can do this either during the update itself (as an extra type of event in TRACENEWEDGE-POLYGON) or in a post-processing step.

In general, a dynamic *deletion* from an MLE is more difficult, for reasons similar to why opening a connection (Section 5.6) is an involved process. If the neighboring obstacles of the dynamic obstacle O all lie in the same layer, then our 2D deletion algorithm will work immediately. However, if the neighboring obstacles lie in multiple layers, projecting them all onto the ground plane P may produce incorrect results, just like in the naive algorithm from Section 5.6.

To prevent this, we would need to include all relevant connections as neighboring obstacles, construct the ECM of these neighbors with all connections closed, and then open the connections as in Chapter 5. We leave the details of such an algorithm for future work. Our experiments in Section 6.8 will focus on 2D environments only.

6.7 Experiments and Results

We have implemented our algorithms for dynamic updates in our ECM software, and we have performed experiments to measure the efficiency of these algorithms. This section describes these experiments and their results.

A dynamic deletion involves the construction of an ECM for a set of neighboring obstacles. For this step, we used the Boost-based implementation of the ECM construction algorithm, described in Section 4.4.

6.7.1 Inserting and Deleting Random Points

The main purpose of our first experiment is to show that a local update is often faster than a complete reconstruction of the ECM. We tested the performance of insertions and deletions of point sites in an environment of 100×100 m with only a bounding box. One point at a time, we obtained a uniformly sampled random position p with a distance of at least 0.2m to the bounding box and to all previously added points, and we dynamically inserted a point obstacle at p into the ECM. We repeated this process, thus gradually increasing the complexity of the environment, until 500 points had been added. Next, we deleted the obstacles from the ECM in opposite order. We repeated the experiment 25 times, with a different set of points each time.

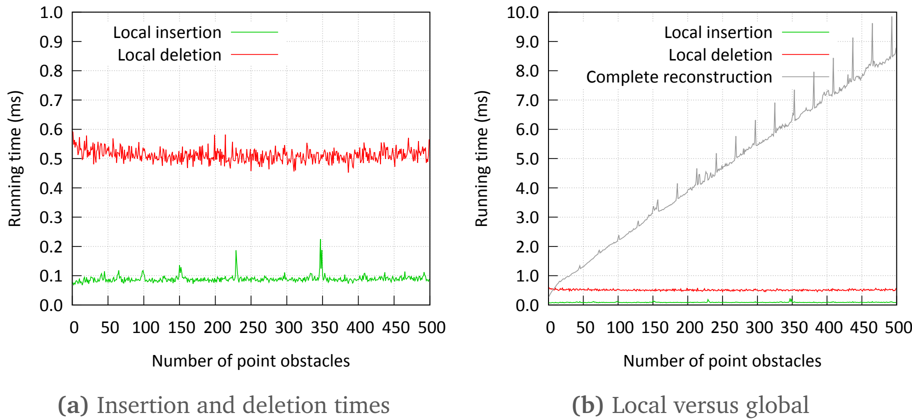
The running times are shown in Figure 6.9a. Dynamic insertions took 0.09 ms on average ($\sigma = 0.01$); dynamic deletions took 0.51 ms on average ($\sigma = 0.02$). For comparative purposes, we performed a similar experiment in which we *recomputed the ECM from scratch* in response to each point insertion, using the Boost-based ECM implementation. This time, the running time quickly increased with the number of points, as shown in Figure 6.9b. This confirms that local updates are useful, especially if the environment grows in complexity.

Figure 6.9a suggests that the running times of our local algorithms remain roughly constant, regardless of the number of obstacles in the environment. In theory, though, dynamic insertions should become gradually slower as the number of points increases: the initial point-location query adds a $\mathcal{O}(\log n)$ factor to the insertion algorithm's running time. Our running times only *appear* constant due to our grid-based point-location data structure combined with a relatively small number of points. However, the purpose of this experiment was to compare local updates to complete reconstruction.

6.7.2 Inserting and Deleting Polygons

Next, we locally inserted and deleted polygonal obstacles in the *Military* and *City* environments from Section 4.5. We manually chose these obstacles such that their dynamic insertions and deletions varied in complexity. The obstacles (27 for *Military* and 89 for *City*) are shown in Figure 6.10. We inserted and deleted the obstacles one by one: after having inserted an obstacle, we deleted it again before continuing with the next obstacle. Thus, the environments always contained at most one dynamic obstacle at a time.

The average insertion and deletion times per obstacle (averaged over 25 runs) are shown in Figure 6.11. In *Military*, all insertions took below 0.5 ms on average, and deletion times varied between 0.5 ms and 2.7 ms depending on the complexity



(a) Insertion and deletion times

(b) Local versus global

Figure 6.9: Running times for insertions and deletions of random point obstacles in an empty environment. **(a)** The horizontal axis denotes the number of point obstacles that were in the environment at the time of insertion or deletion. The vertical axis denotes the running time (in milliseconds) for dynamically inserting or deleting the next point obstacle. **(b)** The same running times compared to recomputing the ECM from scratch.

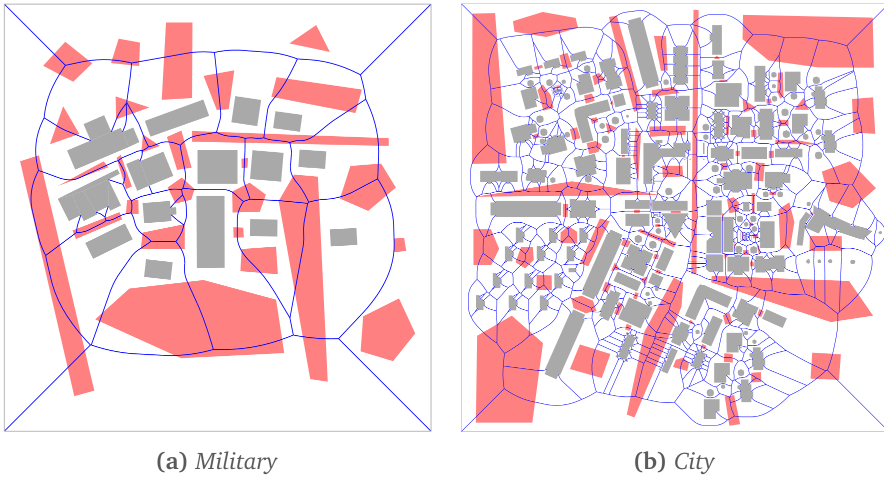
of the dynamic event. In *City*, we could create more complex situations by drawing long obstacles that span a large part of the environment. Here, the most 'difficult' obstacle had a loop of 163 ECM cells around it; on average, it took 2.4 ms to insert and 9.6 ms to delete. Most other obstacles had a smaller influence: between 10 and 60 ECM cells, insertion times below 1 ms, and deletion times below or around 3 ms. For comparison, constructing the ECM from scratch takes around 6.7 ms and 130 ms for *Military* and *City*, respectively (see Section 4.5). Thus, even our most complex dynamic updates were faster than completely reconstructing the ECM.

Because the navigation mesh can be updated dynamically within milliseconds, our algorithms make the ECM suitable for real-time crowd simulation in dynamic environments. In Chapter 8, we will study the problem of re-planning paths in a dynamic navigation mesh.

Chapter 10 will describe how an overall crowd simulation framework can process dynamic updates and re-planning queries during a real-time simulation. Processing the dynamic update *immediately* may briefly slow down the simulation, especially if the crowd is large. By contrast, distributing the updates and re-planning queries over time (possibly in separate threads) can prevent these 'hiccups', but it implies that characters will not respond immediately.

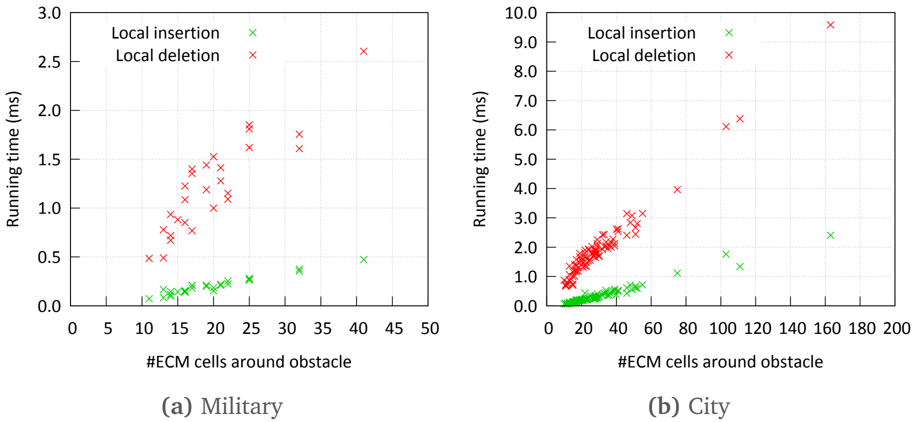
6.8 Conclusions and Future Work

In modern games and simulations that feature (crowds of) virtual characters, the environment can change dynamically: obstacles can appear, disappear, or move at



(a) *Military* (b) *City*

Figure 6.10: The test environments, their medial axis (shown in blue), and their dynamic obstacles (shown in semi-transparent red). The nearest-obstacle annotations of the ECM have been omitted for clarity.



(a) Military (b) City

Figure 6.11: Running times for insertions and deletions of polygons in **(a)** Military and **(b)** City. Each data point corresponds to one of the obstacles in Figure 6.10. The horizontal axis denotes the number of ECM cells in the loop of ECM edges around a dynamic obstacle. The vertical axis denotes the running time (in milliseconds) for dynamically inserting or deleting an obstacle.

run-time. When this happens, the navigation mesh that represents the free space should be updated, preferably in an efficient manner.

In this chapter, we have presented algorithms that update the Explicit Corridor Map (ECM) navigation mesh from Chapters 4 and 5 when an obstacle has been added or removed. Our algorithms are based on existing algorithms for insertions and deletions of sites in Voronoi diagrams. In a 2D environment of complexity n, a convex polygon with n' vertices can be inserted into the ECM in $\mathcal{O}(m_i + n' + \log n)$ time, where $m_i \in \mathcal{O}(n)$ is proportional to the number of edges that are added and removed. An obstacle can be deleted in $\mathcal{O}(\log n + m_d \log m_d)$ time where m_d is the number of ECM cells around the obstacle. Our experiments show that our implementation of these algorithms can update the ECM within milliseconds, which is generally much faster than recomputing the navigation mesh from scratch. Therefore, our local algorithms can be used to model environments with multiple dynamic obstacles in real-time.

After the navigation mesh has been updated, the characters in the simulation need to *re-plan* their paths. A naive but correct solution is to re-plan all paths from scratch. In Chapter 8, we will present Dynamically Pruned A*, a variant of the A* search algorithm optimized for handling dynamic events.

Discussion and future work. As discussed in Section 6.6, our algorithms currently have a number of limitations. Some of these limitations can be overcome, but others are inherent to the fact that we use only the navigation mesh and not the original geometry of the environment. In particular, it is difficult to handle arbitrary deletions in multi-layered environments, or deletions of obstacles that overlap with other obstacles.

Furthermore, although our algorithms are theoretically simple, we have experienced that they are difficult to implement robustly. Our current implementation works in almost all cases, but a dynamic update may still sporadically fail, mainly due to our use of imprecise floating-point numbers. This makes the implementation slightly risky for applications in which not all possible events can be tested beforehand. It also prevented us from performing experiments with e.g. hundreds of thousands of points sites, which has been done successfully before [143]. It is difficult to create robust implementations of Voronoi diagram operations or geometric algorithms in general. Existing techniques for numerically robust Voronoi diagram construction [49, 59, 78, 143], such as the use of exact number types, can most likely be applied to improve our implementation in the future.

An alternative solution would be to recompute the ECM of the entire environment, e.g. in a separate thread while the simulation keeps running. Of course, this implies that characters cannot immediately respond to the update. We may be able to improve performance by pinpointing the area in which the ECM changes and then recomputing the ECM from scratch in this area only. Such a 'hybrid' method would be an interesting alternative to the algorithms from this chapter. It may be easier to implement based on existing robust software libraries, and it would automatically support the extensions from Section 6.6.

7 A Comparative Study of Navigation Meshes

In this chapter, we develop ways to measure the quality of a navigation mesh and its construction algorithm for an input environment. We use them to compare various state-of-the-art navigation meshes, including the ECM, for a range of 2D and 3D environments.

This chapter is based on the following publication:

- W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th ACM SIGGRAPH International Conference on Motion in Games*, pages 91–100, 2016. [152]

7.1 Introduction

As indicated in Part I of this thesis, modern simulations and games feature increasingly large crowds of moving virtual characters. This leads to many queries related to e.g. path planning, path following, point location, and collision avoidance [151]. The simulation is expected to run in real-time despite all these demands. This stresses the need for high-quality data structures and algorithms.

In our domain, the virtual environment can be three-dimensional, but characters are constrained to surfaces that are sufficiently flat to walk on. The walkable surfaces of an environment form the *free space* $\mathcal{E}_{free}$. A *navigation mesh* is a representation of $\mathcal{E}_{free}$ as a set of (usually polygonal) regions, along with a graph that describes how these regions are connected. For path planning, characters first find an optimal path in the graph and then compute a suitable geometric route through the corresponding regions.

Various state-of-the-art navigation meshes exist that are *automatically* constructed from the input geometry, but there is no standardized way of evaluating or comparing them. Each navigation mesh implementation is in a different state of maturity, has been tested on different hardware, uses different example environments, and may have been designed with a different application in mind. To steer subsequent research into interesting directions, an objective comparison between navigation meshes is required.

In this chapter, we conduct the first comparative study of navigation mesh implementations by using the same hardware, quality metrics, and input environments

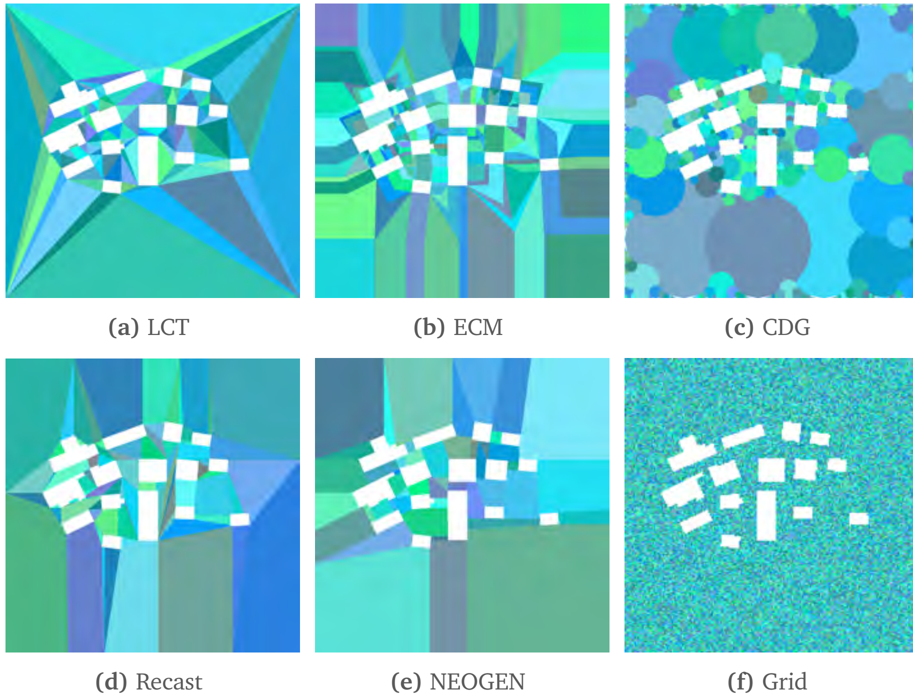


Figure 7.1: Navigation meshes computed for the *Military* environment. Regions are shown in different colors. The corresponding graphs have been omitted for clarity.

for all methods. Our study includes the Local Clearance Triangulation (LCT) [67], the Explicit Corridor Map (ECM), the Clearance Disk Graph (CDG) [121], Recast Navigation [102], NEOGEN [114], and a grid. To convey a first impression, Figure 7.1 shows the output of each navigation mesh for one 2D environment.

We propose a way to objectively measure and compare the quality of navigation meshes for 2D and 3D environments. Because navigation meshes have many applications with different requirements, it is difficult to propose a single criterion that can identify 'the best' navigation mesh. Instead, we present a collection of criteria, each of which is relevant for particular applications.

Our main goal is *not* to expose which navigation mesh implementation works best in general; each navigation mesh may have advantages in particular environments. By comparing navigation meshes using a common test platform and settings, we expect that this work will set a standard for the analysis and evaluation of navigation meshes, that it will help developers choose an appropriate navigation mesh for their application, and that it will steer research on navigation meshes in interesting directions.

The remainder of this chapter is structured as follows:

- Section 7.2 describes the relevant related work on navigation meshes and

comparative studies.

- Section 7.3 gives general definitions of environments and navigation meshes to enable an objective comparison.
- Section 7.4 proposes theoretical properties by which the data structures and algorithms of navigation meshes can be classified.
- Section 7.5 uses these properties to analyze and compare various state-of-the-art navigation meshes.
- In Section 7.6, we present quantitative metrics to measure the quality and performance of a navigation mesh implementation for an input environment. In particular, we introduce and address the concept of coverage in 3D.
- In Section 7.7, we combine these metrics into a benchmark tool, and we use it to compare state-of-the-art navigation mesh implementations in a range of 2D and 3D environments.
- Section 7.8 summarizes the chapter and lists a number of interesting directions for future research.

7.2 Related Work

This section summarizes the most relevant related work on navigation meshes and comparative studies. For more general references on navigation meshes in 2D and 3D, we refer the reader to Chapter 2.

7.2.1 Navigation Meshes in 3D

In Section 2.3, we have explained that there are two categories of construction algorithms for 3D navigation meshes. We will briefly repeat and rephrase this explanation here.

Voxel-based methods [24, 102, 114, 121] usually take an unprocessed 3D environment as their input. They discretize the environment into a 3D grid of *voxels* using GPU techniques, extract the voxels that correspond to walkable regions, and summarize this information in a navigation mesh that *approximates* the geometry of $\mathcal{E}_{free}$. Voxel-based methods can handle arbitrary 3D geometry with imperfections such as intersections or small gaps between polygons. However, the precision and efficiency of these methods depends to a certain degree on the grid resolution. The quality of the navigation mesh depends on how well the free space is extracted from the 3D geometry.

Exact methods [33, 43, 67, 112, 153, 156] require that the exact geometry of $\mathcal{E}_{free}$ is known, and that this free space has been pre-processed into one or more planar layers. In exchange, they represent their input *precisely*, and they often have provable worst-case construction times and storage sizes, which implies better

scalability to large environments. However, extracting $\mathcal{E}_{free}$ from a 3D environment without using approximations is still a topic of ongoing research [123].

7.2.2 Comparative Studies

Our comparative study of navigation meshes is inspired by comparisons for *other* aspects of path planning and crowd simulation.

Sturtevant [139] has developed a test set of environments for 2D *grid-based* path planning. This set includes mazes of various complexities and levels from computer games, and it is often used to analyze variants of the A* search algorithm [45]. Although we study general navigation meshes rather than grids, we will also include grid-based environments in our experiments.

SteerBench [135] focuses on local behavior such as collision avoidance. It presents a comprehensive set of scenarios that local methods are expected to solve, such as two characters crossing paths, or characters switching places in a narrow corridor. Given the output of a crowd simulation for such a scenario, SteerBench can compute metrics such as the distance that all characters traverse and the amount of energy that they spend. However, the results need to be put in perspective because steering methods typically have many parameters and implementation choices.

In this chapter, we compare navigation meshes in a similar way based on metrics, input environments, and a single test platform. We will see that parameter settings are influential in our study as well.

7.3 Definitions

In this section, we give definitions of environments and navigation meshes.

7.3.1 Environments

Our comparative study focuses on the same types of environments as Chapters 4 and 5. To make the remainder of our comparative study easier to follow, we will briefly repeat the definitions from these chapters.

A *3D environment* (3DE) is a raw collection of polygons in $\mathbb{R}^3$. A main assumption is that there is a single direction of gravity $\vec{g}$ throughout the environment, perpendicular to a common *ground plane* P. The free space $\mathcal{E}_{free}$ of a 3DE is determined by parameters that describe on which surfaces a character may walk, such as the maximum slope with respect to $\vec{g}$.

Characters are typically approximated by cylinders. Some navigation meshes use a predefined character radius to determine $\mathcal{E}_{free}$. In this chapter, we will use a radius of zero to enable an objective comparison to other navigation meshes.

A *walkable environment* (WE) is a set of interior-disjoint polygons in $\mathbb{R}^3$ on which characters can stand and walk. Thus, a WE represents the free space $\mathcal{E}_{free}$ of a 3DE. In our experiments, all environments will be WEs, so we know beforehand which area should be covered by a navigation mesh. This is required for computing the coverage metrics that will be described in Section 7.6.1.

The main difference to a 2D environment is that a WE can overlap itself when projected onto the ground plane P, i.e. it is not guaranteed that all surfaces are visible from a single top view.

Some navigation meshes require the WE to be subdivided into 2D components. A *multi-layered environment* (MLE) [121, 153] is a subdivision of a WE into *layers* such that the walkable polygons of each individual layer are non-overlapping when projected onto P. The layers are connected by line segments. An example is shown in Figure 7.2a. In our experiments, we will subdivide all WEs into layers to facilitate the construction of navigation meshes.

Finally, for *2D environments*, the definition used in this chapter is slightly different to that of Chapter 4 because we want to treat 2D environments and WEs similarly. Therefore, we define a 2D environment as a bounded subset of the 2D plane with a polygonal boundary and polygonal holes; we refer to these holes as obstacles. Unlike in Chapter 4, we will not consider point or line segment obstacles. The obstacle space $\mathcal{E}_{obs}$ is the union of all obstacles. Its complement is the free space $\mathcal{E}_{free}$. The *complexity* of $\mathcal{E}$ is the number of vertices n required to define $\mathcal{E}_{obs}$ or $\mathcal{E}_{free}$ using simple polygons.

We will embed our 2D environments in $\mathbb{R}^3$ by assigning a height component of zero to each vertex. This way, a 2D environment is a special case of an MLE with only one layer (or, equivalently, a WE that can be projected onto P without overlap).

7.3.2 Navigation Mesh

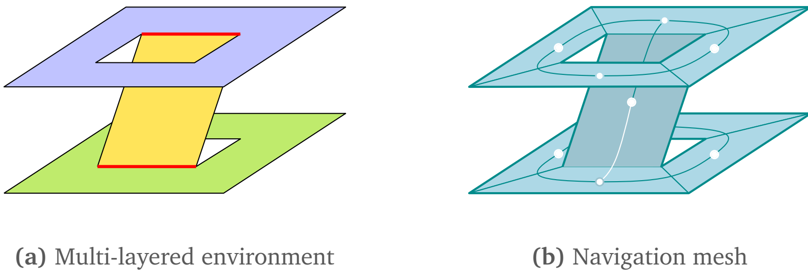
Now that we have a definition of the free space $\mathcal{E}_{free}$, we can define a navigation mesh as a tuple $\mathcal{M} = (\mathcal{R}, \mathcal{G})$:

- $\mathcal{R} = \{R_0, R_1, \ldots\}$ is a collection of geometric *regions* in $\mathbb{R}^3$ that represents $\mathcal{E}_{free}$. Each region R_i is *P-simple*, by which we mean that a region cannot intersect itself when projected onto the ground plane P.

- $\mathcal{G} = (V, E)$ is an undirected *graph* that describes how characters can navigate between the regions in $\mathcal{R}$.

Figure 7.2b shows an abstract example of a navigation mesh.

For many navigation meshes, $\mathcal{R}$ consists of non-overlapping simple polygons, and $\mathcal{G}$ is the dual graph of $\mathcal{R}$, with one vertex per region and one edge per pair of adjacent region sides. However, other possibilities exist. In the Clearance Disk



(a) Multi-layered environment (b) Navigation mesh

Figure 7.2: (a) A simple multi-layered environment. This example was also used in Chapters 2 and 5. **(b)** A navigation mesh is a description of the environment for path planning purposes. It consists of walkable regions and a graph that describes their connectivity.

Graph [121], $\mathcal{R}$ consists of overlapping disks, and $\mathcal{G}$ contains an edge wherever two disks overlap. The Explicit Corridor Map [153] is explicitly defined as a graph, and the mesh regions can be derived from its annotations. Still, all meshes have in common that $\mathcal{R}$ and $\mathcal{G}$ can be obtained from their representation in some way.

In Chapters 2, 3 and 5, we have explained that it is common for a navigation mesh to ignore height differences along a path during planning because all individual surfaces are sufficiently flat. More specifically, the length of an edge in $\mathcal{G}$ is often computed using a projection onto the ground plane P. Therefore, in this chapter, we will *not* judge a navigation mesh by its precise preservation of height differences. We will only assume that the height coordinates in $\mathcal{R}$ are *sufficiently close* to the height coordinates in $\mathcal{E}_{free}$ that they represent, to such a degree that the regions of $\mathcal{R}$ can unambiguously be mapped back onto $\mathcal{E}_{free}$ if desired.

7.4 Properties of Navigation Meshes

In this section, we propose a set of properties that describe a navigation mesh's data structure, algorithms, and limitations. These properties do not depend on a specific implementation or environment. They can serve as a 'checklist' to simplify choosing an appropriate mesh for a particular application. In Section 7.5, we will use these properties to compare various navigation meshes.

Region type The type of regions in $\mathcal{R}$, e.g. triangles or disks.

Graph type A description of the path planning graph $\mathcal{G}$, e.g. 'the dual graph of $\mathcal{R}$' or 'the medial axis of $\mathcal{E}_{free}$'.

Overlap Whether or not the regions in $\mathcal{R}$ can overlap by definition. Having overlapping regions is generally discouraged because geometric algorithms that assume non-overlapping regions may not work properly. Also, a query point (or an agent) can be in multiple regions at the same time in case of overlap, which may complicate path planning and crowd simulation.

Pipeline The environment processing pipeline performed by the construction algorithm, e.g. 'from a 2D environment to a navigation mesh' or 'from a 3DE via an MLE to a navigation mesh'. We also indicate whether this pipeline is voxel-based or exact.

Parameters The parameters that the user needs to set for the construction algorithm of the navigation mesh. Having fewer parameters implies a more automated process for computing the mesh. These parameters are often related to the filtering process that extracts the walkable surfaces from the 3D geometry.

Computational complexity The asymptotic construction time of the navigation mesh. This is usually expressed in terms of the environment complexity or a grid resolution.

Storage complexity The asymptotic size of the navigation mesh data structure.

Clearance Whether or not the navigation mesh supports the computation of paths with an arbitrary clearance from obstacles, i.e. paths for disks with an arbitrary radius.

Dynamic updates Whether or not the navigation mesh supports dynamic insertions and deletions of obstacles.

7.5 Theoretical Comparison

We now describe and compare the state-of-the art navigation meshes that will also be included in our experiments in Section 7.7. The first two navigation meshes are exact; the others are voxel-based and cover the full 3D pipeline. Although more navigation meshes exist, we currently include only the navigation meshes that are designed specifically for the environments described in Section 7.3, and for which we could obtain robust source code from their respective authors.

Table 7.1 summarizes each navigation mesh based on the properties from Section 7.4. An example of the output for each method is shown in Figure 7.1.

7.5.1 Local Clearance Triangulation

The Local Clearance Triangulation (LCT) by Kallmann [67] subdivides a 2D environment of complexity n into $\mathcal{O}(n)$ triangles with all vertices on the boundary of $\mathcal{E}_{free}$. These triangles are the regions of $\mathcal{R}$, and $\mathcal{G}$ is their dual graph. Each triangle edge is annotated with *clearance* values that denote the smallest distance to obstacles when a character would cross this edge. Therefore, during path planning, the LCT can determine in constant time whether a particular move is collision-free for a character with a particular radius.

The triangles of the LCT need to adhere to certain local constraints in order for their clearance annotations to work. The LCT is constructed by first computing a constrained Delaunay triangulation in $\mathcal{O}(n \log n)$ time, and then applying $\mathcal{O}(n)$ refinements until all constraints are satisfied. This sequence of refinements may take $\mathcal{O}(n^2)$ time in the worst case. According to Kallmann (by personal communication in July 2016), the tested implementation runs in $\mathcal{O}(n\sqrt{n})$ expected time by using a special point-location method. The LCT also supports dynamic updates. An extension to MLEs has not been developed, but an approach similar to the Explicit Corridor Map (Chapter 5) should be possible.

The LCT takes a set of 2D line segment constraints as input, so our benchmark program first needs to compute the boundary segments of an environment. The output of the LCT program is a set of constrained and unconstrained segments. We need to convert this to a set of triangles such that the triangles inside the obstacle space are filtered out. These pre-processing and post-processing steps will not be included in our time measurements.

7.5.2 Explicit Corridor Map

The ECM from Chapters 4 and 5 is an exact navigation mesh. Its graph $\mathcal{G} = (V, E)$ is the medial axis of $\mathcal{E}_{\mathit{free}}$, where V contains the medial axis vertices of degree 1, 3, or higher. Each edge in E is a sequence of medial axis arcs between two vertices of V. An edge consists of its two endpoints and a sequence of bending points at which the medial axis changes shape. These points are all annotated with their nearest obstacle point on the left and right side of the medial axis. These annotations induce a subdivision of $\mathcal{E}_{\mathit{free}}$ into polygonal regions. Each region in $\mathcal{R}$ is a (possibly non-convex) polygon of at most 6 distinct vertices.

Because the distance to the nearest obstacle is known at each bending point, the ECM can be used to plan paths for characters of an arbitrary radius. Furthermore, the nearest obstacle point for any query point q can be found in constant time when the region that contains q is known, which is not true for arbitrarily chosen regions. The ECM also supports dynamic updates.

For a 2D environment of complexity n, the ECM has size $\mathcal{O}(n)$ and is computed in $\mathcal{O}(n \log n)$ time. For an MLE with k connections, the *medial axis* has size $\mathcal{O}(n)$ and can be computed in $\mathcal{O}(n \log n \log k)$ time by iteratively opening the connections. Our ECM construction software described in Chapters 4 and 5 is fast and robust. It runs in $\mathcal{O}(kn \log n)$ time, and it splits ECM edges whenever they intersect a connection, which yields a size of $\mathcal{O}(kn)$. In exchange for its advantages, the ECM is conceptually slightly more difficult than e.g. a triangulation; it may be a less intuitive choice at first sight.

7.5.3 Clearance Disk Graph

Pettré et al. [121] presented the first *voxel-based* navigation mesh for 3D environments. They introduced many new concepts that have evolved in later algorithms.

This navigation mesh does not have an official name; in this chapter, we refer to it as the Clearance Disk Graph (CDG).

The CDG uses voxelization to approximate the areas where characters can stand. Next, the voxels are extracted for which the clearance (the distance to the nearest obstacle) is locally largest. These form an approximation of the medial axis of $\mathcal{E}_{free}$, and each cell has an obstacle-free disk associated to it. A subset of these cells is chosen such that their disks overlap but do not contain each other's center points. The resulting disks are the regions of $\mathcal{R}$, and the graph $\mathcal{G}$ has a vertex for each disk and an edge for each pair of overlapping disks. A disadvantage of the CDG is that its disks can never fully cover the free space. Extra disks (that do not lie on the medial axis) can be added to improve coverage.

The asymptotic construction time of the CDG is difficult to assess because the algorithm relies on rendering techniques. Also, the number of disks cannot be expressed in terms of the environment complexity, but it is at least limited by the number of voxels S.

7.5.4 Recast

The Recast Navigation toolkit by Mononen [102] also uses voxelization to approximate $\mathcal{E}_{free}$. However, unlike the CDG, Recast converts the walkable voxels to non-overlapping convex polygonal regions. The method offers many detailed settings for this conversion, e.g. for tracing obstacle boundaries, grouping adjacent polygons, and discarding regions that are too small. This large number of parameters complicates a comparative study because each environment may have its own optimal settings. Another parameter is the character radius, which is subtracted from the navigation mesh during its construction. As mentioned, we will use a radius of zero to allow a fair comparison to other methods.

Recast computes two versions of the navigation mesh: a coarse mesh that is used for path planning, and a detailed mesh that captures height differences more accurately. In our experiments, we will use the coarse mesh to determine the regions $\mathcal{R}$ and the graph $\mathcal{G}$, and the detailed mesh to measure how well the result covers $\mathcal{E}_{free}$.

Recast does not provide theoretical guarantees of running times, accuracy, or storage size. On the other hand, the source code of Recast is considered to be very mature: it is fast, robust, and a popular choice for game development. A variant of Recast is included in the popular Unity3D game engine [158].

7.5.5 NEOGEN

The NEOGEN method by Oliva and Pelechano [114] also starts with voxelization, but it then groups the walkable voxels into 2D layers, which yields an approximation of an MLE. For each layer, NEOGEN uses GPU techniques to obtain a more precise floorplan in a way that does not depend on the voxel size. Compared to

Recast, the overall precision of NEOGEN is therefore less dependent on the grid resolution.

Based on these floorplans, an exact 2D algorithm (ANavMG) [112] is used to compute the navigation mesh for each layer. This 2D algorithm subdivides the free space of a layer into convex polygons. For each convex obstacle corner, the algorithm draws line segments to other obstacles within an angular range. This algorithm runs in $\mathcal{O}(nr)$ time, where n is the total number of obstacle vertices, and $r \in \mathcal{O}(n)$ is the number of convex obstacle corners. Finally, the navigation meshes of each layer are merged into a single data structure. In our experiments, for simplicity, we will use the 'full' voxel-based method in both MLEs and 2D environments.

A contribution of NEOGEN is the *convexity relaxation* parameter that can be used to allow slightly non-convex regions. This decreases the total number of regions in exchange for having more complex region shapes. NEOGEN does not use a predefined character radius, and its regions do not encode clearance information by default. However, Oliva and Pelechano have explained how clearance information can be added to the navigation mesh if desired [113].

7.5.6 Grid

As a baseline for our comparison, we have implemented a simple grid method. It voxelizes the environment similarly to Recast and NEOGEN, but it uses the walkable voxels directly as navigation mesh regions. Therefore, each region in $\mathcal{R}$ is a square.

We include this method because grids are still frequently used for path planning. They are easy to implement and a common choice for games that are grid-based by design [139]. Another advantage is that algorithms such as A* search can be optimized for grids [32, 44, 95, 140]. Many variants of A* are either designed with grids in mind [15] or explained in terms of grids [85, 86, 97]. However, grids are typically more dense than other navigation meshes (e.g. they require many cells to represent large open spaces), which makes them less suitable for planning many paths in real-time. Also, a high grid resolution is required if the environment contains many details.

7.5.7 Comparison

The distinction between exact and voxel-based navigation meshes is clear. If an application features 3D geometry that has not yet been pre-processed into planar layers, then an exact navigation mesh is currently not sufficient. On the other hand, scalability and precision are clear advantages of exact methods. We expect that voxel-based approaches cannot achieve perfect coverage when an environment contains many details.

The LCT and ECM seem similar: they are exact, they do not require any parameters, they encode clearance information, and their size is $\mathcal{O}(n)$. A difference

is that the ECM currently supports MLEs as well. Also, the ECM construction algorithm has a better asymptotic worst-case construction time, but this difference might only be noticeable at a very high complexity. Therefore, we expect that the two algorithms will perform similarly in practice.

NEOGEN and Recast use comparable voxel-based techniques and they both yield convex polygonal regions. Recast has more parameters: it aims at a semi-automatic construction process in which the user tweaks parameters to achieve the desired result. The main advantages of NEOGEN are its techniques to obtain a higher precision per layer, the use of an exact 2D algorithm, and the convexity relaxation parameter. An advantage of Recast is the maturity of its code base.

The CDG has similar parameters to NEOGEN, but its representation with overlapping disks is different. In polygonal environments, disks cannot cover the free space completely. However, the advantage of disks is that they trivially encode clearance information. Another aspect to take into account is that the CDG source code is not optimized for e.g. gaming applications. We expect that the other methods will be more efficient in their current state.

Finally, we expect that our naive grid implementation yields the largest graphs, and that it does not cover $\mathcal{E}_{free}$ well if the obstacles are not aligned with the grid cells. Other voxel-based methods use post-processing steps to convert voxels to smooth regions, which improves coverage and simplifies the graph.

7.6 Quality Metrics for Navigation Meshes

For a navigation mesh $\mathcal{M} = (\mathcal{R}, \mathcal{G})$ that has been constructed for an environment using a particular implementation, we want to objectively measure the quality. Many possible evaluation criteria exist, and each application area may have its own view of what is good or desirable. In this chapter, we choose to focus on the navigation mesh itself and on the performance of its construction algorithm. We will present metrics that answer the following questions:

1. (Coverage) How accurately do the regions of $\mathcal{R}$ cover the geometry of $\mathcal{E}_{free}$? If parts of the free space are not covered, characters might not find a path in $\mathcal{G}$ even though a path exists in $\mathcal{E}_{free}$. If parts outside the free space are covered, characters might accidentally find paths through obstacles.

2. (Connectivity) How accurately does the graph $\mathcal{G}$ represent the connectivity of $\mathcal{E}_{free}$? This question is related to coverage because it determines whether or not appropriate paths can be found; however, it concerns topology rather than geometry.

3. (Complexity) How efficiently does $\mathcal{M}$ represent $\mathcal{E}_{free}$, i.e. how 'compact' is the mesh? This can refer to the size of the graph $\mathcal{G}$ (a smaller graph allows faster path planning queries) or to the complexity of each individual region in $\mathcal{R}$ (simpler regions allow faster basic operations such as point location). It depends on the application which property is more desirable.

Navigation mesh	Region type	Graph type	Overlap	Pipeline	Parameters	Computational complexity	Storage complexity	Dynamic updates	Arbitrary clearance
LCT	Triangles	Dual of $\mathcal{R}$	No	$2D \rightarrow \mathcal{M}$ (exact)	None	2D: $\mathcal{O}(n^2)$ (expected $\mathcal{O}(n\sqrt{n})$)	$\mathcal{O}(n)$	+	+
ECM	Polygons	Medial axis	No	2D/MLE $\rightarrow \mathcal{M}$ (exact)	None	2D: $\mathcal{O}(n \log n)$ MLE: $\mathcal{O}(kn \log n)$	$\mathcal{O}(n)$ $\mathcal{O}(kn)$	+	+
CDG	Disks	Dual of $\mathcal{R}$	Yes	$3D \rightarrow \mathcal{M}$ (voxel-based)	3D filtering Voxel precision Min character radius Min/max disk size	?	$\mathcal{O}(S)$	+/-	+
Recast	Convex polygons	Dual of $\mathcal{R}$	No	$3D \rightarrow \mathcal{M}$ (voxel-based)	3D filtering Voxel precision Region refinement Character radius	?	?	+/-	-
NEOGEN	(Convex) polygons	Dual of $\mathcal{R}$	No	$3D \rightarrow$ MLE $\rightarrow \mathcal{M}$ (voxel-based + exact)	3D filtering Voxel precision Convexity relaxation	2D: $\mathcal{O}(n^2)$ MLE: $\mathcal{O}(n^2)$ 3D: ?	$\mathcal{O}(n)$ $\mathcal{O}(n)$	+/-	+/-
Grid	Squares	Dual of $\mathcal{R}$	No	$3D \rightarrow \mathcal{M}$ (voxel-based)	3D filtering Voxel precision	?	$\mathcal{O}(S)$	+/-	-

Table 7.1: Overview of the navigation meshes compared in this chapter. For the rightmost columns, '+' means that a property is supported in the current implementation, '+/-' means that a property is currently not supported but can be added, and '-' means that a property is not supported by definition.

4. (Performance) How efficiently is $\mathcal{M}$ computed in terms of time and memory? An efficient algorithm allows the construction of navigation meshes in interactive applications such as level editors. Even if the navigation mesh is precomputed in an off-line stage, performance is still desirable.

Of course, many other questions are interesting, e.g. questions related to the peformance of path planning queries, or to the quality or realism of paths. We will discuss a number of options in Section 7.8 as suggestions for future work.

Analyzing coverage and connectivity is only useful for voxel-based navigation meshes that attempt to 'discover' $\mathcal{E}_{free}$ themselves; exact methods are known to yield perfect coverage. Also, some properties can only be analyzed if the ground truth (the structure of $\mathcal{E}_{free}$) is known. Therefore, each input environment in our experiments will be a 'clean' walkable environment, i.e. a manifold that contains only walkable polygons. While this implies that voxel-based methods will not fully use their advantage of handling raw (non-clean) 3D geometry, we believe that using the same input for all methods yields a more objective comparison.

Since the outcome of each metric depends on implementation details, the results should always be judged in combination with the theoretical properties of Section 7.4.

7.6.1 Coverage

The first set of metrics describes how well the free space is covered. Coverage is a complicated property to evaluate due to the 3D structure of $\mathcal{R}$ and $\mathcal{E}_{free}$. We need to introduce a number of concepts before we can define actual metrics. These concepts are based on the assumption that the environment has a consistent direction of gravity. Coverage is the only category of metrics in which this assumption comes into play.

Mapping the Navigation Mesh onto the Free Space

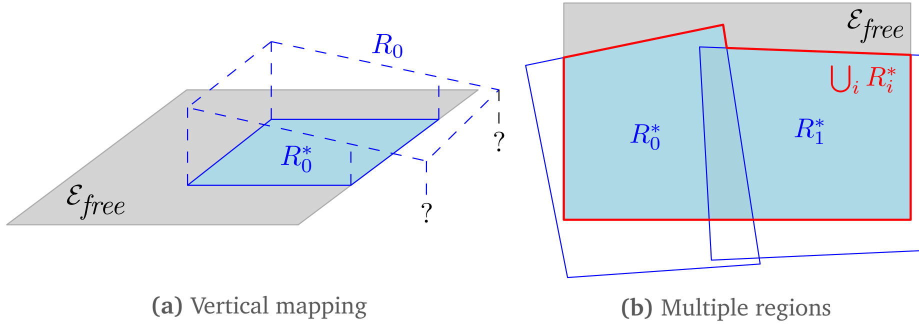
Comparing the geometry of $\mathcal{R}$ to the geometry of $\mathcal{E}_{free}$ requires us to vertically map these two structures onto each other. This is straight-forward if the environment consists of a single layer because everything can then be projected onto the ground plane P.

However, for general WEs in $\mathbb{R}^3$, mapping $\mathcal{R}$ onto $\mathcal{E}_{free}$ is ambiguous. In an abstract sense, there should be a function m such that, for any point p in a navigation mesh, $m(p)$ *vertically* maps p to an appropriate point in $\mathcal{E}_{free}$ if possible (and if not, p is assumed to lie in $\mathcal{E}_{obs}$). Several choices can be made here, such as the maximum allowed height difference between p and $m(p)$. We will describe our own implementation of m in Section 7.7.

Using the function m, we define a *mapped region* R_i^* as a region R_i that has been mapped onto $\mathcal{E}_{free}$ wherever possible, i.e. $R_i^* = \{m(p) \mid p \in R_i \text{ and } m(p) \text{ exists}\}$. Figure 7.3a shows an example of a region that can only partly be mapped onto $\mathcal{E}_{free}$.

Because each mapped region is a subset of $\mathcal{E}_{free}$, we can use the mapped regions to define unions, coverage, and overlap. An example is illustrated in Figure 7.3b.

Let the *mapped region set* $\mathcal{R}^*$ be a version of $\mathcal{R}$ in which all regions have been mapped onto $\mathcal{E}_{free}$, i.e. $\mathcal{R}^* = \{R_i^* \mid R_i \in \mathcal{R}\}$. The regions in $\mathcal{R}^*$ may overlap: in that case, some points of $\mathcal{E}_{free}$ are represented more than once.



(a) Vertical mapping

(b) Multiple regions

Figure 7.3: Mapping and coverage. **(a)** 3D view of a navigation mesh region R_0. Only a part of R_0 can be vertically mapped onto $\mathcal{E}_{free}$. The mapped region R_0^* is highlighted in blue. **(b)** Top-view of a different example with two regions R_0 and R_1. The mapped regions R_0^* and R_1^* highlighted in blue, partly overlap. The union of all mapped regions, $\bigcup_i R_i^*$, is well-defined because the mapped regions are subsets of $\mathcal{E}_{free}$.

Computing the Projected Area

Because we ignore height differences in our problem domain, our coverage metrics are based on projected areas onto the ground plane P. We define the *projected area* of a shape S as follows:

- If S does not overlap itself when projected onto P (i.e. if S is a P-simple shape as defined in Section 7.3.2), the projected area $||S||$ is the signed area of the projection of S onto P.
- Otherwise, let $\{S_0, \ldots, S_{s-1}\}$ be any subdivision of S into P-simple shapes. The projected area of S is the sum of projected areas of these components, i.e. $||S|| = \sum_i ||S_i||$.

We assume that $\mathcal{E}_{free}$ is given as a subdivision into P-simple shapes, such that $||\mathcal{E}_{free}||$ can be computed.

Coverage Metrics

We introduce three coverage metrics. Each metric has a regular version M with range $\mathbb{R}_{\geq 0}$ and a normalized version M' with range $[0, 1]$, as described below.

Free space covered The area of $\mathcal{E}_{free}$ that is correctly covered by at least one navigation mesh region. Because the regions in $\mathcal{R}^*$ may overlap and we do

not want to count overlapping regions twice, we first compute the union of $\mathcal{R}^*$ in $\mathbb{R}^3$. High coverage is desirable: it allows characters to use more of $\mathcal{E}_{free}$.

$$Cov = ||\bigcup_i R_i^*|| \text{ and } Cov' = \frac{Cov}{||\mathcal{E}_{free}||}$$

Incorrect area The area of the mesh that could not be mapped to $\mathcal{E}_{free}$, i.e. the area of the mesh that 'overshoots' $\mathcal{E}_{free}$ and lies in the obstacle space. Intuitively, this is the difference between $\mathcal{R}$ and the part of $\mathcal{R}$ that can be mapped onto $\mathcal{E}_{free}$. Ideally, the incorrect area should be zero because areas outside $\mathcal{E}_{free}$ should not be accessible to characters.

$$A_{inc} = \sum_i (||R_i|| - ||R_i^*||) \text{ and } A'_{inc} = \frac{A_{inc}}{\sum_i ||R_i||}$$

Note: while it may seem more intuitive to express this metric as 'the area of the obstacle space $\mathcal{E}_{obs}$ that is covered', this would be impossible because (for WEs in 3D) $\mathcal{E}_{obs}$ does not have an area.

Overlap The amount of overlap among the regions in the navigation mesh. Intuitively, overlap is the sum of all region areas minus the area that is covered at least once. Because coverage is only defined properly inside $\mathcal{E}_{free}$, overlap is also based on the mapped region set $\mathcal{R}^*$. The normalized version indicates which fraction of $\mathcal{R}^*$ is redundant.

$$Ov = \sum_i ||R_i^*|| - ||\bigcup_i R_i^*|| \text{ and } Ov' = \frac{Ov}{\sum_i ||R_i^*||}$$

If a navigation mesh is deliberately based on overlapping regions (e.g. [121]), then this metric simply indicates how much space is covered more than once. Otherwise, overlap may indicate an implementation bug, which is not the focus of our comparative study.

7.6.2 Connectivity

The second set of metrics analyzes how well the graph $\mathcal{G} = (V, E)$ represents the dual graph of $\mathcal{E}_{free}$.

 # Connected components The number of connected components in $\mathcal{G}$. Ideally, this value is equal to the number of connected components in $\mathcal{E}_{free}$. Having more components implies that not all adjacencies in $\mathcal{E}_{free}$ are captured. Having fewer components implies that regions have been made adjacent when there are actually obstacles in-between.

 # Boundaries The number of environment boundaries perceived by the navigation mesh. Ideally, this value is equal to the actual number of boundaries

of $\mathcal{E}_{free}$. It can be computed by traversing the graph $\mathcal{G}$, checking the corresponding regions in $\mathcal{R}$, and collecting the region edges that are not shared by multiple regions. The number of boundaries is the number of closed loops that are traced. Note: if the number of boundaries is perfect, the geometry of $\mathcal{R}$ is not necessarily correct.

7.6.3 Complexity

The third set of metrics measures how efficiently the navigation mesh represents $\mathcal{E}_{free}$. The size of $\mathcal{G}$, the number of regions, and the complexity of these regions may have implications for the efficiency of path planning and crowd simulation.

 # Vertices, # Edges The number of vertices and the number of edges in $\mathcal{G}$, i.e. $|V|$ and $|E|$. A larger graph implies that path planning queries (and other algorithms that browse the graph) typically take more time to answer. Therefore, lower numbers imply faster path planning.

 # Regions The number of regions in the navigation mesh: $|\mathcal{R}|$. This indicates how efficiently the free space is represented by elementary parts. It also suggests how often a character in the simulation may move from one region to another. Moving to another region typically triggers computational overhead in the simulation. Hence, having fewer regions may cause some aspects of the simulation to run more efficiently. Note that $|\mathcal{R}| = |V|$ if $\mathcal{G}$ is simply the dual graph of $\mathcal{R}$.

Region complexity The number of floating-point numbers required to describe the regions in $\mathcal{R}$. Since we treat regions as shapes in $\mathbb{R}^3$, we will say that a polygonal region with p vertices has complexity $3p$. A disk has a complexity of 4 because it can be defined by a center point in $\mathbb{R}^3$ and a radius. Naturally, other choices are possible as well. Some navigation meshes have extra annotations, such as the maximum allowed radius of a character for an edge traversal [67]. We will not include such annotations in this metric.

We measure three variants: the average complexity among all regions, the standard deviation, and the total complexity of all regions combined. A low region complexity implies that geometric operations within these regions are computationally cheap. If a mesh has a small number of regions, a low region complexity, *and* high coverage, then it is a very efficient description of $\mathcal{E}_{free}$.

7.6.4 Performance

The final set of metrics concerns the practical performance of the navigation mesh implementations. One issue to take into account is that voxel-based methods perform more steps than exact methods. Another issue is that different implementations are in drastically different states: some are a 'proof of concept' for research

purposes, while others are highly optimized for the gaming and simulation industry. Still, these metrics can indicate if an implementation corresponds to the asymptotic complexity of a navigation mesh, and how well a navigation mesh scales to large or complex environments.

Construction time The time (in milliseconds) spent on computing the navigation mesh. Naturally, fast construction is encouraged because it makes the algorithm suitable for interactive applications.

Memory usage The maximum amount of memory (in MB) used during the execution of the program. A small value implies that the mesh can be computed in situations with limited resources, e.g. on a game console with little working memory.

To obtain more reliable results, we will run each navigation mesh program 10 times and report the average values and standard deviations. This is not needed for the other categories of metrics because the output of each program is deterministic.

7.6.5 Summary

Table 7.2 summarizes all metrics described in this section. The metrics for coverage and connectivity are easy to interpret because we know their optimal values. The metrics for performance are also intuitive: the more efficiently a mesh is constructed, the better. The complexity metrics are more difficult to judge because not all values can be minimized at the same time: for example, a small set of regions will typically imply that the regions themselves are complex.

Finally, we acknowledge that different applications may assign different priorities to each metric. For instance, in games where the mesh needs to be computed at run-time, it is likely that efficiency and real-time performance are preferred over exact coverage. By providing all metrics, we leave their interpretation to the developers of the application at hand.

7.7 Experimental Comparison

In this section, we use our metrics to experimentally compare various navigation meshes in a range of environments.

7.7.1 Implementation

We have converted each navigation mesh program to a stand-alone executable that reads an input file, computes a navigation mesh, and returns the result. We have written a benchmark tool that communicates with these programs, converts environments between different file formats, and calculates all metrics. Also, the CDG requires the walkable surfaces to be visible from all sides; to ensure this, we extrude all input polygons downwards by a small amount.

Category	Metric	Range	Preferred value
Coverage	Free space covered	$\mathbb{R}_{\geq 0}$	Ground truth
		Normalized: $[0, 1]$	1
	Incorrect area	$\mathbb{R}_{\geq 0}$	0
		Normalized: $[0, 1]$	0
	Overlap	$\mathbb{R}_{\geq 0}$	0
		Normalized: $[0, 1]$	0
Connectivity	# Connected components	$\mathbb{N}$	Ground truth
	# Boundaries	$\mathbb{N}$	Ground truth
Complexity	# Graph vertices	$\mathbb{N}$	As small as possible
	# Graph edges	$\mathbb{N}$	As small as possible
	# Regions	$\mathbb{N}$	As small as possible
	Region complexity	Total: $\mathbb{N}$	As small as possible
		Avg/SD: $\mathbb{R}_{\geq 0}$	
Performance	Construction time (ms)	Avg/SD: $\mathbb{R}_{\geq 0}$	As small as possible
	Memory usage (MB)	Avg/SD: $\mathbb{R}_{\geq 0}$	As small as possible

Table 7.2: Summary of the navigation mesh quality metrics described in Section 7.6.

An important detail is our choice of the mapping function m that is used to compute coverage. For a point p in the navigation mesh, we define $m(p)$ as the *nearest* point in $\mathcal{E}_{free}$ above or below p up to a threshold distance T. The threshold distance is required to prevent erroneous points of $\mathcal{R}$ from getting mapped onto surfaces that are too far away. We choose a value of $T = 1$ meter because the vertical clearance is at least 2 meters in all our test environments. Admittedly, this choice for m requires that the height coordinates of the navigation mesh are sufficiently close to the ground truth. It may fail in environments with gradual yet large height differences that are not captured by the navigation mesh. (In 2D environments, T can be ignored because a vertical mapping is already unambiguous.)

We have implemented our coverage metrics using a CGAL-based program [17] that can compute the intersection of two OBJ files based on the threshold distance T. For this program, we approximated the disks of the CDG by polygons of 16 vertices. We used *inner* approximations: the approximated disks were smaller than the actual disks. This leads to slightly lower numbers for coverage, incorrect area, and overlap, but the chosen precision is sufficient for comparative purposes.

7.7.2 Parameter Settings

Most navigation meshes are built based on various parameters. For simplicity, we use one set of parameter settings for all experiments.

Precision. For the CDG, Recast, and NEOGEN, we used voxels of 0.2 meters in all three dimensions. This is the smallest voxel size at which we could obtain results in most environments. For the CDG, we enforced a maximum resolution

of 512 pixels in all dimensions. This kept the construction times manageable for environments that were too large to allow a precision of 0.2 m.

For the grid baseline method, we reduced the voxel size to 1×1 m in the horizontal plane to prevent the program from taking too much time and memory. We could keep the voxel height at 0.2 m.

For the CDG, we used a minimum disk radius of 0.1 m (to prevent the method from generating many small disks) and a maximum disk radius of $100{,}000$ m (which is essentially infinite in our examples).

Filtering. We have created our input environments such that they are entirely walkable and no more surfaces need to be filtered out. Therefore, for all voxel-based methods, we used a maximum surface slope of 60 degrees, a character radius of 0, and a character height of 0.5 m. These settings ensured that all environments could (in theory) be covered completely.

Recast offers various other parameters that need to be tweaked for each environment to get the best results. Through preliminary experiments, we obtained the following values that gave good results in most environments: Tiling: Off; Max climb: 0.5; Min region size: 0; Merged region size: 10,000; Partitioning: Watershed; Max edge length: 500; Max edge error: 1.3; Vertices per polygon: 6; Detail mesh sample distance: 6; Detail mesh max error: 1.

Other. To compute ECMs, we used the implementation based on the Boost Voronoi library [14]. It computes a 2D Voronoi diagram in $\mathcal{O}(n \log n)$ worst-case time, and it can process multiple layers of an MLE at the same time by using parallel threads.

For NEOGEN, we used a convexity relaxation parameter of 0. Increasing this parameter leads to fewer regions (i.e. a smaller dual graph) and a higher region complexity. Thus, the 'best' value for this parameter depends on the application.

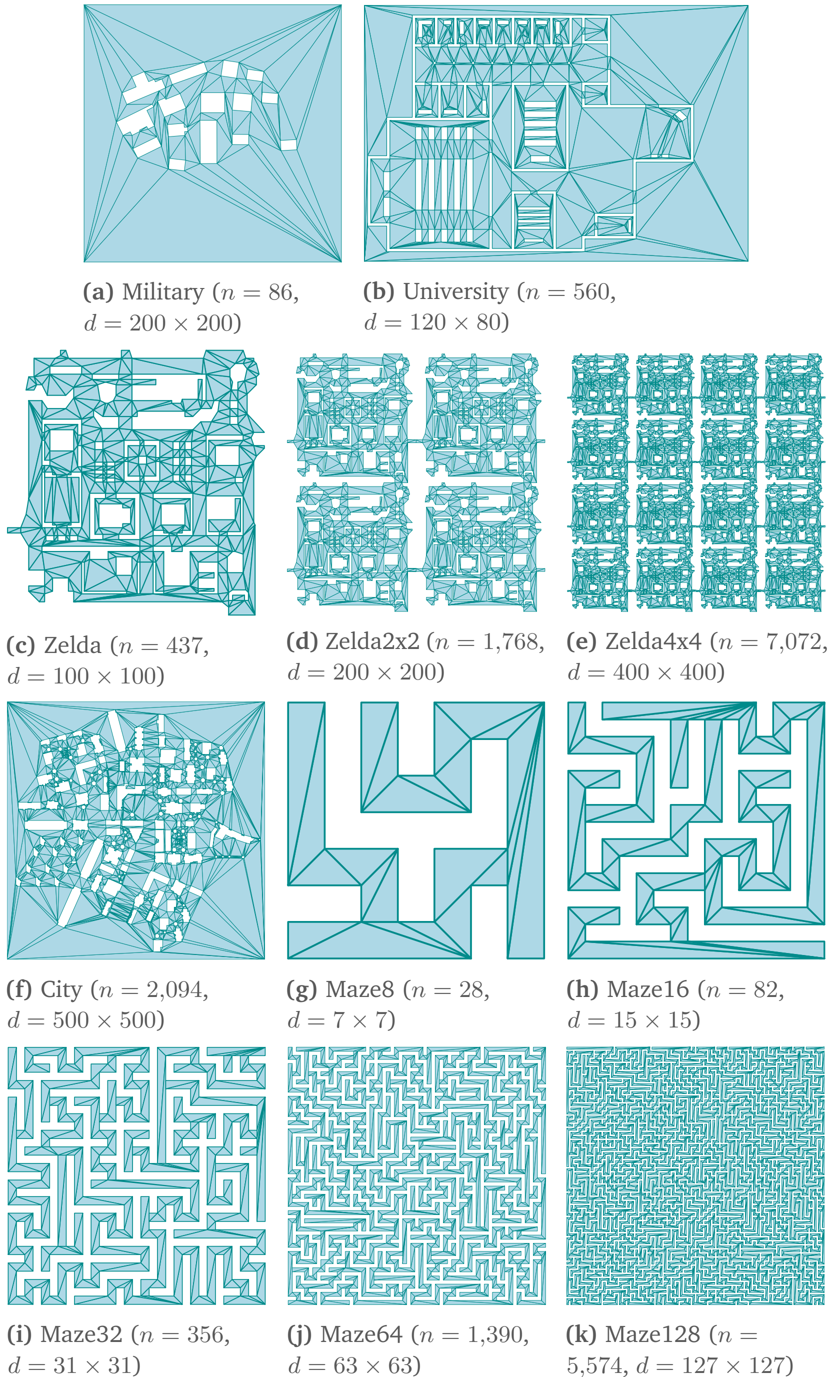
7.7.3 Environments

We have computed navigation meshes for the 2D and 3D input environments shown in Figures 7.4 and 7.5. We have converted each environment to a clean representation of $\mathcal{E}_{free}$, subdivided into layers whenever necessary. Most of the 2D environments have also appeared in Chapter 4. To test for scalability, we have also added randomly generated 2D mazes of various sizes, inspired by Sturtevant [139].

The environments in our test set vary in scale and complexity, but it is not yet a complete set of environments that can expose the strengths and weaknesses of each navigation mesh. In future work, we will propose a more comprehensive set of benchmark environments.

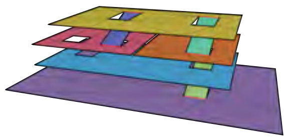
7.7.4 Results

As visual examples, the navigation meshes produced by each method are shown for three of our environments: *Military* (Figure 7.1), *University* (Figure 7.6), and

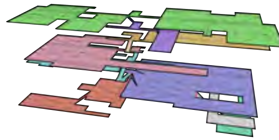


(a) Military ($n = 86$, $d = 200 \times 200$)
(b) University ($n = 560$, $d = 120 \times 80$)
(c) Zelda ($n = 437$, $d = 100 \times 100$)
(d) Zelda2x2 ($n = 1{,}768$, $d = 200 \times 200$)
(e) Zelda4x4 ($n = 7{,}072$, $d = 400 \times 400$)
(f) City ($n = 2{,}094$, $d = 500 \times 500$)
(g) Maze8 ($n = 28$, $d = 7 \times 7$)
(h) Maze16 ($n = 82$, $d = 15 \times 15$)
(i) Maze32 ($n = 356$, $d = 31 \times 31$)
(j) Maze64 ($n = 1{,}390$, $d = 63 \times 63$)
(k) Maze128 ($n = 5{,}574$, $d = 127 \times 127$)

Figure 7.4: Top views of the 2D environments used in our experiments. The number of polygon vertices n and the physical dimensions d (in meters) are shown in brackets.



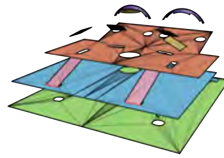
(a) ParkingLot ($n = 120$, $d = 36 \times 21 \times 9$)



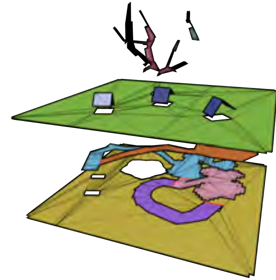
(b) Library ($n = 489$, $d = 59.5 \times 23.9 \times 21.1$)

(c) Oilrig ($n = 1{,}644$, $d = 273.6 \times 225.6 \times 84.4$)

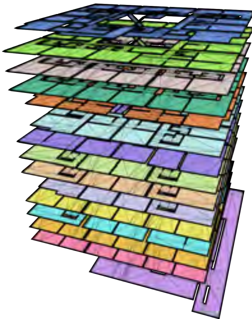
(d) Neogen1 ($n = 5{,}473$, $d = 63.5 \times 63.5 \times 15.7$)



(e) Neogen2 ($n = 1{,}089$, $d = 63.5 \times 63.5 \times 31.1$)



(f) Neogen3 ($n = 672$, $d = 63.5 \times 63.5 \times 57.4$)



(g) Tower ($n = 9{,}677$, $d = 34.8 \times 37.3 \times 39.2$)

(h) BigCity ($n = 61{,}785$, $d = 500 \times 500 \times 39.2$)

Figure 7.5: Renderings of the multi-layered environments used in our experiments. Each layer of an environment is shown in a different color. Connections between layers are shown in red. The number of polygon vertices n and the physical dimensions d (width $\times$ depth $\times$ height, in meters) are shown in brackets.

Library (Figure 7.7). The tables at the end of this chapter contain the quantitative results for all environments. We have created separate tables for each category of metrics: coverage (Tables 7.3 and 7.4), connectivity (Tables 7.5 and 7.6), complexity (Tables 7.7 and 7.8), and performance (Tables 7.9 and 7.10). We will now highlight the most important observations from these results.

Coverage. We have chosen our parameters to maximize coverage. Still, in terms of absolute values, the voxel-based methods sometimes missed large areas or covered large incorrect parts, up to hundreds of square meters in large enviroments. However, the *relative* coverage was still high (typically over 90%).

The maze environments are an exception: even though their free space was perfectly aligned with grid cells of 1×1m, the CDG and Recast could not capture them accurately. However, these mazes are quite detailed relative to their size, so a finer grid resolution could improve the results. NEOGEN generally yielded better coverage due to its extra processing step per layer. It would be interesting to let methods automatically choose an appropriate resolution based on a user-specified balance between coverage and performance. A theoretically stronger alternative would be to reconstruct $\mathcal{E}_{free}$ without relying on a grid resolution.

For completeness, we have also included the results for the *exact* methods. As expected, these methods usually scored perfectly in terms of coverage, with a few minor exceptions due to small measurement errors.

Connectivity. Recast and NEOGEN captured connectivity quite well for most environments, except in the mazes where each accidental gap causes parts to become disconnected. For the CDG, the graph usually contained many connected components, and gaps in the covered space led to a large number of boundaries. The grid also contained unexpected gaps at times, due to small errors in the current implementation. Still, the grid method works sufficiently well for the purpose of a comparison.

Again, we have also included the results for the LCT and the ECM, for completeness. These methods usually yielded perfect connectivity values. There were a few exceptions for the LCT, most likely caused by our own process of converting $\mathcal{E}_{free}$ to a boundary representation.

It is motivating to see that bad connectivity values corresponded to navigation meshes that were also *visually* incorrect. For example, Recast gave overlapping regions in some mazes, and it accidentally connected layers vertically in the *Tower* environment. Still, we acknowledge that metrics can never fully replace visual inspection.

Complexity. NEOGEN typically yielded the smallest graph in exchange for the highest average region complexity. This makes sense because the algorithm is deliberately designed to produce a small number of regions [112]. Its convexity relaxation parameter could enlarge this effect. The ECM often produced smaller graphs than the LCT, while the LCT usually had a lower total region complexity.

Recast appears to average between graph complexity and region complexity using our current settings. The method contains several parameters (such as the maximum number of vertices per region) with which this balance can be controlled. Recast and NEOGEN may not always capture all details of $\mathcal{E}_{free}$, but this does generally lead to simpler navigation meshes. This is useful for applications in which low storage requirements and fast path planning are more important than perfect coverage. In fact, exact methods could also benefit from a pre-processing step that simplifies $\mathcal{E}_{free}$.

As expected, our grid implementation always gave the largest graph, except in some of the mazes. This confirms that grids are usually inefficient representations, although we acknowledge their ease of use and their attractiveness for grid-aligned applications.

Performance. The LCT implementation was by far the fastest in all environments, although it required pre-processing that we have not included in our measurements. As expected, exact methods scaled better to large environments than voxel-based methods: while the LCT and the ECM remained fast, the running times increased strongly for Recast and the CDG in particular. NEOGEN was usually the fastest voxel-based method. The *BigCity* environment challenged the limits of all voxel-based methods: only Recast could produce a navigation mesh using our settings. Increasing the voxel resolution caused Recast to crash as well, most likely due to memory usage. Recast can subdivide the environment into *tiles* to alleviate this, but we have currently excluded this extra parameter to simplify our comparison.

The differences in scalability are difficult to judge because voxel-based methods include the reconstruction of $\mathcal{E}_{free}$ in their algorithm. Combined with the results for coverage, this indicates that obtaining $\mathcal{E}_{free}$ *without* voxels is an interesting topic for future work.

7.8 Conclusions and Future Work

A navigation mesh enables path planning and crowd simulation for walking characters in 2D and 3D environments. Various navigation meshes exist that can be computed (semi-)automatically. In this chapter, we have performed a comparative study of multiple state-of-the-art navigation meshes. We have proposed properties by which a mesh and its construction algorithm can be classified, and metrics that measure the quality of a mesh in practice. We have used these components to compare the Local Clearance Triangulation, the Explicit Corridor Map, the Clearance Disk Graph, NEOGEN, and a grid.

While we intend to use more environments, metrics, and settings, our results already suggest interesting properties. Voxel-based methods can be tuned to yield good coverage, but they do not always preserve connectivity, and their construction time does not seem to scale well to physically large environments. Furthermore, grids are usually not space-efficient representations of $\mathcal{E}_{free}$, although they may be

attractive for particular applications.

The goal of this chapter was not to find 'the best' navigation mesh, but to develop a way of comparing navigation meshes based on theoretical properties and quantitative metrics. Users may decide which properties and metrics are the most relevant for their application. We expect that this study will set a new standard for the evaluation and development of navigation meshes, and that it can help users choose an appropriate navigation mesh for particular applications.

Discussion. A limitation of our comparison lies in the current set of input environments. We have focused on examples from previous publications; these are all realistic scenarios that have been considered interesting before. Also, we have deliberately only used 'clean' walkable environments and not raw 3D geometry, to allow a fair comparison between exact and voxel-based methods for the same input. Our current goal was not to provide an exhaustive set of environments, nor to expose all strengths and weaknesses of an implementation. Ultimately, it would be good to create an open database of input environments for researchers to use, similarly to the ones that currently exist for local character steering [135] and grid-based A* search [139].

We would also like to investigate more types of metrics. For instance, it would be useful to measure the efficiency of a navigation mesh for path planning: how much time does it take to compute paths in $\mathcal{G}$, and how efficiently can these be converted to geometric routes using the regions of $\mathcal{R}$? Another option is to look at the *quality* of these routes: how short are they, and how well do they correspond to real-life behavior? If shortness is important, a dense grid may yield better results than a navigation mesh with a small dual graph. Ultimately, for real-world applications, we would like to quantify how well a navigation mesh captures the navigation abilities of real humans. This is a challenging direction for future work. We expect that not everything can be analyzed mechanically, and that user studies will also be required.

Finally, to simplify the comparison, we have chosen a single set of parameter settings for all methods. It would be interesting to see how different settings influence the results for each method, and how these settings can be optimized for particular metrics. For example, Oliva and Pelechano have discussed how the voxel size affects the results of Recast and NEOGEN [114]. We would like to combine such ideas with the quantitative metrics of this chapter.

Future work. Aside from these discussion points, a topic for future work lies in developing *exact* algorithms that automatically extract the walkable space from arbitrary 3D input in real-time. Our experiments suggest that voxel-based approaches do not always preserve coverage and connectivity, and that they are not very scalable to large environments. However, an advantage of voxelization is that the input is automatically simplified to a certain level of precision. *Exact* filtering algorithms should yield a perfect representation of $\mathcal{E}_{free}$ within provable time bounds, but they may be sensitive to small details or imprecisions in the

input (such as gaps or overlap). We have used a preliminary version of exact filtering software to generate some of our input environments, but this program does not yet achieve real-time performance. In the end, it may turn out that the best solution is to combine various approaches, e.g. a filtering algorithm without voxels that is based on rounded coordinates.

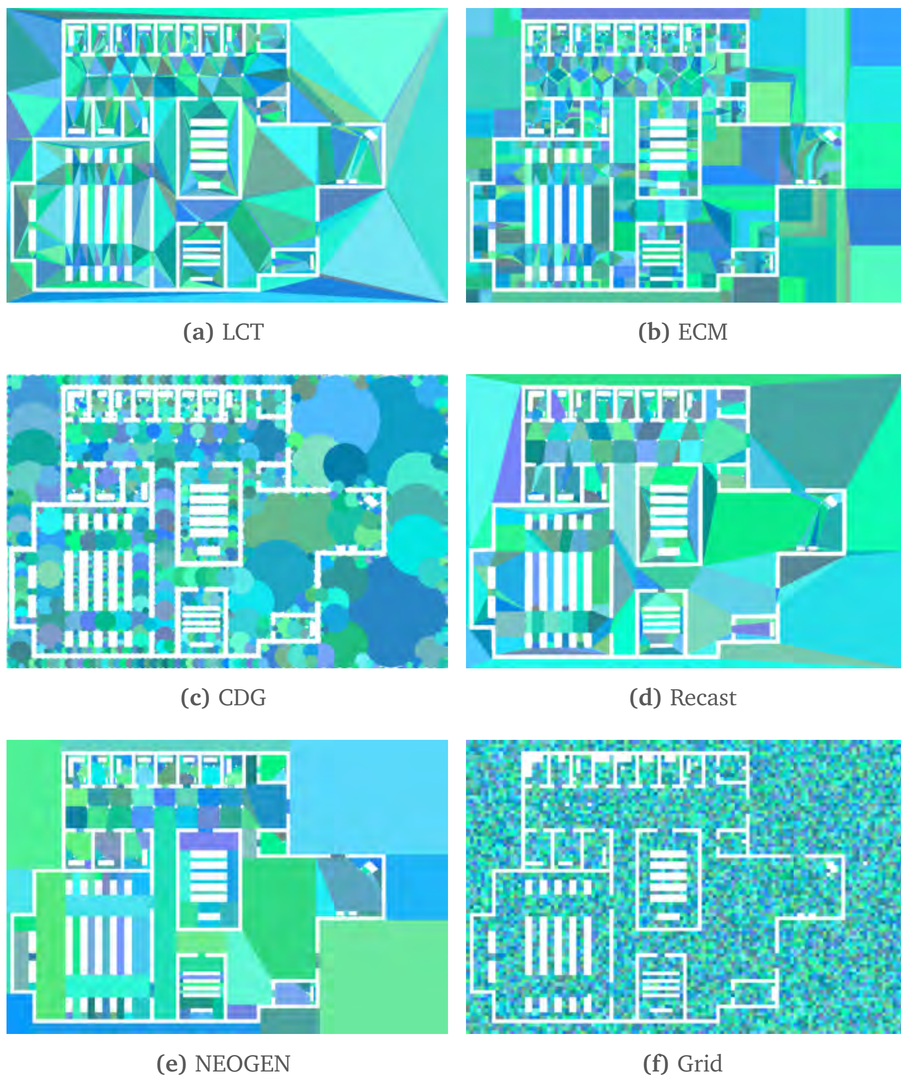
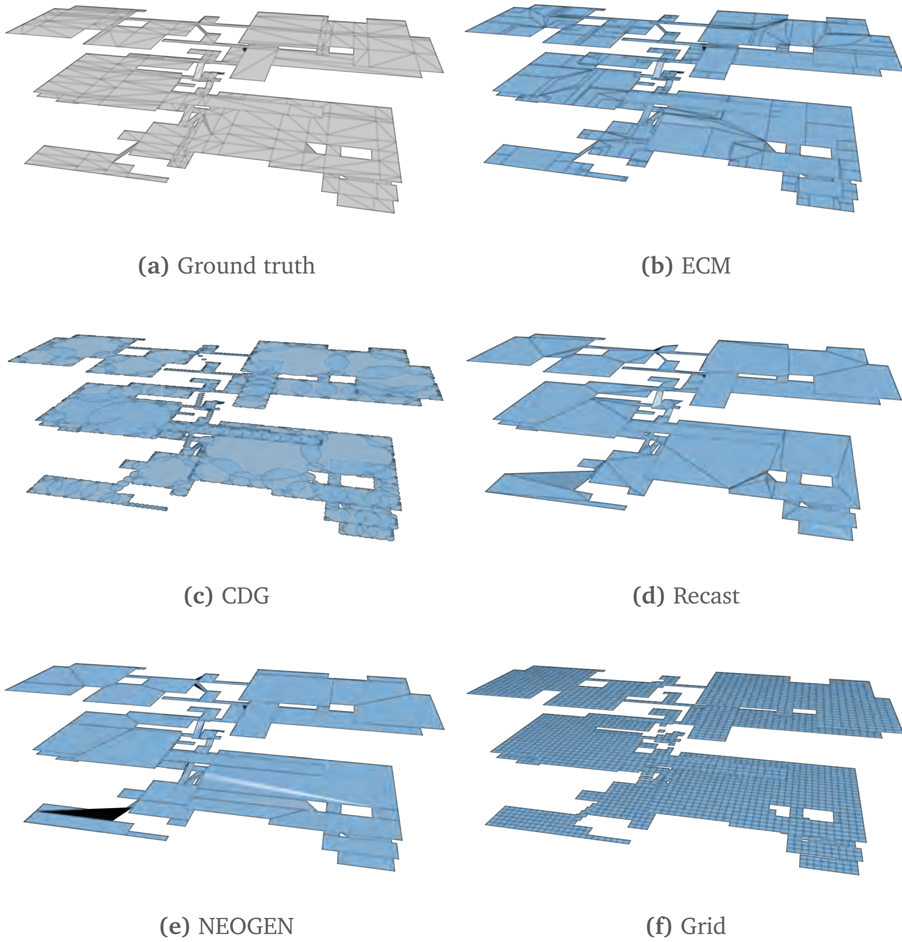


Figure 7.6: Navigation meshes computed for the *University* environment. Regions are shown in different colors. The corresponding graphs have been omitted for clarity.



(a) Ground truth

(b) ECM

(c) CDG

(d) Recast

(e) NEOGEN

(f) Grid

Figure 7.7: Renderings of the navigation meshes computed for the *Library* environment. Regions are shown in blue and outlined in black. The corresponding graphs have been omitted for clarity.

Environment	Total area (m^2)	Navigation mesh	Free space covered Absolute	Free space covered Relative	Incorrect area Absolute	Incorrect area Relative	Overlap Absolute	Overlap Relative
Military	36,876.82	LCT	36,876.82	1.00	0.00	0.00	0.01	0.00
		ECM	36,876.77	1.00	0.00	0.00	0.04	0.00
		CDG	36,198.31	0.98	7.90	0.00	15,613.55	0.30
		Recast	36,629.42	0.99	7.27	0.00	0.01	0.00
		NEOGEN	36,876.82	1.00	0.00	0.00	0.00	0.00
		Grid	36,736.63	1.00	107.37	0.00	0.00	0.00
University	8,370.68	LCT	8,370.68	1.00	0.18	0.00	2.22	0.00
		ECM	8,370.68	1.00	0.00	0.00	0.00	0.00
		CDG	7,821.35	0.93	11.42	0.00	3,387.29	0.30
		Recast	8,106.06	0.97	51.38	0.01	0.00	0.00
		NEOGEN	8,370.68	1.00	0.00	0.00	0.00	0.00
		Grid	8,335.13	1.00	27.87	0.00	0.00	0.00
Zelda	5,642.25	LCT	5,642.25	1.00	0.00	0.00	0.00	0.00
		ECM	5,642.23	1.00	0.00	0.00	0.01	0.00
		CDG	5,262.52	0.93	10.40	0.00	2,311.22	0.31
		Recast	5,457.23	0.97	35.56	0.01	0.00	0.00
		NEOGEN	5,642.24	1.00	0.00	0.00	0.00	0.00
		Grid	5,545.39	0.98	135.61	0.02	0.00	0.00
Zelda2x2	22,632.42	LCT	22,632.42	1.00	0.00	0.00	0.00	0.00
		ECM	22,632.39	1.00	0.02	0.00	0.02	0.00
		CDG	19,364.63	0.86	85.68	0.00	6,790.28	0.26
		Recast	21,900.31	0.97	122.55	0.01	0.00	0.00
		NEOGEN	22,632.42	1.00	0.00	0.00	0.00	0.00
		Grid	22,178.87	0.98	484.12	0.02	0.01	0.00
Zelda4x4	90,529.70	LCT	90,529.69	1.00	0.00	0.00	0.00	0.00
		ECM	90,529.58	1.00	0.07	0.00	0.07	0.00
		CDG	65,922.95	0.73	842.89	0.01	16,245.82	0.20
		Recast	87,652.50	0.97	542.35	0.01	0.00	0.00
		NEOGEN	90,529.69	1.00	0.01	0.00	0.00	0.00
		Grid	88,679.73	0.98	1,911.23	0.02	0.03	0.00
City	207,518.40	LCT	207,518.40	1.00	0.00	0.00	0.01	0.00
		ECM	207,518.10	1.00	0.09	0.00	0.19	0.00
		CDG	197,001.40	0.95	302.53	0.00	82,909.38	0.30
		Recast	206,304.50	0.99	103.55	0.00	0.40	0.00
		NEOGEN	207,518.40	1.00	0.04	0.00	0.00	0.00
		Grid	206,050.10	0.99	1,524.91	0.01	0.00	0.00
Maze8	31.00	LCT	31.00	1.00	0.00	0.00	0.00	0.00
		ECM	31.00	1.00	0.00	0.00	0.00	0.00
		CDG	22.10	0.71	0.35	0.01	6.37	0.22
		Recast	23.23	0.75	0.57	0.02	0.00	0.00
		NEOGEN	29.00	0.94	0.00	0.00	0.00	0.00
		Grid	31.00	1.00	0.00	0.00	0.00	0.00
Maze16	127.00	LCT	127.00	1.00	0.00	0.00	0.00	0.00
		ECM	127.00	1.00	0.00	0.00	0.00	0.00
		CDG	90.96	0.72	1.77	0.01	35.08	0.28
		Recast	100.47	0.79	2.15	0.02	0.00	0.00
		NEOGEN	125.00	0.98	0.00	0.00	0.00	0.00
		Grid	126.00	0.99	0.00	0.00	0.00	0.00
Maze32	511.00	LCT	511.00	1.00	0.00	0.00	0.00	0.00
		ECM	511.00	1.00	0.00	0.00	0.00	0.00
		CDG	352.45	0.69	6.36	0.01	74.51	0.17
		Recast	418.44	0.82	10.61	0.03	0.00	0.00
		NEOGEN	507.00	0.99	0.00	0.00	0.00	0.00
		Grid	511.00	1.00	0.00	0.00	0.00	0.00
Maze64	2,047.00	LCT	2,047.00	1.00	0.00	0.00	0.00	0.00
		ECM	2,047.00	1.00	0.00	0.00	0.00	0.00
		CDG	1,445.65	0.71	25.04	0.01	503.57	0.26
		Recast	1,415.99	0.69	28.25	0.02	0.00	0.00
		NEOGEN	2,003.00	0.98	0.00	0.00	0.00	0.00
		Grid	2,045.00	1.00	0.00	0.00	0.00	0.00
Maze128	8,191.00	LCT	8,191.00	1.00	0.00	0.00	0.00	0.00
		ECM	8,191.00	1.00	0.00	0.00	0.00	0.00
		CDG	5,066.94	0.62	109.58	0.02	896.21	0.15
		Recast	7,464.19	0.91	6,428.45	0.52	4,807.87	0.39
		NEOGEN	7,951.00	0.97	0.00	0.00	0.00	0.00
		Grid	8,184.00	1.00	0.00	0.00	0.00	0.00

Table 7.3: Results for the coverage metrics in the 2D environments. The descriptions of all metrics can be found in Section 7.6.

Environment		Navigation mesh	Coverage					
			Free space covered		Incorrect area		Overlap	
	Total area (m^2)		Absolute	Relative	Absolute	Relative	Absolute	Relative
ParkingLot	1,921.50	ECM	1,921.50	1.00	0.00	0.00	0.00	0.00
		CDG	1,819.68	0.95	0.07	0.00	675.84	0.27
		Recast	1,849.52	0.96	1.24	0.00	0.00	0.00
		NEOGEN	1,920.88	1.00	0.63	0.00	0.00	0.00
		Grid	1,861.50	0.97	4.51	0.00	0.00	0.00
Library	3,154.06	ECM	3,153.78	1.00	8.23	0.00	0.00	0.00
		CDG	2,951.06	0.94	9.62	0.00	1,108.37	0.27
		Recast	3,046.73	0.97	9.14	0.00	0.41	0.00
		NEOGEN	3,132.65	0.99	21.41	0.01	0.00	0.00
		Grid	2,947.75	0.93	243.26	0.08	0.00	0.00
Oilrig	75,746.52	ECM	?	?	?	?	?	?
		CDG	70,551.81	0.93	3,688.22	0.04	27,946.73	0.28
		Recast	74,933.66	0.99	124.29	0.00	3.30	0.00
		NEOGEN	73,475.38	0.97	2,271.13	0.03	0.00	0.00
		Grid	74,183.09	0.98	1,714.92	0.02	0.00	0.00
Neogen1	4,748.44	ECM	?	?	?	?	?	?
		CDG	4,341.39	0.91	550.79	0.09	1,731.02	0.29
		Recast	4,659.20	0.98	16.38	0.00	7.65	0.00
		NEOGEN	4,519.90	0.95	203.78	0.05	0.00	0.00
		Grid	4,449.82	0.94	70.04	0.02	0.00	0.00
Neogen2	9,600.56	ECM	?	?	?	?	?	?
		CDG	9,237.81	0.96	214.07	0.02	4,120.91	0.31
		Recast	9,431.68	0.98	14.29	0.00	5.07	0.00
		NEOGEN	9,371.87	0.98	20.71	0.00	0.00	0.00
		Grid	9,334.44	0.97	237.17	0.03	0.00	0.00
Neogen3	9,642.51	ECM	?	?	?	?	?	?
		CDG	9,349.64	0.97	35.66	0.00	3,884.67	0.29
		Recast	9,440.83	0.98	8.82	0.00	0.57	0.00
		NEOGEN	9,527.26	0.99	15.06	0.00	0.00	0.00
		Grid	9,297.01	0.96	133.04	0.01	0.00	0.00
Tower	12,093.88	ECM	?	?	?	?	?	?
		CDG	10,615.44	0.88	141.53	0.01	3,746.06	0.26
		Recast	11,672.40	0.97	425.70	0.04	5.26	0.00
		NEOGEN	-	-	-	-	-	-
		Grid	11,121.91	0.92	945.09	0.09	0.00	0.00
BigCity	280,897.00	ECM	?	?	?	?	?	?
		CDG	-	-	-	-	-	-
		Recast	277,326.20	0.99	2,820.01	0.01	15.14	0.00
		NEOGEN	-	-	-	-	-	-
		Grid	-	-	-	-	-	-

Table 7.4: Results for the coverage metrics in the multi-layered environments. An empty row indicates that the navigation mesh could not be computed for the corresponding algorithm and environment. A question mark indicates that the navigation mesh exists, but that our software could not compute its coverage metrics due to memory limitations.

Environment			Navigation mesh	Connectivity	
	#CCs	#Boundaries		#CCs	#Boundaries
Military	1	16	LCT	1	16
			ECM	1	16
			CDG	3	176
			Recast	1	16
			NEOGEN	1	16
			Grid	1	17
University	1	82	LCT	2	87
			ECM	1	82
			CDG	274	214
			Recast	1	81
			NEOGEN	1	82
			Grid	1	45
Zelda	1	57	LCT	2	57
			ECM	1	57
			CDG	609	211
			Recast	1	57
			NEOGEN	1	57
			Grid	1	59
Zelda2x2	1	226	LCT	1	226
			ECM	1	226
			CDG	9	594
			Recast	1	226
			NEOGEN	1	226
			Grid	1	227
Zelda4x4	1	906	LCT	1	906
			ECM	1	906
			CDG	244	1,536
			Recast	1	906
			NEOGEN	1	906
			Grid	1	921
City	1	181	LCT	1	181
			ECM	1	181
			CDG	204	367
			Recast	3	183
			NEOGEN	4	185
			Grid	2	203
Maze8	1	1	LCT	1	1
			ECM	1	1
			CDG	11	4
			Recast	1	1
			NEOGEN	2	2
			Grid	1	1
Maze16	1	1	LCT	1	1
			ECM	1	1
			CDG	29	39
			Recast	1	1
			NEOGEN	3	3
			Grid	2	2
Maze32	1	1	LCT	1	1
			ECM	1	1
			CDG	92	135
			Recast	1	1
			NEOGEN	11	11
			Grid	1	1
Maze64	1	1	LCT	1	1
			ECM	1	1
			CDG	618	374
			Recast	30	11
			NEOGEN	45	45
			Grid	4	1
Maze128	1	1	LCT	1	1
			ECM	1	1
			CDG	1,188	3,548
			Recast	123	89
			NEOGEN	164	164
			Grid	8	8

Table 7.5: Results for the connectivity metrics in the 2D environments. '#CCs' is an abbreviation of 'number of connected components'. The descriptions of all metrics can be found in Section 7.6.

Environment	#CCs	#Boundaries	Navigation mesh	Connectivity #CCs	Connectivity #Boundaries
ParkingLot	1	9	ECM	1	9
			CDG	178	81
			Recast	2	10
			NEOGEN	1	9
			Grid	4	11
Library	1	4	ECM	1	4
			CDG	256	137
			Recast	1	4
			NEOGEN	1	4
			Grid	1	4
Oilrig	1	26	ECM	1	26
			CDG	578	468
			Recast	1	29
			NEOGEN	1	26
			Grid	1	30
Neogen1	3	12	ECM	3	12
			CDG	43	176
			Recast	3	17
			NEOGEN	3	15
			Grid	6	46
Neogen2	11	33	ECM	11	33
			CDG	175	159
			Recast	10	27
			NEOGEN	6	27
			Grid	10	36
Neogen3	10	20	ECM	10	20
			CDG	58	347
			Recast	15	22
			NEOGEN	10	18
			Grid	28	27
Tower	1	19	ECM	1	19
			CDG	2,052	1,105
			Recast	1	256
			NEOGEN	-	-
			Grid	1	225
BigCity	1	301	ECM	1	301
			CDG	-	-
			Recast	8	1,796
			NEOGEN	-	-
			Grid	-	-

Table 7.6: Results for the connectivity metrics in the multi-layered environments. An empty row indicates that the navigation mesh could not be computed for the corresponding algorithm and environment.

Environment	Navigation mesh	Complexity					
					Region complexity		
		$\vert V\vert$	$\vert E\vert$	$\vert \mathcal{R}\vert$	Average	SD	Total
Military	LCT	120	134	120	9.00	0.00	1,080
	ECM	58	72	214	14.83	2.16	3,174
	CDG	1,168	2,078	1,168	4.00	0.00	4,672
	Recast	101	115	101	11.55	2.88	1,167
	NEOGEN	52	66	52	15.40	3.58	801
	Grid	36,844	72,755	36,844	12.00	0.00	442,128
University	LCT	732	812	732	9.00	0.00	6,588
	ECM	329	409	1,134	15.04	2.29	17,055
	CDG	3,309	4,369	3,309	4.00	0.00	13,236
	Recast	402	481	402	11.88	2.78	4,776
	NEOGEN	261	341	261	16.80	8.13	4,386
	Grid	8,363	15,460	8,363	12.00	0.00	100,356
Zelda	LCT	554	608	554	9.00	0.00	4,986
	ECM	289	344	895	14.93	2.30	13,359
	CDG	3,579	4,233	3,579	4.00	0.00	14,316
	Recast	321	376	321	12.12	2.77	3,891
	NEOGEN	205	260	205	16.04	6.31	3,288
	Grid	5,681	10,193	5,681	12.00	0.00	68,172
Zelda2x2	LCT	2,248	2,472	2,248	9.00	0.00	20,232
	ECM	1,148	1,372	3,602	14.92	2.30	53,754
	CDG	5,636	8,850	5,636	4.00	0.00	22,544
	Recast	1,281	1,505	1,281	12.16	2.84	15,573
	NEOGEN	820	1,044	820	16.15	6.35	13,245
	Grid	22,663	40,658	22,663	12.00	0.00	271,956
Zelda4x4	LCT	9,007	9,911	9,007	9.00	0.00	81,063
	ECM	4,580	5,484	14,436	14.92	2.30	215,424
	CDG	11,996	16,564	11,996	4.00	0.00	47,984
	Recast	5,105	6,009	5,105	12.17	2.84	62,148
	NEOGEN	3,289	4,193	3,289	16.15	6.36	53,103
	Grid	90,591	162,537	90,591	12.00	0.00	1,087,092
City	LCT	2,553	2,732	2,553	9.00	0.00	22,977
	ECM	1,442	1,621	4,679	14.42	2.16	67,491
	CDG	3,451	5,278	3,451	4.00	0.00	13,804
	Recast	1,527	1,706	1,527	11.90	3.08	18,168
	NEOGEN	1,164	1,343	1,164	13.61	5.17	15,846
	Grid	207,575	408,383	207,575	12.00	0.00	2,490,900
Maze8	LCT	26	25	26	9.00	0.00	234
	ECM	30	29	51	14.71	2.32	750
	CDG	68	59	68	4.00	0.00	272
	Recast	14	13	14	11.57	1.05	162
	NEOGEN	12	10	12	14.25	3.70	171
	Grid	31	30	31	12.00	0.00	372
Maze16	LCT	81	80	81	9.00	0.00	729
	ECM	84	83	156	14.85	2.25	2,316
	CDG	310	319	310	4.00	0.00	1,240
	Recast	42	41	42	11.71	1.44	492
	NEOGEN	39	36	39	14.62	3.05	570
	Grid	126	124	126	12.00	0.00	1,512
Maze32	LCT	363	362	363	9.00	0.00	3,267
	ECM	358	357	686	14.89	2.23	10,212
	CDG	1,084	1,138	1,084	4.00	0.00	4,336
	Recast	184	183	184	11.77	1.34	2,166
	NEOGEN	168	157	168	14.75	2.86	2,478
	Grid	511	510	511	12.00	0.00	6,132
Maze64	LCT	1,422	1,421	1,422	9.00	0.00	12,798
	ECM	1,392	1,391	2,672	14.88	2.24	39,756
	CDG	4,805	4,488	4,805	4.00	0.00	19,220
	Recast	602	572	602	11.63	1.45	6,999
	NEOGEN	636	591	636	14.74	2.91	9,372
	Grid	2,045	2,041	2,045	12.00	0.00	24,540
Maze128	LCT	5,674	5,673	5,674	9.00	0.00	51,066
	ECM	5,567	5,566	10,710	14.88	2.24	159,336
	CDG	25,135	44,910	25,135	4.00	0.00	100,540
	Recast	2,590	2,480	2,590	11.73	2.01	30,393
	NEOGEN	2,574	2,410	2,574	14.73	2.98	37,914
	Grid	8,184	8,176	8,184	12.00	0.00	98,208

Table 7.7: Results for the complexity metrics in the 2D environments. The descriptions of all metrics can be found in Section 7.6.

Environment	Navigation mesh	Complexity					
					Region complexity		
		$\lvert V \rvert$	$\lvert E \rvert$	$\lvert \mathcal{R} \rvert$	Average	SD	Total
ParkingLot	ECM	61	68	108	15.08	2.33	1,629
	CDG	779	873	779	4.00	0.00	3,116
	Recast	60	66	60	11.35	1.98	681
	NEOGEN	24	31	24	20.75	4.58	498
	Grid	1,866	3,493	1,866	12.00	0.00	22,392
Library	ECM	216	218	377	14.51	2.35	5,469
	CDG	1,758	2,173	1,758	4.00	0.00	7,032
	Recast	111	113	111	12.19	2.65	1,353
	NEOGEN	74	76	74	20.64	8.85	1,527
	Grid	3,191	5,801	3,191	12.00	0.00	38,292
Oilrig	ECM	603	629	1,283	14.72	2.23	18,891
	CDG	4,858	6,316	4,858	4.00	0.00	19,432
	Recast	324	353	324	12.35	3.08	4,002
	NEOGEN	253	279	253	23.44	12.99	5,931
	Grid	75,898	147,409	75,898	12.00	0.00	910,776
Neogen1	ECM	438	444	1,148	14.67	2.41	16,842
	CDG	1,017	1,651	1,017	4.00	0.00	4,068
	Recast	103	114	103	11.80	3.27	1,215
	NEOGEN	193	202	193	23.21	28.35	4,479
	Grid	4,519	8,494	4,519	12.00	0.00	54,228
Neogen2	ECM	390	403	1,240	14.86	2.32	18,426
	CDG	2,170	2,911	2,170	4.00	0.00	8,680
	Recast	198	213	198	11.74	2.97	2,325
	NEOGEN	295	312	295	15.31	7.11	4,515
	Grid	9,571	18,374	9,571	12.00	0.00	114,852
Neogen3	ECM	439	439	984	14.54	2.35	14,304
	CDG	2,070	3,319	2,070	4.00	0.00	8,280
	Recast	275	276	275	11.66	3.08	3,207
	NEOGEN	218	218	218	15.26	8.77	3,327
	Grid	9,430	17,962	9,430	12.00	0.00	113,160
Tower	ECM	4,988	5,019	9,416	14.37	2.38	135,300
	CDG	12,643	14,281	12,643	4.00	0.00	50,572
	Recast	1,155	1,437	1,155	12.47	3.02	14,400
	NEOGEN	-	-	-	-	-	-
	Grid	12,067	21,755	12,067	12.00	0.00	144,804
BigCity	ECM	32,167	32,554	69,176	14.41	2.37	997,053
	CDG	-	-	-	-	-	-
	Recast	9,394	11,345	9,394	12.27	3.11	115,260
	NEOGEN	-	-	-	-	-	-
	Grid	-	-	-	-	-	-

Table 7.8: Results for the complexity metrics in the multi-layered environments. An empty row indicates that the navigation mesh could not be computed for the corresponding algorithm and environment.

Environment	Navigation mesh	Performance			
		Construction time (ms)		Memory usage (MB)	
		Average	SD	Average	SD
Military	LCT	0.80	0.40	0.88	0.00
	ECM	7.23	0.01	29.99	0.02
	CDG	44,256.03	85.58	79.74	0.01
	Recast	1,055.51	9.84	24.76	0.00
	NEOGEN	15.90	0.70	66.53	0.32
	Grid	205.39	2.24	77.88	0.38
University	LCT	5.90	0.30	1.16	0.01
	ECM	31.14	0.06	33.02	0.02
	CDG	27,866.70	17.94	68.72	0.00
	Recast	209.72	1.79	9.08	0.00
	NEOGEN	94.90	2.51	86.46	3.07
	Grid	106.69	1.23	54.05	0.31
Zelda	LCT	4.60	0.49	1.07	0.00
	ECM	23.53	0.07	32.34	0.01
	CDG	28,802.21	26.55	68.47	0.00
	Recast	159.42	0.80	8.14	0.00
	NEOGEN	59.10	2.39	80.67	0.41
	Grid	92.35	0.85	51.69	0.28
Zelda2x2	LCT	19.00	0.77	1.79	0.00
	ECM	107.45	0.47	41.89	0.03
	CDG	31,915.01	57.39	75.47	0.03
	Recast	736.57	3.04	24.00	0.17
	NEOGEN	246.60	8.49	125.54	1.80
	Grid	201.52	1.97	73.82	0.24
Zelda4x4	LCT	96.21	1.08	4.73	0.00
	ECM	438.66	2.81	78.67	0.31
	CDG	39,629.89	40.84	105.19	0.01
	Recast	3,275.23	8.29	89.36	0.68
	NEOGEN	1,040.90	19.45	253.26	2.49
	Grid	796.62	13.17	153.34	0.30
City	LCT	33.40	0.49	2.11	0.00
	ECM	162.69	0.41	45.54	0.02
	CDG	44,140.58	380.99	83.88	0.02
	Recast	11,833.70	9.93	135.15	0.00
	NEOGEN	330.70	6.18	126.86	0.79
	Grid	1,431.77	10.35	200.95	3.17
Maze8	LCT	0.20	0.40	0.84	0.00
	ECM	1.54	0.01	29.40	0.00
	CDG	1,105.43	9.50	32.18	0.01
	Recast	1.00	0.00	2.12	0.00
	NEOGEN	3.20	0.60	63.72	2.02
	Grid	62.39	1.44	41.17	0.17
Maze16	LCT	0.70	0.46	0.88	0.00
	ECM	4.28	0.02	29.84	0.01
	CDG	2,508.26	9.16	38.90	0.01
	Recast	7.30	0.46	2.25	0.00
	NEOGEN	9.20	0.75	63.98	2.66
	Grid	63.77	1.32	42.09	0.22
Maze32	LCT	3.40	0.49	1.02	0.00
	ECM	18.05	0.03	31.87	0.00
	CDG	5,458.99	5.88	45.90	0.00
	Recast	391.64	0.66	2.83	0.00
	NEOGEN	38.90	0.83	77.59	1.89
	Grid	72.58	1.51	46.24	0.11
Maze64	LCT	12.50	0.67	1.64	0.00
	ECM	70.34	0.25	40.16	0.01
	CDG	13,448.17	19.51	55.80	0.01
	Recast	11,834.30	11.52	4.64	0.00
	NEOGEN	161.30	6.08	109.03	2.69
	Grid	94.05	0.62	54.51	0.31
Maze128	LCT	62.21	0.75	3.99	0.00
	ECM	306.97	0.63	71.75	0.23
	CDG	56,150.68	93.81	92.84	0.01
	Recast	52,746.20	19.62	11.19	0.00
	NEOGEN	654.80	12.03	199.44	1.89
	Grid	250.38	2.70	81.05	0.26

Table 7.9: Results for the performance metrics in the 2D environments. The descriptions of all metrics can be found in Section 7.6.

Environment	Navigation mesh	Performance			
		Construction time (ms)		Memory usage (MB)	
		Average	SD	Average	SD
ParkingLot	ECM	7.16	1.53	31.45	0.20
	CDG	6,686.30	18.02	53.33	0.01
	Recast	31.60	0.66	3.53	0.00
	NEOGEN	53.00	0.00	63.07	0.01
	Grid	88.84	3.13	49.18	0.29
Library	ECM	13.07	2.27	33.48	0.38
	CDG	11,374.29	31.39	59.02	0.01
	Recast	62.41	0.80	4.72	0.00
	NEOGEN	56.10	4.35	71.39	0.05
	Grid	211.99	22.58	54.12	0.29
Oilrig	ECM	55.67	10.68	38.52	0.76
	CDG	45,585.04	61.95	100.93	0.01
	Recast	2,280.03	6.43	49.79	0.23
	NEOGEN	219.80	6.18	107.94	0.49
	Grid	2,827.53	51.75	707.46	10.02
Neogen1	ECM	158.49	25.01	45.62	0.00
	CDG	19,911.95	24.38	101.37	0.03
	Recast	281.58	0.92	8.89	0.00
	NEOGEN	1,924.90	18.25	195.75	0.02
	Grid	1,498.18	43.75	192.09	0.12
Neogen2	ECM	43.40	6.22	37.58	0.46
	CDG	32,356.20	50.54	95.81	0.01
	Recast	208.62	1.28	8.58	0.00
	NEOGEN	245.60	4.20	93.33	0.45
	Grid	498.23	23.05	80.71	6.43
Neogen3	ECM	24.52	2.67	36.07	0.34
	CDG	32,919.54	80.55	82.33	0.01
	Recast	213.52	1.63	8.40	0.00
	NEOGEN	100.90	3.42	76.77	0.50
	Grid	523.35	54.51	109.36	1.40
Tower	ECM	139.45	17.94	72.63	0.58
	CDG	51,898.69	129.44	169.52	0.03
	Recast	538.25	3.79	16.22	0.05
	NEOGEN	-	-	-	-
	Grid	3,699.97	60.07	192.12	1.08
BigCity	ECM	756.03	5.45	299.44	0.90
	CDG	-	-	-	-
	Recast	16,353.00	28.15	216.66	0.20
	NEOGEN	-	-	-	-
	Grid	-	-	-	-

Table 7.10: Results for the performance metrics in the multi-layered environments. An empty row indicates that the navigation mesh could not be computed for the corresponding algorithm and environment.

PART III

Path Planning and Crowd Simulation Algorithms

8 Dynamic Re-planning

In this chapter, we consider path planning and crowd simulation in *dynamic environments*. We present algorithms for efficient re-planning of optimal or sub-optimal paths in dynamic navigation meshes.

This chapter is based on the following publication:

- W. van Toll and R. Geraerts. Dynamically Pruned A* for re-planning in navigation meshes. In *Proceedings of the 28th IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2051–2057, 2015. [149]

8.1 Introduction

Chapter 2 has explained how a character typically uses the A* algorithm to compute an optimal global path through the dual graph of a navigation mesh. This path can then be converted into an indicative route, which the character traverses while locally avoiding other characters.

In *dynamic environments*, obstacles can appear or disappear at run-time, which may have a large impact on the environment. For example, imagine a bridge collapsing or an explosion opening up a new route. When such a dynamic event occurs, local collision avoidance is not sufficient; instead, the navigation mesh should be updated. Chapter 6 has shown how these dynamic updates can be performed in the ECM.

In response to a dynamic event, characters should *re-plan* their global paths in the newly updated navigation mesh. Efficient re-planning algorithms already exist for graphs with dynamic costs and for high-dimensional motion planning problems. However, many of these algorithms require too much memory for crowds (because characters need to remember parts of the previous search), or they are difficult to implement for graphs in which vertices and edges are added or removed.

This chapter presents *Optimal Dynamically Pruned A** (ODPA*), an extension of A* that efficiently re-plans an optimal global path in a navigation mesh when an obstacle has been inserted or removed. Conceptually, ODPA* has similarities to so-called *adaptive search algorithms* that make the A* heuristic more informed based on previous queries. However, ODPA* prunes the A* search using *only* the previous path and its relation to the dynamic event. Characters do not need to remember additional information from the previous search. Our algorithm is

memory-friendly and robust against structural changes in the graph, which makes it suitable for crowds in dynamic navigation meshes. Because ODPA* is a graph search algorithm, it can be applied to any type of navigation mesh and all types of non-negative edge costs.

Experiments show that ODPA* performs particularly well in complex environments and when the dynamic event is visible to the character. We integrate the algorithm into crowd simulation software to model large crowds in dynamic environments in real-time.

ODPA* is an adaptation of Dynamically Pruned A* (DPA*), which has been presented in one of our publications [149]. DPA* assumed that an *inserted* obstacle could only lead to increased costs in the graph, and that a *deleted* obstacle could only lead to decreased costs. We have discovered that this assumption does not always hold; therefore, DPA* does not always compute optimal paths. This issue has been resolved in ODPA*.

The remainder of this chapter is structured as follows:

- Section 8.2 reviews related work on dynamic environments and re-planning.
- Section 8.3 describes the re-planning problem in an abstract way.
- In Section 8.4, we give a description and pseudocode of the ODPA* algorithm for re-planning optimal paths.
- As a side note, Section 8.5 describes a different re-planning technique that computes a suboptimal path by re-using more of the previous path.
- Section 8.6 analyzes the performance of ODPA* compared to standard A*, and it shows how the algorithms can be used in real-time crowd simulations.
- Section 8.7 concludes the chapter and provides directions for future work.

8.2 Related Work

We refer the reader to Chapter 2 for an overview of related work on navigation meshes and crowd simulation in *static* environments. Also, because the algorithms presented in this chapter are extensions of the A* search algorithm, it is useful to revisit Section 3.2, which describes A* globally.

In Section 6.2, we have discussed a number of possible representations of dynamic environments. For this chapter, we will focus on navigation meshes that can be updated dynamically [41, 67, 154].

8.2.1 Re-planning Algorithms

To efficiently re-plan optimal paths after a dynamic event, *incremental* variants of A* deal with changing costs by remembering information from the previous

query. Algorithms such as *D* Lite* and *Fringe-Saving A** remember the g and h values of each graph vertex and update the values that change due to the event [2, 85, 86, 144]. These are related to *anytime* algorithms that iteratively improve a sub-optimal path [8, 71, 97].

However, remembering the A* search space of each character is not feasible for large crowds. Also, re-planning in a dynamic navigation mesh is more complex than in a graph in which the costs change but the structure does not. A dynamic event may cause parts of the navigation mesh to (dis)appear, which also affects the underlying graph that is used for path planning. We cannot simply apply different costs to graph edges that already existed. Handling these effects in a memory-based algorithm is possible in theory, but difficult and costly in practice. For these reasons, ODPA* does not require memory of the previous search other than the path itself.

Another approach is to use *experience graphs* [122], in which only an abstract higher-level graph is remembered. This is particularly useful for high-dimensional motion planning problems; it is less applicable to our problem, since navigation meshes are already compact.

8.2.2 Comparison to Adaptive A*

The *Adaptive A** algorithm and its successors are closest to our work: they use information from the previous query to make the h values more informed in such a way that h remains consistent [50, 51, 52, 145]. Under certain conditions, the algorithms can immediately stop when a vertex of the old path is expanded. These algorithms require less memory of the previous search than e.g. D* Lite, and they are more suitable for dynamic navigation meshes.

By contrast, in this chapter, we do not make h more informed; instead, we use the estimated 'distance' to the dynamic event to find out if vertices can be skipped. Hence, we *prune* the A* search without changing any costs or heuristics. Our algorithms do not terminate until the goal vertex is expanded, which might be seen as a disadvantage compared to adaptive A*. However, in exchange, we use even less memory: characters only need to remember their paths, and not the way in which these paths have been computed.

In short, ODPA* investigates how parts of the A* search can be skipped after a dynamic event, without requiring extra memory between or during queries. Conceptually, the method lies between regular A* and adaptive re-planning algorithms. ODPA* is effective for real-time crowd simulation in dynamic environments.

8.3 Problem Description

8.3.1 Navigation Mesh and Graph

In this chapter, we assume that the characters use a *graph* for global planning. For navigation meshes, this is typically the *dual graph* of the walkable regions, as

explained in Chapters 2 and 3. An exception is the Explicit Corridor Map (ECM) from Chapters 4 and 5, in which the path planning graph is the medial axis.

We also ensure that characters always plan paths between two *vertices* of the graph. This can be achieved by connecting the start and goal positions to nearby vertices. In the case of the ECM, we can use the *retraction* operation from Chapter 4 that maps points in the free space to points on the medial axis.

It is important to see that we focus only on global path planning in the graph. The output of our ODPA* algorithm is a sequence of graph vertices to which we will simply refer as a *path*. In a crowd simulation, this path needs to be converted to an appropriate *indicative route* that the character can follow in real-time. Throughout this chapter, we will not concern ourselves with the computation of indicative routes, up to Section 8.6.2 in which we integrate ODPA* into crowd simulations.

8.3.2 Dynamic Events

In a dynamic environment, obstacles can be inserted, deleted, or moved during the simulation. Just like in Chapter 6, we focus on *insertions* and *deletions*. Moving obstacles can be represented by sequential deletions and insertions, or by locally avoidable entities until they become stationary. Such a *dynamic event* leads to an update of the navigation mesh: regions can be added, removed, split, or merged. Consequently, the structure of the dual graph also changes. As mentioned, this means that we cannot easily use re-planning algorithms designed for graphs in which only the costs are dynamic and the topology is static.

When an obstacle is inserted or removed, it affects only a certain part of the navigation mesh. Let the *affected region* $\mathcal{R}$ be the part of the graph that has changed, i.e. the set of vertices and edges that have appeared, disappeared, or changed. ODPA* will treat $\mathcal{R}$ as an area in which the *costs* have changed, regardless of what this area looked like before the event. Note that $\mathcal{R}$ is computed during the mesh update; we do not need to find it afterwards. Also, $\mathcal{R}$ can have any shape; it may even consist of multiple disconnected regions.

8.3.3 Re-planning Scenarios

Let S and G be the start and goal vertex of a character, as in Figure 8.1. Initially, the character uses A* to find an optimal path in the graph, which we call $[SG]^-$. The superscript $^-$ refers to *old* paths, computed *before* a dynamic event. Assume that an event occurs later in the simulation, and the character decides to re-plan when it has traversed the path up to a vertex T, e.g. because it can now see the event. The character should re-evaluate its path from T to G, i.e. $[TG]^-$. We assume that the affected region $\mathcal{R}$ (the set of vertices and edges with updated costs) was already computed during the mesh update.

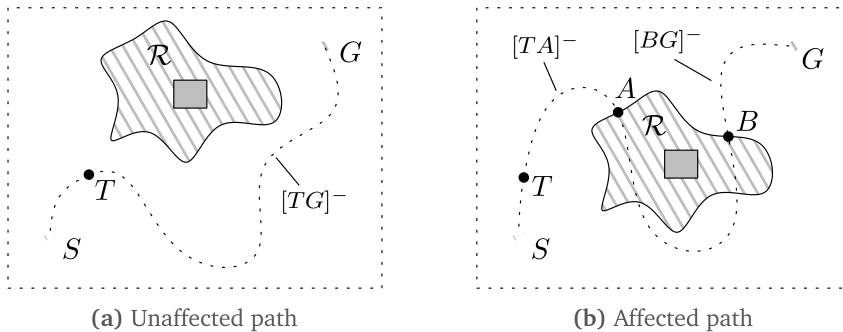
Our goal is to compute a new optimal path $[TG]^+$. The superscript $^+$ refers to *new* paths, i.e. paths computed *after* the dynamic event. The most straightforward way to compute $[TG]^+$ is to perform A* 'from scratch', i.e. to compute a new path

from T to G without re-using any information. However, ODPA* will improve the search by re-using information from $[TG]^-$.

There are two possible *re-planning scenarios*: $[TG]^-$ either does or does not run through $\mathcal{R}$. If $[TG]^-$ does *not* run through $\mathcal{R}$ (Figure 8.1a), we say that the old path is *unaffected*: $[TG]^-$ is still valid, but it may not be optimal anymore.

If $[TG]^-$ *does* run through $\mathcal{R}$ (Figure 8.1b), we say that the old path is *affected*. In this case, $[TG]^-$ can enter and exit $\mathcal{R}$ multiple times because $\mathcal{R}$ can be arbitrarily shaped. Let A and B be the first and last vertex in $\mathcal{R}$ that occur in $[TG]^-$. We split the path into three sections: two *valid* subsections $[TA]^-$ and $[BG]^-$, and one *invalid* subsection $[AB]^-$.

In some cases, the valid subsections $[TA]^-$ and $[BG]^-$ can be empty. If $[TA]^-$ is empty, then the graph has changed in the character's vicinity, and the character will need to perform a new point-location query in the navigation mesh. If $[BG]^-$ is empty, then the graph has changed near the goal, and we need a new point-location query to map the goal *position* to a new goal *vertex*. For simplicity, we will still use T and G to denote the new start and goal vertex, respectively.



(a) Unaffected path **(b)** Affected path

Figure 8.1: Re-planning scenarios after a dynamic event. A character is following a path from S to G and decides to re-plan at a vertex T. The inserted or deleted obstacle is shown as a gray rectangle, surrounded by the affected graph region $\mathcal{R}$. **(a)** If the old path $[TG]^-$ does not run through $\mathcal{R}$, then it is still valid, but possibly not optimal. **(b)** Otherwise, we define A and B as the first and last path vertex intersecting $\mathcal{R}$. The subpaths $[TA]^-$ and $[BG]^-$ are still valid and could be re-used in a re-planning algorithm.

In general, a new optimal path may enter and exit $\mathcal{R}$ multiple times; we cannot know in advance when this will happen. However, we do know that $\mathcal{R}$ is the *only* region in which the graph and its costs have changed.

8.4 Optimal Dynamically Pruned A*

We now present *Optimal Dynamically Pruned A** (ODPA*), an algorithm that adds *pruning rules* to standard A* search based on information from the old path. In

contrast to many other re-planning algorithms, we do not require extra memory of the search space throughout the simulation.

We have deliberately designed ODPA* in an abstract way such that it can be applied to all graphs with non-negative edge costs, including the dual graphs of navigation meshes. When using an admissible heuristic, ODPA* computes optimal paths. If the heuristic is not admissible, ODPA* may compute sub-optimal paths just like standard A*. However, inadmissible heuristics are only used when paths are *allowed* to be sub-optimal. In such cases, it may be preferable to use even faster algorithms that do not guarantee optimality; we will discuss this briefly in Section 8.5.

8.4.1 Scenario 1: Old Path Unaffected

We first consider the case in which $[TG]^-$ does *not* run through $\mathcal{R}$. Recall from Section 8.3.3 that the old path $[TG]^-$ is still entirely valid in this scenario. Because the graph costs have not changed outside of $\mathcal{R}$, $[TG]^-$ is still an optimal path from T to G among all possible paths that do not involve $\mathcal{R}$. More generally, for each vertex V on $[TG]^-$, the path $[VG]^-$ is still optimal among all paths that do not visit $\mathcal{R}$. Consequently, *if* a better path than $[TG]^-$ has appeared, then such a path *must* pass through $\mathcal{R}$ at least once. ODPA* therefore recognizes vertices for which a better path via $\mathcal{R}$ cannot exist.

For any vertex V in the graph, let $c^*(V, \mathcal{R})$ be the (currently unknown) optimal path cost from V to any vertex in $\mathcal{R}$. Let $h'(V, \mathcal{R})$ be a heuristic that does not overestimate this cost. For example, when using distance-based costs, $h'(V, \mathcal{R})$ could be the Euclidean distance from V to a bounding polygon of $\mathcal{R}$. Note that $h'(V, \mathcal{R}) = 0$ if $V \in \mathcal{R}$.

When expanding a vertex V, the costs for reaching the goal from V via $\mathcal{R}$ will be at least $h'(V, \mathcal{R}) + h'(G, \mathcal{R})$. Just like in standard A* (Section 3.2), let $g(V)$ be the new path cost to V that has been found during the current search. The cost of any path from T to G that (re-)visits $\mathcal{R}$ after V will be at least $g(V) + h'(V, \mathcal{R}) + h'(G, \mathcal{R})$. If this value is greater than or equal to the cost of the old path $[TG]^-$, then there is no point in visiting $\mathcal{R}$ from V, and we say that V is *$\mathcal{R}$-worse*. (Intuitively, if distance-based costs are used, we could say that $\mathcal{R}$ is 'too far away' to be used in combination with V.)

When it has been determined that visiting $\mathcal{R}$ from V is not useful, the same will hold for all subsequent vertices that the search will reach from V. In other words, all vertices explored from an $\mathcal{R}$-worse vertex will also be $\mathcal{R}$-worse themselves.

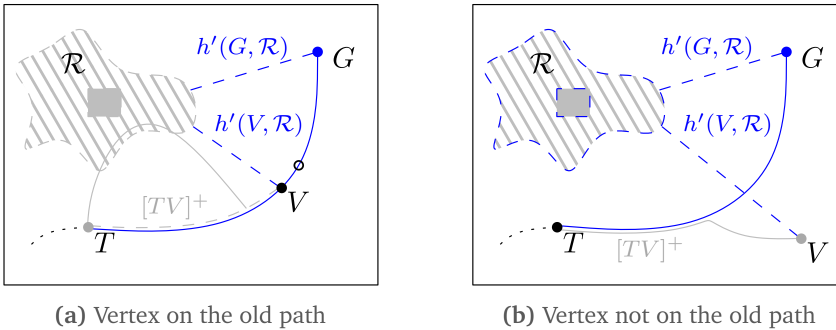
When ODPA* arrives at a vertex V *that is $\mathcal{R}$-worse*, there are two cases in which the search can be pruned:

- If $V \in [TG]^-$, then the old subpath $[VG]^-$ is still the optimal path from V to G. After all, any paths via $\mathcal{R}$ are too costly because V is $\mathcal{R}$-worse, and a path that does not visit $\mathcal{R}$ cannot be better than $[VG]^-$ (otherwise it would

have been found in the previous search). Thus, the best option from V is to follow the old path. Let V' be the *successor* of V in $[VG]^-$. ODPA* adds only V' to the open list; it can safely ignore all other neighboring vertices of V. (See Figure 8.2a.)

Note that the search *does not terminate yet* at this point. We only know the optimal path *via* V, which may not be the overall optimal path. There could still be better paths that meet $[TG]^-$ at a different vertex. Thus, the search continues, but we ignore all paths via V that will definitely not be better. (Some variants of Adaptive A* [52] *do* halt the search here, but this is only possible because they use more memory-intensive heuristics.)

- If $V \notin [TG]^-$ and $[TV]^+$ has not passed through $\mathcal{R}$ yet, then there is no better path via V at all. After all, $\mathcal{R}$ must be visited at least once to improve upon $[TG]^-$, but we have now established that V is $\mathcal{R}$-worse: we cannot reach $\mathcal{R}$ from V and still obtain a better path. Hence, ODPA* does not expand V any further. (See Figure 8.2b.)



(a) Vertex on the old path **(b)** Vertex not on the old path

Figure 8.2: Pruning rules of ODPA*. The algorithm has arrived at a vertex V that is $\mathcal{R}$-worse. That is, a better path from V via $\mathcal{R}$ cannot exist, based on the heuristic function h' (indicated in blue). **(a)** If $V \in [TG]^-$, then the best path to the goal is still $[VG]^-$. ODPA* adds only V's successor (black circle) to the open list. **(b)** If $V \notin [TG]^-$ and the new path $[TV]^+$ has not visited $\mathcal{R}$ yet, then there is no need to expand V.

This way, the open list contains only the vertices of $[TG]^-$, plus the vertices for which a better path through $\mathcal{R}$ might still exist. As such, ODPA* is still guaranteed to find an optimal path to G, either via the old path or via $\mathcal{R}$.

Algorithm 8.1 gives the pseudocode of ODPA* for this scenario. For clarity, all lines that also occur in standard A* (given in Algorithm 3.1) are shown in gray, and the extra lines for ODPA* are shown in black. To improve efficiency, we have added case distinction when checking which edges of V to explore (i.e. all neighbors, only its successor, or none). This postpones the check for $\mathcal{R}$-worseness until it can actually lead to pruning.

Our pseudocode also includes a *closed list* of previously expanded vertices to speed up the algorithm. Just like in standard A*, pruning the search based on the closed list is only safe when the heuristic function is consistent.

8.4.2 Scenario 2: Old Path Affected

If the old path $[TG]^-$ *does* pass through $\mathcal{R}$, the situation is more complex. In general, it is unknown if and where a new optimal path will visit $\mathcal{R}$, and it is unknown whether such a path will have a higher or lower cost than $[TG]^-$. However, we can use a weaker version of the algorithm for Scenario 1.

The information in $[BG]^-$, the valid subpart of $[TG]^-$ *after* $\mathcal{R}$, can be re-used almost as in Scenario 1. Assume that $[BG]^-$ is not empty. For each $V \in [BG]^-$, the old path $[VG]^-$ is still the optimal path from V to G among all paths that do *not* use $\mathcal{R}$. The only way to possibly improve upon $[VG]^-$ is to visit $\mathcal{R}$ at least once. Thus, whenever $h'(V,\mathcal{R}) + h'(G,\mathcal{R}) \geq \mathit{cost}([VG]^-)$, we know that visiting $\mathcal{R}$ from V cannot lead to improvements, and it suffices to add only the successor of V in $[TG]^-$ to the open list.

For vertices that do *not* lie on $[BG]^-$, we cannot do this because we do not have an upper bound of the new optimal path cost to G. Even for vertices on the other valid subsection $[TA]^-$, the new optimal path to G may have a higher cost than the old path. Thus, $[BG]^-$ is the only subsection that ODPA* re-uses. If $[BG]^-$ is empty because the graph has changed near the goal, then ODPA* reduces to standard A*.

The pseudocode of ODPA* for this scenario is given in Algorithm 8.2. To facilitate the algorithm, we precompute $\mathit{cost}([VG]^-)$ for each $V \in [BG]^-$. This can be performed in constant time per vertex by starting at G and tracing $[BG]^-$ backwards. Note that this was not necessary in Scenario 1 because we could use $\mathit{cost}([TG]^-)$ to classify all vertices.

8.5 Local Re-planning

Instead of looking for a new *optimal* path, a character may choose to repair its path only *locally* and re-use more of its previous path. There are many ways to define such a local re-planning strategy. In this section, we choose one version and we formalize it using the same terminology as before. We will not use it in our experiments; instead, this section can be seen as a side note for readers interested in implementing various re-planning strategies.

Again, let $[TG]^-$ be the old path when the character decides to re-plan.

- If $[TG]^-$ does *not* run through the affected region $\mathcal{R}$, then this path is still valid and the character keeps it.
- If $[TG]^-$ *does* run through $\mathcal{R}$, then the character plans a detour around the affected region. Specifically, it computes a new subpath $[AB]^+$ from A to B

Algorithm 8.1: ODPA*-PathUnaffected($T, G, [TG]^-, \mathcal{R}$)

1: $g(T) \leftarrow 0$, $T.parent \leftarrow$ NULL
2: $T.visitedR \leftarrow$ **false**, $T.rworse \leftarrow$ **false**
3: $OPEN \leftarrow \{T\}$, $CLOSED \leftarrow \emptyset$
4: **while** $OPEN \neq \emptyset$
5:     $V \leftarrow \text{argmin}_{V' \in OPEN}\{g(V') + h(V')\}$
6:     Remove V from $OPEN$
7:     Add V to $CLOSED$
8:     **if** $V = G$
9:         **return** the path from T to G via parent pointers
    {Determine how to expand V}
10:     $V.visitedR \leftarrow V.parent.visitedR$ **or** $V \in \mathcal{R}$
11:     $V.rworse \leftarrow V.parent.rworse$
12:     **if** $V \notin [TG]^-$ **and** $V.visitedR$
13:         $checkAll \leftarrow$ **true**
14:     **else**
15:         $V.rworse \leftarrow V.rworse$ **or** $g(V) + h'(V, \mathcal{R}) + h'(G, \mathcal{R}) > cost([TG]^-)$
16:         **if not** $V.rworse$
17:             $checkAll \leftarrow$ **true**
18:         **else if** $V \in [TG]^-$
19:             $checkAll \leftarrow$ **false**
20:         **else**
21:             **continue**
22:     **for each** edge (V, W)
23:         **if** $W \in CLOSED$
24:             **continue**
25:         **if not** $checkAll$ **and** $W \neq succ(V, [TG]^-)$
26:             **continue**
27:         **if** $g(V) + c(V, W) < g(W)$
28:             $g(W) \leftarrow g(V) + c(V, W)$
29:             $W.parent \leftarrow V$
30:             Insert or update W in $OPEN$
31: **return** NULL

Algorithm 8.2: ODPA*-PATHAFFECTED($T, G, [TG]^-, \mathcal{R}$)

```
1:  g(T) ← 0, T.parent ← NULL
2:  T.rworse ← false
3:  for each V ∈ [BG]⁻, starting at G
4:      Precompute cost([VG]⁻)
5:  OPEN ← {T}, CLOSED ← ∅
6:  while OPEN ≠ ∅
7:      V ← argmin_{V'∈OPEN}{g(V') + h(V')}
8:      Remove V from OPEN
9:      Add V to CLOSED
10:     if V = G
11:         return the path from T to G via parent pointers
        {Determine how to expand V}
12:     V.rworse ← V.parent.rworse
13:     if V ∉ [BG]⁻
14:         checkAll ← true
15:     else
16:         V.rworse ← V.rworse or h'(V, R) + h'(G, R) > cost([VG]⁻)
17:         if not V.rworse
18:             checkAll ← true
19:         else
20:             checkAll ← false
21:     for each edge (V, W)
22:         if W ∈ CLOSED
23:             continue
24:         if not checkAll and W ≠ succ(V, [BG]⁻)
25:             continue
26:         if g(V) + c(V, W) < g(W)
27:             g(W) ← g(V) + c(V, W)
28:             W.parent ← V
29:             Insert or update W in OPEN
30: return NULL
```

using regular A*. Figure 8.3 shows that $[AB]^+$ may partly overlap with the old subpaths $[TA]^-$ and $[BG]^-$. To keep the character from visiting a vertex more than once along its path, we resolve this overlap. Let A' be the first vertex along $[TA]^-$ that occurs in $[AB]^+$, and let B' be the last vertex along $[BG]^-$ that occurs in $[AB]^+$. We define the final path as the concatenation of $[TA']^-$, $[A'B']^+$, and $[B'G]^-$.

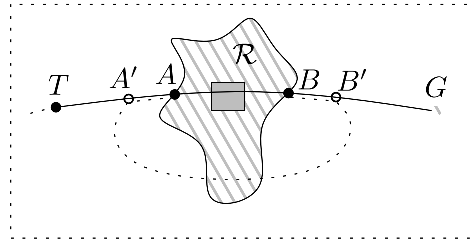


Figure 8.3: Local re-planning of an affected path. We first plan a new subpath $[AB]^+$ (dotted). To resolve overlap with $[TG]^-$, we locate the vertices A' and B' (shown as circles) where this overlap starts and ends. We use the concatenation of $[TA']^-$, $[A'B']^+$, and $[B'G]^-$ as our new result.

Naturally, local re-planning is usually very efficient because the path is only partly updated. However, the new path is not necessarily optimal. The difference to the optimal path grows in particular when many detours are planned in a row. On the other hand, this behavior might actually be desirable in some situations; hence, this local re-planning strategy can be used deliberately to obtain different character behavior.

8.6 Experiments and Results

We have implemented ODPA* for the dynamic Explicit Corridor Map (ECM) navigation mesh from Chapters 4 and 6.

We define the cost of an ECM edge e as the 2D curve length of e, and we use the 2D Euclidean distance to the goal as the heuristic function h. For dynamic deletions, we estimate the cost to $\mathcal{R}$ (i.e. the second heuristic h') by the distance to the axis-aligned bounding box of all affected vertices. This approximation allows us to compute h' values in constant time. We include a closed list in both ODPA* and regular A*, which is safe because h is consistent.

8.6.1 ODPA* versus A*

We compared the running times of ODPA* and A* in the environments shown in Figures 8.4 and 8.5. Details of the environments and their ECM navigation meshes can be found in Table 8.1. In each environment, we defined a number of

(a) *Military* **(b)** *City* **(c)** *Zelda*

Figure 8.4: The first three environments used in our experiment. Static geometry is shown in gray; dynamic obstacles are shown in black.

dynamic obstacles (squares of 2×2 m). For each such obstacle O, we performed the following steps:

1. Create 500 pairs of random positions (s, g) that do not intersect the environment or O.

2. For each position pair (s, g), compute a shortest path from s to g on the medial axis using the retraction method described in Section 4.3.3, and using a character radius of 0.7 m (a size that fits through all passages). Convert this ECM path to a short indicative route (IR) with a preferred clearance of 0.5 m (on top of the character's radius), as described in Section 4.3.4. This is the route that the character would follow in a crowd simulation.

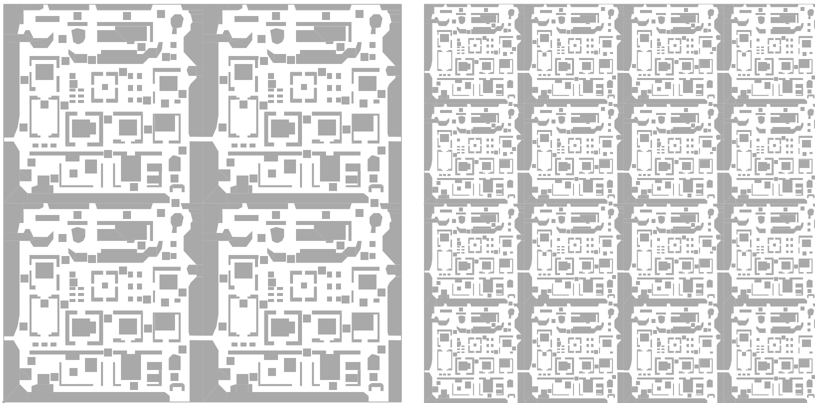
3. Insert O into the ECM dynamically. For each position pair, recompute the path in the ECM using both ODPA* and A*. Convert the new ECM path to an IR again; this route is recomputed from scratch.

4. Delete O dynamically. For each position pair, recompute the path using ODPA* and recompute the IR. (We can skip regular A*: it would give the same result as in Step 2.)

We always performed all steps for one obstacle before moving on to the next obstacle; hence, the environments contained at most one dynamic obstacle at a time. The average running time of all dynamic insertions in all environments was 0.34 ms (σ=0.16); the average time for deletions was 1.79 ms (σ=0.89).

Table 8.2 summarizes the performance of ODPA* compared to A* for both re-planning scenarios. We computed the *relative improvement* as $(A - D)/A \cdot 100\%$, where A is the sum of all A* times and D is the sum of all ODPA* times, over all trials that fit in one scenario. This is a good indication of the time that can be gained by using ODPA* instead of A* on a crowd with random characters.

For completeness, we have measured two variants of the relative improvement. The first variant (labelled as 'Path only' in Table 8.2) is based only on the running



(a) *Zelda2x2* **(b)** *Zelda4x4*

(c) *Zelda8x8*

Figure 8.5: The last three environments used in our experiment. Static geometry is shown in gray; dynamic obstacles are shown in black.

Environment	Size (m)	ECM vertices	ECM edges	Dynamic obstacles
Military	200x200	56	70	17
Zelda	100x100	288	343	24
City	500x500	1451	1631	70
Zelda2x2	200x200	1144	1368	106
Zelda4x4	400x400	4560	5464	235
Zelda8x8	800x800	18304	21936	470

Table 8.1: Details of the experimental environments. The third and fourth columns show the complexity of the ECM graph without dynamic obstacles. These numbers are slightly different from Chapter 4 because a different version of our software was used, with a slightly different treatment of e.g. degree-4 ECM vertices and nearly-collinear obstacle vertices. The fifth column shows the number of dynamic obstacles (the black squares in Figures 8.4 and 8.5).

time for ODPA* and A* itself, i.e. the time spent on recomputing the path in the ECM. This is the most pure comparison between ODPA* and A*. The second (labelled as 'Path + IR' in Table 8.2) includes both the path planning time *and* the time to (re)compute the indicative route. This is the most realistic comparison for crowd simulation purposes. Because indicative routes are always recomputed from scratch, the improvement for 'Path + IR' is generally slightly lower than the improvement for 'Path only'. We will only discuss the results for 'Path + IR' in the remainder of this section.

Results. ODPA* performs fewer operations on the A* open list in exchange for overhead, e.g. for finding the affected part of a path, and for estimating the distance to $\mathcal{R}$. In the *Military* environment, the ECM graph was too small for this to be beneficial, and regular A* was faster overall.

In the 'path unaffected' scenario, the relative improvement was positive in all other environments, and it increased along with the complexity of the graph, up to an improvement of 40% in *Zelda4x4*. This is logical because the dynamic event in our experiment had a relatively small effect on larger graphs, which allowed larger fractions of paths to be re-used. Hence, ODPA* appears to be good at checking whether an unaffected path in a large graph is still optimal. In preliminary experiments, we also observed that the algorithm is particularly fast when the dynamic obstacle is farther away from the path.

In the 'path affected' scenario, the improvements were small, up to 6% in *Zelda4x4*. However, as shown in the rightmost column of Table 8.2, this scenario occurred considerably less often than the 'path unaffected' scenario. Especially in complex graphs, the chance of a random path being affected by a local dynamic event is small.

While we expected to obtain the best improvement rates in the most complex environment, the results for *Zelda4x4* turned out better than those for *Zelda8x8*. A possible explanation for this is that the repetition of the *Zelda* environment in a grid pattern leads to many possible paths of comparable length, which may

	Relative improvement				
	Path unaffected		Path affected		% Affected
Environment	Path only	Path + IR	Path only	Path + IR	
Military	-16.85%	-18.18%	-13.97%	-15.08%	11.76%
Zelda	13.16%	11.15%	-6.90%	-8.07%	8.59%
City	23.42%	22.85%	0.90%	0.57%	6.12%
Zelda2x2	28.59%	27.77%	2.46%	2.05%	4.53%
Zelda4x4	40.54%	**40.24%**	6.30%	**6.09%**	2.03%
Zelda8x8	38.42%	38.56%	2.15%	2.32%	1.01%

Table 8.2: Results of the first experiment. All percentages denote the relative improvement of ODPA* over A*, computed as described in Section 8.6.1. A negative percentage (gray) means that A* was faster combined over all trials; a positive percentage (black) means that ODPA* was faster. The two percentages in bold blue are analyzed further in Figures 8.6a and 8.6b. The last column shows the percentage of all trials that corresponded to the 'path affected' scenario.

make the search slower and lead to less pruning. In future work, we would like to experiment with other large environments that have a different structure.

Visibility. We repeated this experiment with the extra constraint that all start positions lie in the *visibility polygon* [35] of the dynamic obstacle's center of mass. This simulates the effect that characters re-plan when they *see* the event. We computed visibility polygons using an algorithm based on the adjacency between ECM cells. Chapter 4 provides more details about this algorithm and its performance.

The results for this variant are shown in Table 8.3. As shown in the rightmost column, more paths were affected this time (up to 46% in *Zelda*). This was to be expected: if a character can see a dynamic event, it is more likely that this event affect the character's path. Fewer paths were affected in *Military* and *City* because these environments have more large open spaces: there, dynamic obstacles are more easily visible without necessarily being on the character's path.

In the 'path unaffected' scenario, we obtained a lower improvement than before because there was less room for pruning this time: if a dynamic event is visible, it is most likely nearby and ODPA* can less quickly qualify it as being 'too far away'. Still, the results were positive in all environments except *Military*, and we obtained an improvement of 28% in *Zelda4x4*.

In the 'path affected' scenario, which now occurred more often, the results strongly improved with respect to the first experiment. For instance, we obtained a relative improvement of 23% in *Zelda4x4*. In this scenario, if an event is visible from the starting point, then the event is more likely to be located at the beginning of the route than at the end. Thus, the valid subsection $[BG]^-$ is relatively long, which facilitates pruning. Overall, this experiment suggests that ODPA* can be used for efficient visiblity-based replanning in complex environments where dynamic events have a relatively small influence.

	Relative improvement (Source in visibility polygon)				
	Path unaffected		Path affected		% Affected
Environment	Path only	Path + IR	Path only	Path + IR	
Military	-17.38%	-19.00%	-18.88%	-20.77%	27.40%
Zelda	4.79%	2.98%	-5.27%	-7.12%	46.46%
City	15.84%	15.27%	6.44%	5.76%	31.61%
Zelda2x2	16.47%	15.81%	10.14%	9.25%	45.44%
Zelda4x4	28.43%	**28.15%**	23.18%	**22.83%**	45.84%
Zelda8x8	16.09%	16.26%	13.10%	13.17%	45.65%

Table 8.3: Results of the second experiment, in which the source point was always inside the visibility polygon of the dynamic obstacle. The two percentages in bold blue are analyzed further in Figures 8.6c and 8.6d.

Close-up. Figure 8.6 shows the results for the *Zelda4x4* environment in more detail: it compares the average running times of A* and ODPA* for each distinct number of vertices on the re-planned paths.

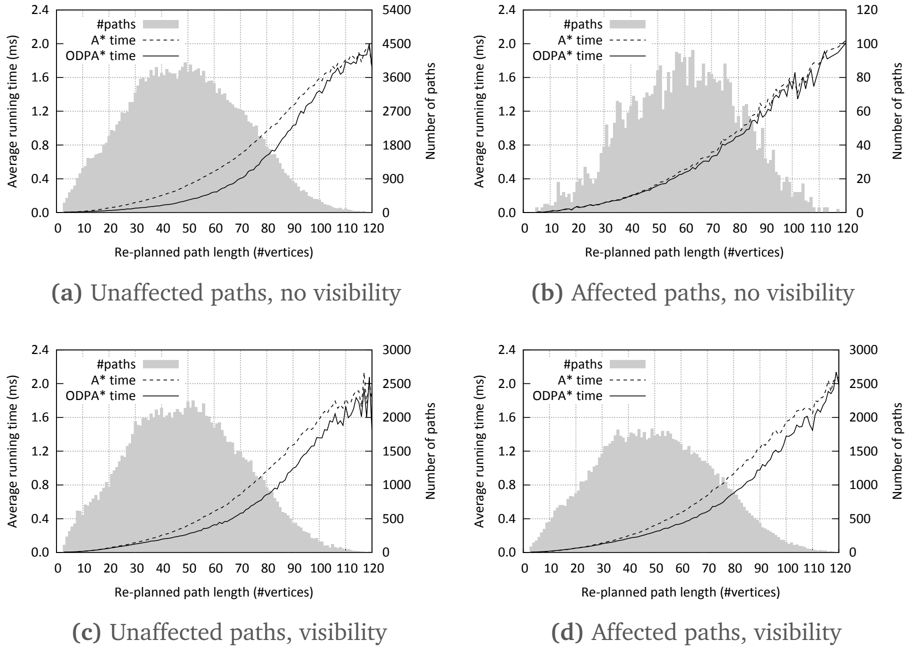
For paths of only a few vertices, both algorithms are fast and there is no clear difference between ODPA* and A*. As the paths become more complex and planning takes more time, the improvement of ODPA* becomes more evident, up to a certain point where the difference to A* starts decreasing again. A possible explanation for the latter effect is that long paths typically run between opposite corners of the environment. In such cases, it takes longer until vertices can be marked as $\mathcal{R}$-worse. Also, note that only a small fraction of all paths consisted of 100 or more vertices; therefore, the average running times vary more in this range.

We conclude that ODPA* is useful for checking if a path is still optimal after an event has occurred far away, and for updating an affected path as soon as the dynamic event is visible to the character. The improvement upon A* is generally better in larger graphs. In such environments, memory-based algorithms like D* Lite [85] and Adaptive A* [145] are likely to be too expensive for large crowds. The supplementary video of our original publication [149] shows visual examples of visibility-based re-planning compared to instantaneous re-planning.

8.6.2 Crowd Simulation

Finally, we have integrated ODPA* in our ECM-based crowd simulation software that will be described further in Chapter 10. More details on the architecture and performance of this program will be provided in Chapter 10; for now, it is sufficient to know that the software can simulate large crowds in real-time, and that ODPA* can be plugged into the framework.

Obstacles can be added and removed interactively; members of the crowd can use ODPA* to compute a new path through the ECM, which can then be converted to an indicative route. When using visibility as a trigger, re-planning



(a) Unaffected paths, no visibility

(b) Affected paths, no visibility

(c) Unaffected paths, visibility

(d) Affected paths, visibility

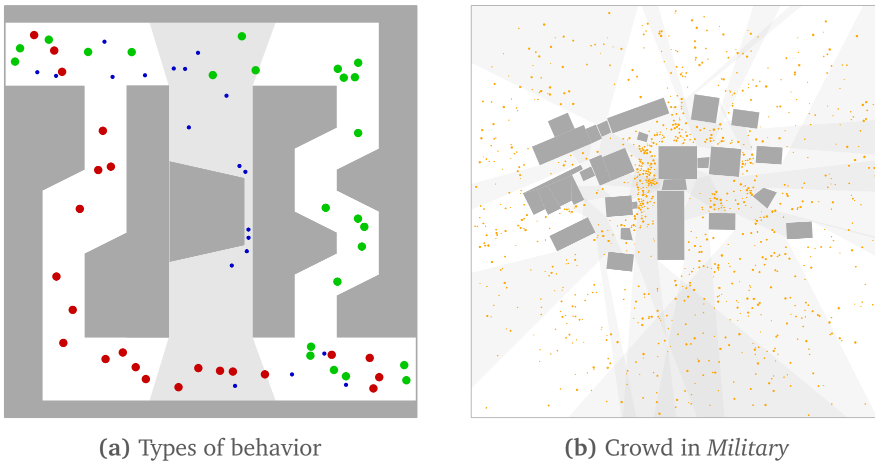
Figure 8.6: Detailed results of the re-planning experiment in *Zelda4x4*. The horizontal axis denotes the length (the number of vertices) of the re-planned path. The left vertical axis denotes the average running time of A* or ODPA* for each distinct path length. The gray histogram and the right vertical axis show how often each path length occurred. These figures correspond to the relative improvements of 40.24%, 6.09%, 28.15%, and 22.83% in Tables 8.2 and 8.3.

actions are automatically divided over time, allowing real-time performance. This performance may drop when an event happens to affect many characters at the same time. A solution could be to allow only a maximum number of re-planning actions per simulation step; the remaining characters would then respond slightly later in the simulation. Another idea would be to perform re-planning in a parallel thread while the simulation keeps running.

We will now show two examples of ODPA* being used in crowd simulations. Figure 8.7a demonstrates how visibility-based re-planning allows us to model different types of behavior. In this example, characters enter the environment on the left side and plan a shortest path to the right side. During the simulation, a dynamic obstacle is added in the middle. From that moment on, red characters know about this obstacle in advance and take the left route. Blue characters are small enough to take the middle route, thanks to the ECM's clearance information. Green characters do not know about the dynamic obstacle until they *see* it; by then, they re-plan and discover that the right route has become the shortest.

In Figure 8.7b, a crowd moves through the *Military* environment while obstacles

are dynamically inserted and deleted. We invite the reader to watch the original paper's supplementary video [149] for more (animated) examples.



(a) Types of behavior

(b) Crowd in *Military*

Figure 8.7: Crowd simulations using ODPA*. **(a)** An example with three routes and a dynamic obstacle in the middle. The characters take different routes based on their own (re-)planning properties. **(b)** A crowd of 1,000 characters moving through the *Military* environment. Dynamic obstacles can be added and removed during the simulation.

8.7 Conclusions and Future Work

In simulations and games, a dynamic navigation mesh represents an environment in which obstacles are inserted or deleted at runtime. After a dynamic event, characters in a virtual crowd should re-plan their paths. One way to re-plan a path after a dynamic event is to perform A* search from scratch, but it is intuitive to re-use information from the old path. Many existing re-planning algorithms are efficient in other applications, but they are not designed for large crowds or structural changes in the search space.

In this chapter, we have presented *Optimal Dynamically Pruned A** (ODPA*), which re-plans a path by adding pruning rules to A*, using only the old path and its relation to the dynamic event. The algorithm is defined for arbitrary graphs and costs, and it yields optimal paths when using admissible heuristics. Its focus is different to that of other re-planning algorithms: ODPA* is primarily meant as an improvement of A* for applications that have limited memory per character, such as simulations of large crowds.

Experiments show that A* is faster in small graphs, but that ODPA* can decrease the re-planning time in complex environments. Our algorithm is particularly efficient when there is more room for pruning, e.g. when the dynamic event is far away from an unaffected path or when the path is affected only at the

beginning. The latter situation is likely to occur when the dynamic event is within the character's visibility range. In conclusion, ODPA* is an intuitive extension of A* that can improve real-time crowd simulation in large dynamic environments.

Discussion and future work. The environments in our experiments were chosen to represent a range of graph complexities. However, the *Zelda* variants are all structurally similar, and the grid-based repetition pattern of *Zelda8x8* appeared to affect the results. To draw more general conclusions about the performance of ODPA*, we would like to test other large environments that are structured differently, such as maps of real-world cities.

It would be interesting to extend ODPA* to handle multiple dynamic events in a single re-planning query. One possible approach is to combine the affected regions from each event into a single region $\mathcal{R}$. However, our algorithm will be less efficient if $\mathcal{R}$ is large or if it consists of sub-regions scattered throughout the environment. In many cases, it may be better to perform a separate re-planning query per event.

It is also likely that ODPA* can be further specialized and improved for navigation meshes, e.g. by exploiting the facts that the graph is planar and costs are distance-based. However, for this chapter, we have chosen to develop a general algorithm that does not depend on these details.

Computing an *optimal* path may not be necessary in all applications. It would therefore be interesting to compare ODPA* to algorithms that are known to yield *suboptimal* paths in exchange for faster performance. Examples include our original DPA* algorithm [149] and the local re-planning strategy from Section 8.5. Such a comparison should consider running times as well as the length of the resulting paths and indicative routes.

We would like to simulate *incomplete knowledge* in the crowd by giving each character its own set of known and unknown events. Currently, characters know about all events when re-planning because they always use the most recent version of the mesh. However, implementing this incomplete knowledge efficiently will be challenging. Keeping multiple versions of the navigation mesh in memory will become infeasible at some point, as the number of versions will grow exponentially with the number of dynamic events. Instead, it seems better to compute the required version of the navigation mesh at the time of re-planning, but this will obviously affect the performance of the simulation.

Furthermore, we want to explore how knowledge about events propagates through the crowd. For instance, characters may recognize events via the behavior of other characters.

Finally, we are interested in other types of dynamic geometry, e.g. moving platforms that connect to different areas at different points in time. This asks for new types of navigation meshes and path planning algorithms.

9 Density-Based Crowd Simulation

In this chapter, we use *crowd density information* to improve global path planning for characters in a crowd, based on the observed relation between density and walking speed. This leads to more efficient and natural-looking crowd flows.

This chapter is based on the following publication:

- W.G. van Toll, A.F. Cook IV, and R. Geraerts. Real-time density-based crowd simulation. *Computer Animation and Virtual Worlds*, 23(1):59–69, 2012. [155]

9.1 Introduction

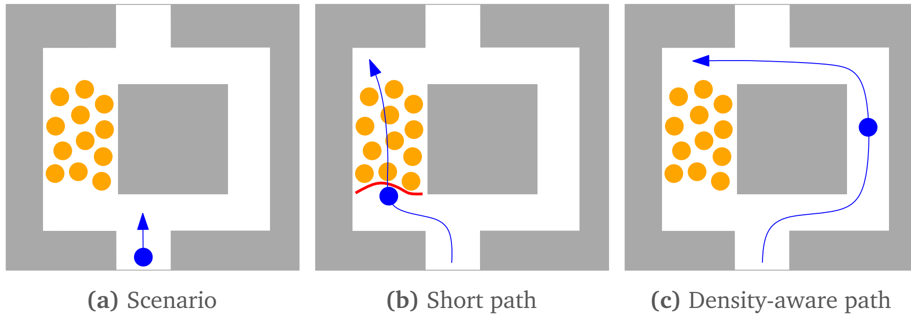
Virtual characters often need to plan visually convincing paths through a crowded environment. Such paths should be easy to compute and should permit characters to avoid static obstacles as well as other moving characters. In many applications, it is sufficient to let characters take the *shortest* path through the dual graph of a navigation mesh, as outlined in Chapter 3. However, in crowded environments, this may cause some areas to be used by many characters while other areas remain largely unused. An example is shown in Figure 9.2a.

The real-world concept of *crowd density* is often expressed in persons per square meter. Crowd density is an indicator of the safety of pedestrians in a crowd. High densities are preferably avoided because they can lead to dangerous situations and, in the worst case, many human casualties. Examples of crowded events with disastrous outcomes include the 2010 Love Parade in Duisburg [47] and the 2015 Hajj pilgrimage in Mina[1].

In this chapter, we use the concept of crowd density to improve global path planning. Figure 9.1 shows the intuition behind our approach: density-based path planning allows characters to prefer detours around congested areas, which avoids more high-density scenarios and collision-avoidance problems than a simulation *without* density information.

More specifically, our method keeps track of the current crowd density in each region of a navigation mesh. Based on the real-world relation between crowd density and typical walking speed [160], also known as a *fundamental diagram* [130], we convert these density values to edge costs in the path planning graph.

[1] See e.g. https://en.wikipedia.org/wiki/2015_Mina_stampede.



(a) Scenario (b) Short path (c) Density-aware path

Figure 9.1: Introduction to density-based path planning. **(a)** A blue character wants to move to the top-left corner of an environment. The left half of the environment is occupied by many other characters (shown in orange). **(b)** If the blue character follows the shortest route to its goal, it is likely to get stuck in the congested region. **(c)** If we include *density* information, the character can prefer a detour along which the crowd density is lower.

These costs can roughly be interpreted as the 'estimated traversal time' for an edge. Characters perform an A* search on this weighted graph to plan a density-aware path through the environment. Such a path can then be converted to an indicative route for the character to follow in real-time, but (as in Chapter 8) we will not focus on these aspects of the simulation.

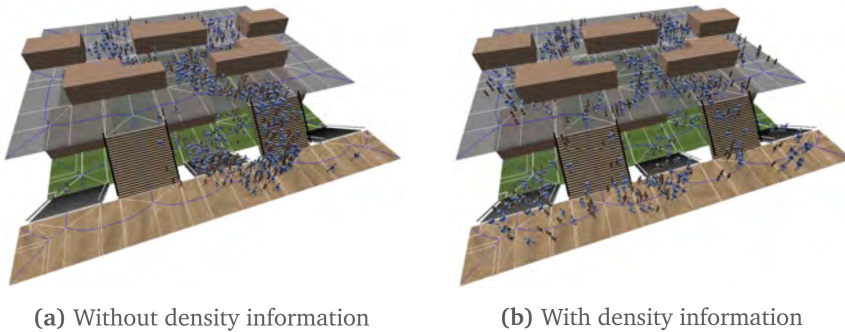
Periodic *re-planning* allows characters to respond to changes in density. We present an algorithm that lets characters re-plan their paths *partially* by ignoring density information that is far away. When our algorithms are applied to a crowd, the characters will spread among multiple routes, such as in Figure 9.2b. This behavior *emerges* automatically based on the individual choices of each character.

Our concept of density-based crowd simulation can be applied to all navigation meshes. This also means that it automatically applies to both 2D environments and multi-layered environments. Throughout this chapter, we will use the Explicit Corridor Map (ECM) from Chapters 4 and 5, but other navigation meshes can be used as well. We choose the ECM because it provides a convenient mapping from densities to edge costs, and because it allows us to plug the method into our crowd simulation framework of Chapter 10.

Compared to our original publication [155], this chapter explains our method in a more general way without focusing only on the ECM. Also, we explain the method's relation to fundamental diagrams, and we re-run experiments using our updated crowd simulation software to obtain more relevant results.

The remainder of this chapter is structured as follows:

- Section 9.2 summarizes related work on crowd density, fundamental diagrams, and density-aware planning.
- Section 9.3 describes how we store and update the current crowd density in each region of a navigation mesh.



(a) Without density information (b) With density information

Figure 9.2: A crowd moving through a simple multi-layered environment. **(a)** Without density information, most of the characters follow the same short path. This looks unnatural and leads to a traffic jam. **(b)** When density information is considered, the characters will naturally spread out among the available routes.

- Section 9.4 uses this information to compute more informed cost values in the navigation graph, which induces a density-based path planning algorithm.
- Section 9.5 adapts this algorithm for re-planning paths over time.
- Section 9.6 shows that our algorithm can be used in real-time crowd simulations, and it analyzes how the algorithm affects the behavior of a crowd.
- Finally, Section 9.7 concludes the chapter and highlights our method's limitations and options for future research.

9.2 Related Work

We refer the reader to Chapter 2 for an overview of related work on path planning and crowd simulation. For this chapter, we will focus on previous research on crowd density, which has not yet been covered in Chapter 2.

9.2.1 Crowd Density and Fundamental Diagrams

Crowd density is an important topic in the field of *pedestrian dynamics*, which revolves around analyzing and simulating the behavior of pedestrians in real-world scenarios. The crowd density in an area has an impact on the safety of pedestrians in that area. As stated in Section 9.1, several crowd-related disasters in real life can be attributed to a high crowd density.

Fruin [31] has proposed a system that uses 'Levels of Service' (LoS) to indicate how safe or comfortable a certain crowd density value is. This system subdivides the range of possible crowd density values into intervals that are labelled with a letter (A to F) and a color, as summarized in Table 9.1. These labels are useful for

analyzing when and where the crowd density reaches a critical value. However, the density itself can be measured in a variety of ways, and the chosen measurement method strongly influences the outcome of such an analysis [168].

In a widely cited technical report, Weidmann [160] has shown that a pedestrian's movements are influenced by environmental factors (e.g. weather conditions or the incline of a surface) and personal factors (e.g. age or gender). The study also revealed that the typical walking speed of a person decreases as the crowd density around that person increases. These observations have influenced several simulation models [22, 58].

The relation between crowd density and typical walking speed is often summarized using a chart called a *fundamental diagram*. Fundamental diagrams were first used in car traffic studies to capture the relations between driving speed, traffic density, and flow (throughput) for roads [40]. In recent years, they have become an increasingly popular concept for pedestrian dynamics as well [130]. Based on real-world measurements, researchers have obtained fundamental diagrams for pedestrian flows in small scenarios such as straight corridors [29, 89, 167]. Therefore, many different versions of the fundamental diagram exist; examples are illustrated in Figure 9.3. The exact relation between crowd density and walking speed appears to depend on many factors, including culture and the exact layout of the environment. However, all versions of the fundamental diagram have in common that the walking speed decreases as the density increases.

The fundamental diagram can also be computed for the behavior of *simulated* crowds rather than real-world observations. Comparing the fundamental diagram of a crowd simulation to its real-world counterpart indicates how well the simulation corresponds to real behavior at a macroscopic level [11]. However, because pedestrian flows are more complex and scenario-specific than standard traffic flows, there are still many open research questions on how to use the fundamental diagram properly for such purposes.

In this chapter, we will use the concepts behind the fundamental diagram to influence the global path planning of individual characters. This will lead to a more diverse crowd flow that often avoids high-density situations.

9.2.2 Density-Based Paths and Crowds

Karamouzas et al. [73] have presented a grid-based method for density-based crowd simulation. They mark each cell in a grid as 'dense' when a character enters it, and this density value decreases gradually over time. It is shown that path planning on this grid leads to natural variety among characters. However, as described earlier, a grid is typically a less accurate and more expensive representation of the environment than a navigation mesh. Furthermore, their technique is not based on the real-world concepts of crowd density and fundamental diagrams.

Pettré et al. [120] have proposed subdividing a crowd into separate flows according to density. They used *Navigation Flow* queries to dispatch many entities

LoS	Color	Range	Description
A	Blue	< 0.2 P/m^2 $\rho \in [0..0.036\rangle$	Pedestrians can move freely. Conflicts are unlikely.
B	Green	$0.2 - 0.45$ P/m^2 $\rho \in [0.036..0.081\rangle$	Pedestrians can move freely, but they begin to notice others and might adapt their paths.
C	Yellow	$0.45 - 0.7$ P/m^2 $\rho \in [0.081..0.126\rangle$	Sufficient space for normal walking. Possible conflicts in case of bi-directional flows.
D	Orange	$0.7 - 1.1$ P/m^2 $\rho \in [0.126..0.198\rangle$	Restricted freedom of selecting an individual walking speed. Friction is very likely.
E	Red	$1.1 - 2.0$ P/m^2 $\rho \in [0.198..0.36\rangle$	Pedestrians often slow down or shuffle. Overtaking is difficult, as are crossing flows.
F	Purple	> 2.0 P/m^2 $\rho \in [0.36..1]$	Pedestrians resort to shuffling. Crossing and bi-directional flows are nearly impossible.

Table 9.1: The Level-of-Service system used by Fruin [31]. 'P/m^2' stands for 'persons per square meter'. We have translated this to a density range of $[0, 1]$ using an average pedestrian area of 0.18 m^2. The LoS descriptions have been suggested by Daamen [23].

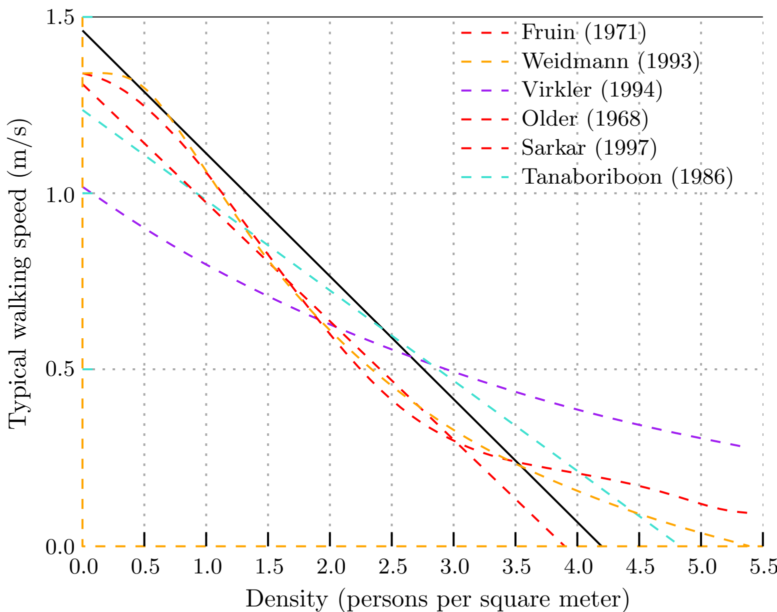


Figure 9.3: A fundamental diagram describes the relation between crowd density (typically expressed in persons per square meter) and the average walking speed of pedestrians (in meters per second). Different researchers have obtained different diagrams based on real-world observations. This is an adapted version of an image by Daamen [23].

that move between shared locations. However, in our crowd simulations, characters may have individual start and goal positions that cannot be grouped, and we would like to let each character use its own personal sensitivity to density. We will therefore use navigation meshes and individual planning.

Our density-based planning algorithm is a generalization of the *Fastest-Path Algorithm* by Höcker et al. [56]. This algorithm uses the density-speed relation to compute the *estimated traversal time* of edges in the graph. By using these values as weights during the search, the (estimated) fastest path can be obtained. Kneidl and Borrmann [84] have shown that this Fastest-Path Algorithm can lead to behavior that matches real crowds. However, this method uses a collection of squares to approximate the local density information. These squares can overlap, which causes some parts of the walkable space to be represented more than once. This leads to a bias where some parts of the walkable space are implicitly considered to be more important than other parts. Furthermore, some parts of the walkable space may not be represented at all.

By contrast, our method maps density information onto the regions of a navigation mesh, which are (usually) non-overlapping by definition. The use of navigation meshes also automatically supports multi-layered environments. We also generalize the edge cost function to let characters have individual sensitivities to density influence. Another improvement is that we address the issue of (partial) re-planning during the simulation.

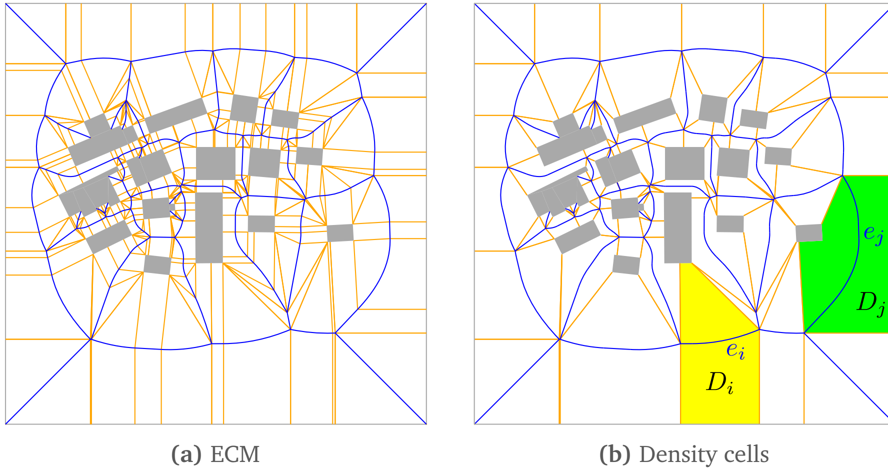
9.3 Density-Annotated Navigation Mesh

This section describes how we annotate a navigation mesh with density information. We will primarily use the Explicit Corridor Map (ECM); Section 9.3.3 will show how the concept can be applied to other navigation meshes.

9.3.1 Density Cells

Recall from Chapter 4 that the ECM is a medial axis annotated with nearest-obstacle information. Each edge in the ECM consists of two or more *bending points*, each of which is annotated with its nearest obstacle point on the left and right side. The edge therefore induces a sequence of one or more simple polygonal *ECM cells*. For this chapter, our goal is to assign a single density-based cost to each ECM edge. Therefore, we group the ECM cells of an edge e_i into a single *density cell* D_i. In Chapter 4, we have shown that an environment with n obstacle vertices yields an ECM with $\mathcal{O}(n)$ edges. Thus, there are $\mathcal{O}(n)$ density cells, each of which corresponds to a unique ECM edge. An example is shown in Figure 9.4.

During the simulation, we will keep track of the current crowd density in each density cell, which is an indication of the crowd density along the corresponding ECM edge. Section 9.4 will describe how we convert these density values to character-dependent edge traversal costs.



(a) ECM **(b)** Density cells

Figure 9.4: Example of an ECM and its density cells. **(a)** The ECM of the recurring *Military* environment. **(b)** ECM cells can be grouped per edge to obtain a set of density cells. Two examples of density cells have been colored yellow and green.

9.3.2 Maintaining Density Values

Crowd density is often expressed in persons per square meter [160]. However, to account for characters of *different sizes*, we define crowd density as the *fraction* of an area that is occupied by characters.

As usual, we model characters as disks. To simplify the computation of densities in real-time, we associate each character to a single cell, namely the cell in which its centroid lies. This approximation is justified because characters move in each simulation step and they are typically much smaller than the density cells.

Let C_i be the set of characters whose centroids lie inside density cell D_i. We compute the crowd density ρ_i of D_i as follows:

$$\rho_i = \min(1, \frac{\sum_{c \in C_i} ||c||}{||D_i||})$$

where $||S||$ denotes the area of a shape S (e.g. a disk or a polygon). Technically, the measured density could exceed 1 because we associate each character to a single cell, and because characters may overlap if local collision avoidance fails. However, we explicitly enforce a maximum density of 1 because this is intuitively the highest density that can be achieved. (If the simulation would be able to model characters getting stacked or crushed, higher densities would be possible after all.)

During the simulation, it is straight-forward to update these density values whenever a character is added, is removed, or has moved to a different density cell. In Section 9.6, we will show that this can be done without affecting the performance of the simulation.

9.3.3 Using Other Navigation Meshes

We use the ECM because of its advantages mentioned in Chapter 4: the ECM enables efficient path planning for disks of any radius, and it supports useful operations such as retractions and the computation of indicative routes with clearance. Also, there is a one-on-one correspondence between ECM edges and density cells, which makes the ECM particularly useful for this chapter.

When using a navigation mesh *other than the ECM*, the mapping from cell densities to 'edge densities' is slightly different. As mentioned in Chapter 7, most other navigation meshes are defined in terms of polygonal regions, which can be used directly as density cells. However, path planning is performed on the *dual graph* of these regions, and an edge in this graph does not correspond to a single density cell, but to a *pair* of adjacent cells. An appropriate density value for the edge would be the average of the two cells' densities, or a weighted average based on which fraction of the edge lies in which cell. We will not investigate this further because the mapping from density cells to edges is already clear in the ECM. Still, we emphasize that our method can be applied to other navigation meshes as well.

9.4 Density-Based Path Planning Algorithm

In the following sections, we use the term *path* for a sequence of edges in the graph, as in Chapter 8. A path in the ECM can be converted to an indicative route (an actual curve for a character to follow) by using the techniques from Chapter 4. In this chapter, we are only interested in the computation of paths.

Recall from Section 9.2 that a *fundamental diagram* encodes the relation between crowd density and walking speed. Assuming that the crowd density is given as a fraction, the fundamental diagram describes a function $\phi : [0, 1] \rightarrow \mathbb{R}_{\geq 0}$ that maps any density value to a non-negative typical walking speed.

Different researchers have suggested different definitions of the function ϕ, as indicated in Figure 9.3. The function may even differ between scenarios or cultures. However, all versions of ϕ have in common that the outcome decreases as the density increases. In the following sections, we will use ϕ as an abstract concept. In Section 9.6, we will define the specific function used in our experiments.

9.4.1 Edge Costs

In the ECM, the density ρ_i of a density cell D_i can be used immediately as the density along the corresponding edge e_i. Given an appropriate function ϕ, the *estimated walking speed* along an edge e_i is given by $\phi(\rho_i)$, i.e. the typical walking speed at the edge's current density according to the fundamental diagram. The *maximum walking speed* is $\phi(0)$, i.e. the speed that can be achieved at a density of zero. Let $l(e_i)$ be the arc length of the edge, in meters. We define the cost of e_i as follows:

$$cost(e_i) = t_{\min}(e_i) + w \cdot (t_{\mathrm{ex}}(e_i) - t_{\min}(e_i))$$

where $t_{\min}(e_i) = l(e_i)/\phi(0)$ is the time required to traverse the edge e_i at maximum speed, $t_{\mathrm{ex}}(e_i) = l(e_i)/\phi(\rho_i)$ is the current estimated traversal time due to the density of e_i, and w is a non-negative weight.

The weight w provides a natural means of interpolating between the shortest path and the least dense path in the graph. If $w = 0$, characters will choose the shortest path. If $w = 1$, characters will prefer the (estimated) fastest path as described by Höcker et al. [56]. As w increases further, characters will have an increasing desire to avoid dense regions. Note that it is possible for each character to have its own personal value of w, or even its own function ϕ. This allows us to model various types of behavior within a single crowd.

9.4.2 Algorithm

Planning a path from a point s to a point g is largely the same as in Chapter 4: we retract s and g onto the medial axis and use the A* algorithm to compute a path from the retraction $Retr(s)$ to the retraction $Retr(g)$. However, the algorithm is now based on (weighted) *time* instead of distance. For the cost of an edge e_i, we use $cost(e_i)$ instead of $l(e_i)$. For the heuristic function h that estimates the remaining cost for reaching the goal from a vertex V, we use the time to reach the goal via a straight line at maximum speed, i.e. $h(V) = d(V, g)/\phi(0)$. Note that this heuristic is consistent and admissible: an edge is never shorter than a straight line, and the actual density along an edge is never smaller than 0.

9.5 Re-planning

When a crowd is moving, the density values in the navigation mesh can change rapidly. This means that characters should regularly *re-plan* their global paths to respond to the latest changes in density.

In Chapter 8, we have presented ODPA*, an adaptation of the A* algorithm for re-planning an optimal path after an obstacle has been added or removed. However, this algorithm assumes that the graph costs have only changed in a limited region. We cannot use the same approach in this chapter because the crowd density may have changed arbitrarily throughout the entire environment.

Other re-planning algorithms such as D* Lite [85] do support arbitrary changes in edge costs, but (as argued in Chapter 8) they may require too much memory for crowds of many characters. Therefore, to let a character re-plan its path in the presence of changing densities, it is preferable to *compute a new path from scratch* using the standard A* algorithm.

We can, however, improve the efficiency of re-planning by letting characters use density information *only up to a certain distance*, which we call the *density viewing*

distance d_D. This allows us to re-use parts of paths that were computed without density information. The resulting paths are not necessarily optimal, but the decrease in path planning time makes this approach attractive for large crowds. Furthermore, if the crowd density changes rapidly, it is justified to ignore densities of areas that are far away: these densities are likely to have changed by the time the character re-plans again.

9.5.1 Planning with a Density Viewing Distance

We first explain how the density viewing distance d_D is used in the *regular* planning algorithm, using the same terminology as in Chapter 8. A character initially plans a path from a graph vertex S to a graph vertex G. This initial A* search uses density information as long as the path from S has a curve length of at most d_D.

More specifically, let V be any vertex that is being expanded during the search, and let $[SV]$ be the path computed so far, with curve length $l([SV])$.

- If $l([SV]) \geq d_D$, we assume zero density for all outgoing edges of V.
- Otherwise, we do use density information, but possibly only for *parts* of the outgoing edges. For each neighboring vertex W, let e_W be the edge from V to W, with curve length $l(e_W)$.
  - If $l([SV]) + l(e_W) \leq d_D$, then e_W lies entirely within d_D. We use the density of e_W normally, as described in Section 9.4.
  - Otherwise, we use the density of e_W only for the fraction f of the edge that lies within d_D, and we assume a density of zero for the remaining part:

$$t_{\text{ex}}(e_W) = f \cdot \frac{l(e_W)}{\phi(\rho_W)} + (1-f) \cdot \frac{l(e_W)}{\phi(0)}$$

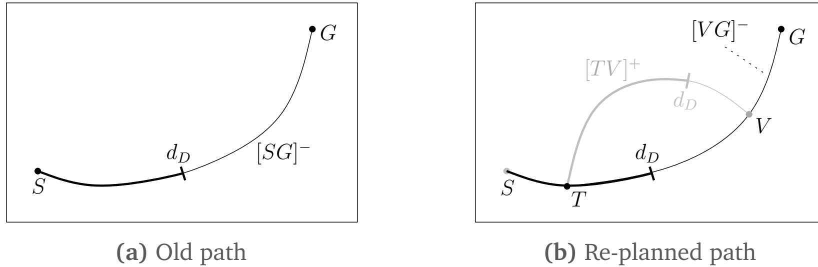
The result of this search will be a path $[SG]^-$ for which crowd density information was only used up to a curve length d_D, as shown in Figure 9.5a. Again, we use the subscript $^-$ to denote the 'old' situation before the character re-plans.

9.5.2 Partial Re-planning with a Density Viewing Distance

Now assume that the character decides to *re-plan* its path at a vertex $T \in [SG]^-$. The goal is to compute a new path $[TG]^+$. This path may overlap with the old subpath $[TG]^-$, but it can be quite different for two reasons: the crowd density may have changed, and the character can now perceive the density of other areas.

The re-planning algorithm works largely as in the previous subsection: we use density information as long as the path from T is not longer than d_D. However, a *special case* occurs when we arrive at a vertex of $[TG]^-$ that lies beyond d_D in both the old *and* the new situation. Let V be the first expanded vertex for which this holds, i.e. the first expanded vertex $V \in [TG]^-$ for which $l([SV]^-) \geq d_D$ and $l([TV]^+) \geq d_D$. This situation is shown in Figure 9.5b. Because the old subpath

$[VG]^-$ was already computed without using density information, and we cannot use any density information to compute a new subpath $[VG]^+$, we know that $[VG]^+ = [VG]^-$. That is, the optimal path from V to G remains unchanged.



(a) Old path

(b) Re-planned path

Figure 9.5: Re-planning using a limited density viewing distance d_D. **(a)** The initial path $[SG]^-$ uses density information up to a path length of d_D. **(b)** When re-planning from a vertex T, we may arrive at a vertex $V \in [SG]^-$ that lies beyond d_D in both the old and the new situation. In this case, the old subpath $[VG]^-$ is still optimal.

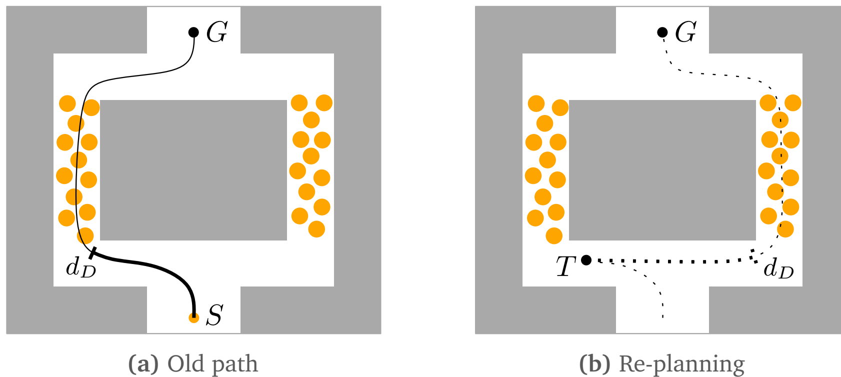
We can now halt the search and return the concatenation of $[TV]^+$ and $[VG]^-$ as the new path $[TG]^+$. Note that this path is merely the optimal path *via* V, and not necessarily the overall optimal path. If we would continue searching, we might find a longer detour that yields a better path to G. However, we choose to stop the search here because the character is already using limited information, so even an 'optimal' path is not necessarily optimal with respect to *all* density information. We conclude that partial re-planning using d_D may yield suboptimal paths in exchange for faster re-planning. This is an important trade-off when simulating large crowds.

9.5.3 Effect of the Density Viewing Distance

The density viewing distance d_D provides a trade-off between efficiency and path quality. If $d_D = 0$, then no density information is used at all, and re-planning becomes unnecessary. If $d_D = \infty$, then the character uses all available density information, but re-planning is equivalent to a standard A* search from scratch.

A risk of choosing a small (but non-zero) value for d_D is that characters can become indecisive. The example shown in Figure 9.6 features two crowded areas on the left and right side. Assume that a character has chosen to traverse the left area because it could not yet see that area's density information. If the character re-plans when this area has become visible, the character will notice a high density and plan a detour through the right area, which is currently invisible. As the character traverses this detour, the right area becomes visible and the left area is too far away again. Thus, when the character re-plans again, it will choose a new detour through the left area. This pattern will repeat itself indefinitely.

To prevent such effects, we require a more sophisticated model of a character's knowledge about the environment. However, as indicated in Chapter 8, it is currently unclear how to apply such memory models to large crowds efficiently.



(a) Old path

(b) Re-planning

Figure 9.6: A small density viewing distance d_D can make characters indecisive. **(a)** The initial path $[SG]^-$ uses the left area, but that area's density is not yet taken into account. **(b)** The re-planned path $[TG]^+$ uses the right area because the high density on the left is now visible. The character will keep switching back and forth.

We leave these potential improvements for future work. In our experiments, we will show how d_D influences the re-planning time and not the crowd's behavior.

9.6 Experiments and Results

In this section, we show how our density-based path planning algorithm performs in a crowd simulation, and we analyze the influence of the algorithm's parameters. Only one CPU core was used, unless stated otherwise. We do not yet analyze the real-time performance of the simulation; we save such experiments and discussions for Chapter 10, in which the entire crowd simulation framework and its efficiency will be treated.

9.6.1 Simulation Settings

Although our crowd simulation software allows characters to have different individual sizes, we modelled characters as disks with a radius of 0.24 m and a preferred walking speed of 1.4 m/s, in line with the observations of Weidmann [160]. We ran the simulations using a fixed simulation time step of 0.1 s. Chapter 10 will describe our simulation software in more detail.

In our original publication [155], we performed all experiments without collision avoidance. In this thesis, we do include collision avoidance because it gives a more realistic image of how our method can improve a crowd simulation. We used our own implementation of a vision-based collision-avoidance method by Moussaïd et al. [105]. We also included the *Stream* method by van Goethem et al. [36] because it allowed us to simulate higher crowd densities. Furthermore, we

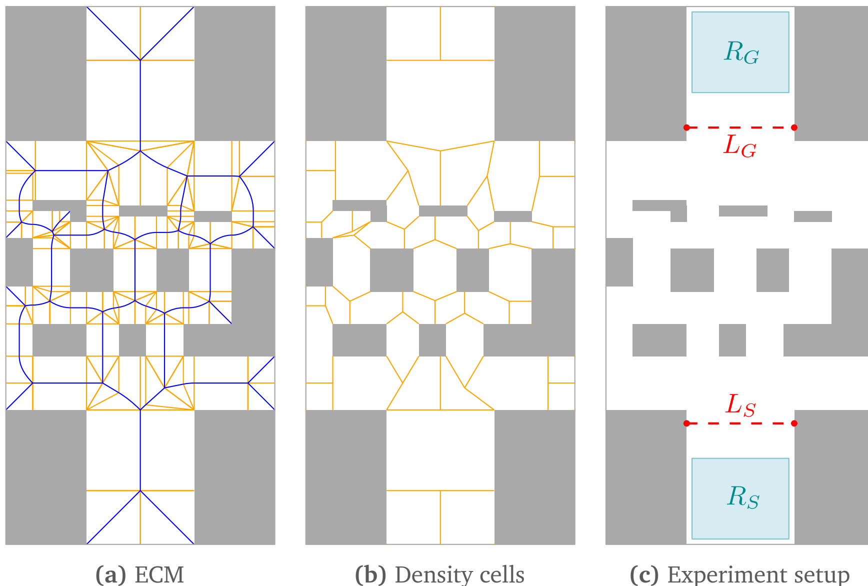
let characters use the *Indicative Route Method* [74] to follow their indicative routes smoothly. Chapter 10 will describe these components in more detail.

For the function ϕ that maps densities to typical walking speeds, we have used a simple function that decreases linearly from $\phi(0) = 1.4$ to $\phi(1) = 0$. More sophisticated functions can be implemented, but we expect that they will have similar effects.

9.6.2 Explanation of the Density Weight Experiments

Our first experiments will show how the density weight w of our path planning method affects the behavior of the crowd.

Environment. We use the *Blocks* environment shown in Figure 9.7. This is a test environment from a previous paper on crowds and densities by Karamouzas and Overmars [73]. We have added entry and exit regions that allow us to create a dense crowd flow. The *Blocks* environment measures 50×100 meters and contains 64 obstacle vertices. Its ECM consists of 40 vertices, 46 edges, and 150 bending points; it was computed in 3.5 ms.



(a) ECM **(b)** Density cells **(c)** Experiment setup

Figure 9.7: The *Blocks* environment used in our experiments. **(a)** The medial axis is shown in blue; nearest-obstacle annotations are shown in orange. **(b)** The ECM edges induce a subdividion into density cells. **(c)** The start region R_S and goal region R_G are shown in blue. The red line segments L_S and L_G denote where our measurements for a character start and stop.

In this environment, we ran simulations in which 15 or 30 characters per second (i.e. an average of 1.5 or 3 per simulation step) were added at uniformly

chosen random positions in a start region R_S at the bottom of the environment. We will use the symbol c to denote the number of characters that are spawned per second. Each character received a random goal position in a region R_G at the top. These regions are shown in Figure 9.7c. Characters computed an optimal path in the ECM, which was then converted to a short path with a preferred clearance of 5m to obstacles, using the techniques from Chapter 4. The preferred clearance was set this high because lower values caused characters to get stuck at obstacle corners too often.

Variables. We varied three simulation properties: the number of characters per second c (15 and 30), the density weight w (0, 1, 3, 5, 10, and 20), and whether or not the characters could re-plan. The results for all combinations of settings will be discussed in Sections 9.6.3 and 9.6.4.

Our method allows each character to use its own personal parameter settings, such for the density weight w. However, to show the effect of the parameter w more clearly, we will not vary this value between characters in the entire crowd.

Measurements. We measured the path length and average walking speed of each character. We performed these measurements only between the line segments L_S and L_G shown in Figure 9.7c, to focus on the areas in which characters could make decisions. For the same reason, we removed a character as soon as it crossed L_G, to prevent overcrowding in the goal region R_G.

To allow the environment to get filled with characters first, we excluded the results of characters that crossed L_S within the first 100 simulation seconds. We stopped measuring at 500 simulation seconds, and we only used the data of characters that had reached their goal by that time. In other words, if T denotes the total simulation time that has passed, we performed measurements between $T = 100$ and $T = 500$. We also kept track of the average and maximum density of each region between $T = 100$ and $T = 500$.

Visualization. We will show three figures for each combination of settings. The left figure will show the crowd at $T = 500$, just before the simulation ends. The middle figure will show the average density per region, and the right figure will show the maximum density per region.

We visualize densities using the color scale suggested by Fruin [31] and shown in Table 9.1. To convey more information, we interpolate between the colors of this scale: for example, we interpolate from green at $\rho = 0.036$ to yellow at $\rho = 0.081$. All densities above $\rho = 0.36$ are displayed in purple.

9.6.3 Density Weight Without Re-planning

We first performed the experiment *without* re-planning. Visual results are shown in Figures 9.10 to 9.13. Quantitative results are shown in Table 9.2.

We first discuss the results for a spawn rate of 15 characters per second ($c = 15$). At $w = 0$, all characters compute the same shortest path in the ECM. Many parts of the environment are never used, which results in an average and maximum density of zero in many regions. As we increase the value of w, characters gradually start using more alternative paths. However, these alternatives are explored in a wave pattern. Initially, all characters follow the shortest path. As soon as the corresponding regions get too crowded, the subsequent characters will choose a different path— until this path get too crowded again and another path becomes more attractive. Various paths take turns in being the most attractive option, but not many paths are used at the same time because characters do not re-plan.

This effect is clearly visible in Figures 9.10 and 9.11. At each point in time, the crowd is focused on particular routes (shown in the left subfigures). For higher values of w, the *average* density (shown in the middle) is spread more evenly throughout the environment, but the *maximum* densities (shown on the right) remain high because many regions are overused at some point in the simulation.

In crowded areas, some characters were pushed away by the crowd to such a degree that they could not follow their routes anymore, which caused them to get stuck behind obstacles. However, a rate of 15 characters per second is not yet high enough to lead to highly problematic traffic jams. Table 9.2 confirms this: higher values of w lead to longer paths because longer detours are used, but the average speed of the characters does not change because the crowd is never too dense.

When we increase the spawn rate to $c = 30$, the crowd becomes too dense for our simulation model in this scenario, and traffic jams occur for all values of w. The worst situation occurs at $w = 0$; increasing w does lead to more diversity, but each value of w yields its own bottlenecks at which characters get stuck. The second half of Table 9.2 shows that higher values of w still allow characters to explore longer paths. This time, higher values of w also appear to yield higher walking speeds, but the results are strongly influenced by which congestion is (coincidentally) the most unfortunate.

Admittedly, the occurrence of traffic jams depends on many other simulation settings as well, such as the type of indicative routes, the collision avoidance method, and the inclusion or exclusion of *Stream*. We have attempted to choose the best settings for this scenario, i.e. the settings at which we could test our method at the highest possible densities. Our results indicate that density-based planning alone cannot solve all problems in the crowd because many other factors are involved. In Chapter 10, we will describe our overall simulation framework, and we will discuss its limitations.

9.6.4 Density Weight With Re-planning

Next, we repeated this experiment with the additional property that each character *re-planned* its path every 5 seconds. This reduced the wave patterns and resolved

the congestions for $c = 30$. Figures 9.14 to 9.17 show crowds and densities; Table 9.3 provides quantitative results.

For $c = 15$, increasing w causes characters to explore more alternative routes, just like in the previous experiment. With re-planning included, we now also see that more different routes are used at the same time. (Even at $w = 0$, some characters use an alternative path if they happen to be near the left side of the first obstacle when they re-plan.) This results in a better distribution of average *and* maximum densities. Compared to the experiment *without* re-planning, we also obtain lower traversal times and higher speeds. After all, the characters can now repeatedly update their paths based on the most recent density information.

The results for $c = 30$ are similar to those for $c = 15$, but the improvement caused by re-planning is more clearly visible. While the crowd still gets congested at $w = 0$, setting w to 1 immediately resolves this problem. Table 9.3 reflects this: the average walking speed is much higher at $w = 1$, and many more characters reach their goal in time. Increasing w further leads to longer paths and higher walking speeds, as expected.

These results show that the combination of density-based planning and re-planning is a powerful tool: it creates a diverse crowd flow, *and* it allows characters to use the environment more efficiently.

Even with re-planning enabled, though, we can still observe clusters of characters that follow the same route at the same time. This is partly due to the *Stream* method that lets each character adapt its speed and direction to its neighboring characters. Preliminary experiments have shown that disabling *Stream* reduces the effect but causes local collision-avoidance problems at $c = 30$. Another solution is to assign different values of w to different characters, such that characters in the same area at the same time are less likely to make the same decisions.

9.6.5 Efficiency of Partial Re-planning

In our last experiment, we tested the performance of our re-planning algorithm from Section 9.5 using various values of the density viewing distance d_D. We only look at the *running time* of the algorithm and not at the behavior of the crowd.

For this experiment, we used the *City* environment from previous chapters because it is more complex than the *Blocks* environment. We simultaneously added $10{,}000$ characters to the environment, each with a (uniformly sampled) random start and goal position. We gave each character a density weight $w = 10$ so that many different paths would be explored, and we let each character re-plan its path every 10 seconds, using the adapted version of A* from Section 9.5. Whenever a character reached its goal, it received a new goal at a new random position, and its re-planning timer was reset.

All characters in the crowd used the same density viewing distance. We performed this experiment multiple times with the following values of d_D: 0.1,

c	w	#Finished characters	Average time (s)	Average distance (m)	Average speed (m/s)
15	**0**	4320	111.71 [3.79]	116.51 [0.70]	1.04 [0.03]
	1	4270	117.34 [15.40]	119.61 [3.76]	1.03 [0.10]
	3	4273	117.53 [12.95]	122.11 [4.80]	1.05 [0.10]
	5	4278	119.77 [18.06]	122.50 [4.91]	1.04 [0.11]
	10	4179	124.42 [26.98]	124.54 [7.94]	1.03 [0.14]
	20	4256	121.34 [13.5]	126.83 [9.49]	1.05 [0.10]
30	**0**	3000	199.39 [54.75]	122.39 [2.96]	0.66 [0.17]
	1	4984	168.22 [46.51]	124.90 [5.42]	0.79 [0.20]
	3	3050	142.72 [31.00]	124.25 [6.12]	0.91 [0.18]
	5	4317	150.24 [35.23]	127.03 [10.69]	0.88 [0.18]
	10	4647	146.35 [27.75]	128.38 [12.88]	0.90 [0.16]
	20	5399	143.05 [26.87]	132.04 [15.33]	0.94 [0.13]

Table 9.2: Results of the density weight experiment *without* replanning. Each row corresponds to a combination of settings for c (the number of characters that were added per second) and w (the density weight). The third column denotes the number of characters that have reached their goal during the measurements. The remaining columns contain statistics about the traversed paths, averaged over all characters that have reached their goal. Standard deviations are shown between square brackets.

c	w	#Finished characters	Average time (s)	Average distance (m)	Average speed (m/s)
15	**0**	4441	104.52 [3.44]	115.90 [0.62]	1.11 [0.04]
	1	4488	101.61 [2.54]	117.35 [2.19]	1.16 [0.03]
	3	4485	102.40 [3.55]	119.75 [3.39]	1.17 [0.03]
	5	4410	106.09 [5.19]	123.35 [5.91]	1.16 [0.04]
	10	4323	109.99 [7.57]	128.91 [10.16]	1.17 [0.04]
	20	4235	113.94 [10.18]	134.11 [12.59]	1.18 [0.04]
30	**0**	4767	193.09 [43.95]	123.15 [2.78]	0.67 [0.16]
	1	8457	117.04 [8.06]	120.00 [3.05]	1.03 [0.08]
	3	8404	118.10 [8.05]	126.24 [8.84]	1.07 [0.07]
	5	8338	120.47 [9.27]	130.71 [11.15]	1.09 [0.06]
	10	8326	122.85 [11.33]	134.20 [12.63]	1.09 [0.05]
	20	8329	124.48 [12.60]	136.20 [13.44]	1.10 [0.05]

Table 9.3: Results of the density weight experiment *with* replanning.

100, 200, 350, 500, and 100,000 meters. (The highest value is essentially infinite in the *City* environment). All other character settings remained the same as in the previous experiments.

We ran the simulation for 50 seconds to ensure that many re-planning actions were performed. For each re-planning action, we measured the time spent on re-planning the path based on the previous path.

Figure 9.8 shows the re-planning times for particular values of d_D. These running times comprise the entire process of obtaining a path as a sequence of ECM vertices. This includes the time spent on copying parts of the old path into the new path.

With $d_D = 100{,}000$ m (Figure 9.8a), the paths are essentially re-planned from scratch, and the characters can use all available density information. While outliers exist, most of the re-planning queries took under 0.5 ms. The most complex paths (of 60 or more vertices) took between 0.4 ms and 0.5 ms to compute on average. Note that lower path complexities occurred more often.

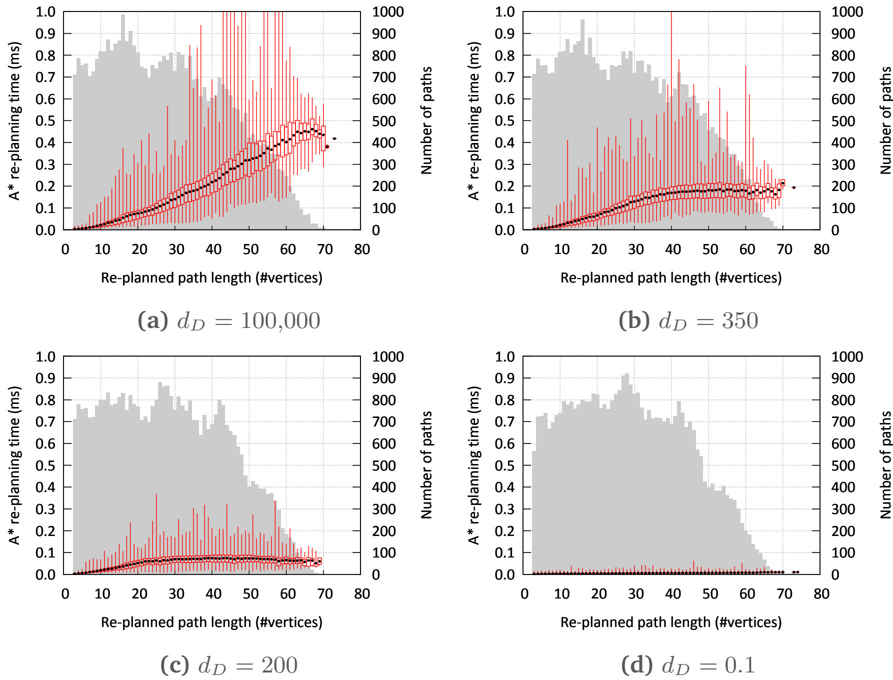
Decreasing d_D to 350 m (Figure 9.8b) or 200 m (Figure 9.8c) decreases the re-planning time for complex paths in particular. This is logical because paths of many vertices are typically longer, which means that they exceed d_D more quickly. At $d_D = 200$ m, the average re-planning time was always below 0.1 ms, and they remained almost constant for paths of 30 or more vertices. Finally, at $d_D = 0.1$ m (Figure 9.8d), the search practically terminates immediately, and re-planning is reduced to simply copying a path.

Figure 9.9a compares the A* times for each value of d_D in a single diagram. For simplicity, we have averaged the running times for each distinct path length. Therefore, the curves in this figure correspond to the black dots in Figure 9.8.

This figure clearly shows how lower values of d_D lead to faster re-planning, especially for paths with many vertices. On the other hand, a lower value of d_D means that characters use less density information. Thus, the viewing distance can improve re-planning times in exchange for a loss of information.

Finally, Figure 9.9b compares the *total* re-planning times. As in Section 8.6, the total re-planning time includes the time for A* *and* the time for computing a short indicative route with clearance. Because this indicative route is always recomputed from scratch, the differences between values of d_D are less apparent. At $d = 0.1$, the running time scales roughly linearly with the path length. The results can be improved further by repairing the indicative routes only partly. We leave this improvement for future work.

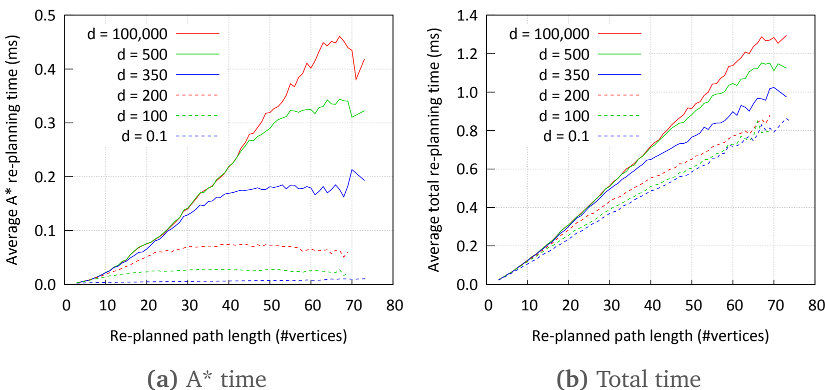
In summary, the efficiency of re-planning can be improved by lowering the density viewing distance d_D. This gives characters limited density information, but it may be useful if the application at hand requires higher performance. However, in this particular environment, many paths can already be re-planned within a millisecond even without using d_D. This is most likely sufficiently fast for many real-time applications with large crowds. In Chapter 10, we will analyze how many



(a) $d_D = 100{,}000$ **(b)** $d_D = 350$

(c) $d_D = 200$ **(d)** $d_D = 0.1$

Figure 9.8: Running times for re-planning paths using various values of the density viewing distance d_D. The horizontal axis denotes the number of ECM vertices on the re-planned path. The left vertical axis denotes the running time (in milliseconds) of A*. The gray histogram and the right vertical axis indicate how often each path length occurred. For each path length, the average planning time is shown as a black dot, and the full range of planning times is shown as a red box plot.



(a) A* time **(b)** Total time

Figure 9.9: Comparison of the average re-planning times for different values of d_D. Times have been averaged for each distinct path length. **(a)** The time spent on recomputing a path of ECM vertices. **(b)** The total re-planning time, including the time spent on recomputing an indicative route from scratch.

characters can be simulated in real-time in a range of environments, combined with re-planning, collision avoidance, and all other simulation components.

9.7 Conclusions and Future Work

In crowd simulations, it is common to let characters compute short paths to their goals. However, at high densities, popular areas of the environment can become congested while other areas are underutilized. In this chapter, we have shown how to annotate a 2D or multi-layered navigation mesh with *crowd density* information. Our density-based path planning algorithm converts this density information to edge costs in the navigation graph, based on a real-world relation between density and speed (known as the *fundamental diagram*). This algorithm allows characters to prefer detours around crowded areas. Each character can have its own sensitivity to density-based delays. These concepts can be applied to any navigation mesh.

Characters need to re-plan their paths regularly to respond to the most recent changes in density. The efficiency of re-planning can be improved by giving characters a limited distance along which they perceive density information.

We have implemented density-based crowd simulation based on the Explicit Corridor Map (ECM) from Chapters 4 and 5. Our experiments suggest that density-based path planning and re-planning can automatically divide the crowd over multiple routes. This leads to a more efficient and realistic-looking crowd flow. Furthermore, this behavior is *emergent*: it is based only on the individual decisions of each character. Crowd densities in the navigation mesh can be maintained efficiently, and the planning algorithm can be used in simulations of tens of thousands of characters in real-time.

Discussion. A limitation of our approach is that density information is summarized per region of the navigation mesh. Our planning algorithm will not recognize a crowded sub-area within a large region, such as a cluster of characters in the middle of a long or wide cell. A related problem is that global and local planning are still detached in our simulation model. If a group of characters is blocking the way without contributing sufficiently to the crowd density, it may be better to model this group as a dynamic obstacle using the techniques from Chapters 6 and 8.

We would like to obtain more insight in the effects of our simulation's parameters, including the density weight w and the density viewing distance d_D. While the meaning of the weight w is intuitive, it is not yet clear which value achieves the best behavior in a particular environment. We have also noticed that many other simulation aspects (e.g. the collision-avoidance method, the types of indicative routes, or the presence of the Streams method) affect the overall results, and that it is difficult to choose the appropriate settings for a particular scenario.

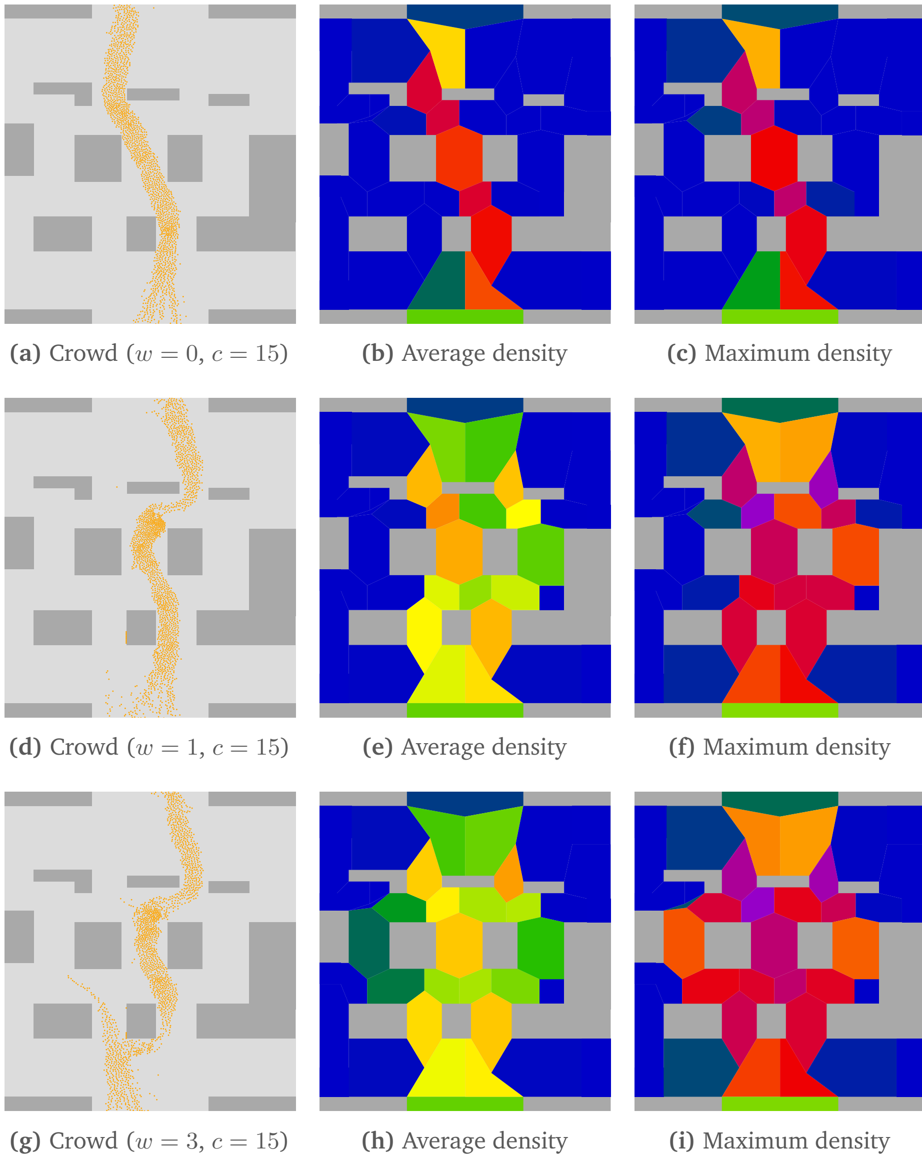
Thus, we do not claim that our density-based path planning method can solve all potential density-related problems in a crowd simulation. It should rather be

seen as a component that can be plugged into the simulation (without strongly affecting the overall computation time) to make the crowd more diverse, visually pleasing, and possibly more efficient when using the appropriate parameter settings for the scenario at hand.

Future work. Next to using density information, we are also interested in looking at the *flow direction* of a crowd. For instance, a character may prefer paths along which not too many characters are moving in the opposite direction. Experimental research [89, 167] has shown that bidirectional pedestrian flows have different fundamental diagrams than unidirectional flows, but that the difference in walking speed is significant only at higher densities. We therefore expect that flow information will play an important role in high-density scenarios such as crowd evacuation.

We have shown that re-planning is necessary to obtain a convincing crowd flow. It would be interesting to trigger re-planning actions based on specific *events* rather than simply re-planning paths periodically. An example of such an event would be the discovery of a high-density region in a character's vicinity. This would require a *memory model* to represent a character's knowledge of density information. This model could be updated based on visibility, such that a character learns about the density of a region when it becomes visible. Such a model could also prevent characters from being too indecisive, as noted in Section 9.5.3. Just like for the re-planning algorithm of Chapter 8, the main challenge is to find a model that is sufficiently sophisticated while still allowing real-time simulations of large crowds.

Finally, we have used real-world concepts to improve global planning, but we have yet to discover whether the resulting behavior is actually similar to that of real crowds. Comparing simulations to real-world data (e.g. based on fundamental diagrams [11]) is a research topic in development that poses many challenges.

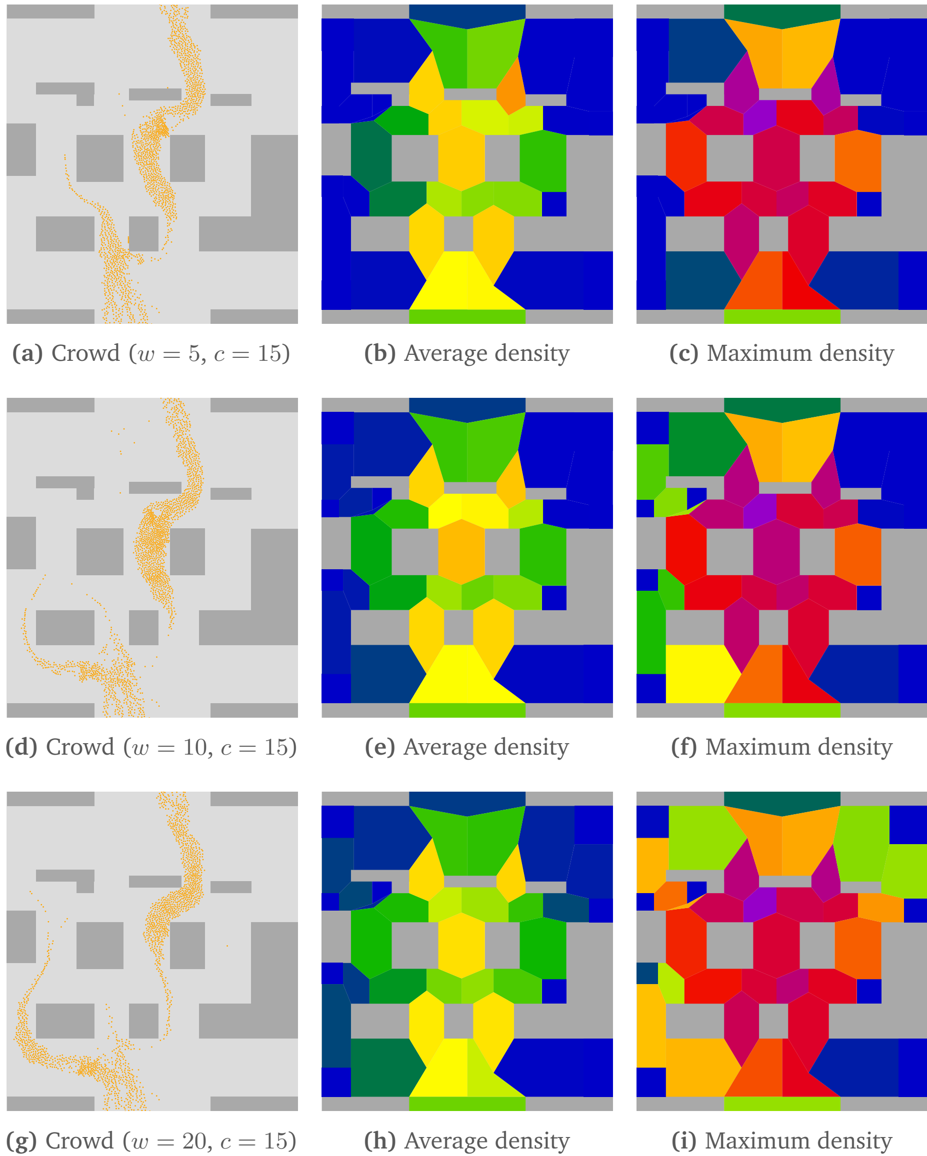


(a) Crowd ($w = 0$, $c = 15$) **(b)** Average density **(c)** Maximum density

(d) Crowd ($w = 1$, $c = 15$) **(e)** Average density **(f)** Maximum density

(g) Crowd ($w = 3$, $c = 15$) **(h)** Average density **(i)** Maximum density

Figure 9.10: Results of the density weight experiment from Section 9.6.3 for $w = 0$, $w = 1$, and $w = 3$, with 15 characters per second, and *without* re-planning.

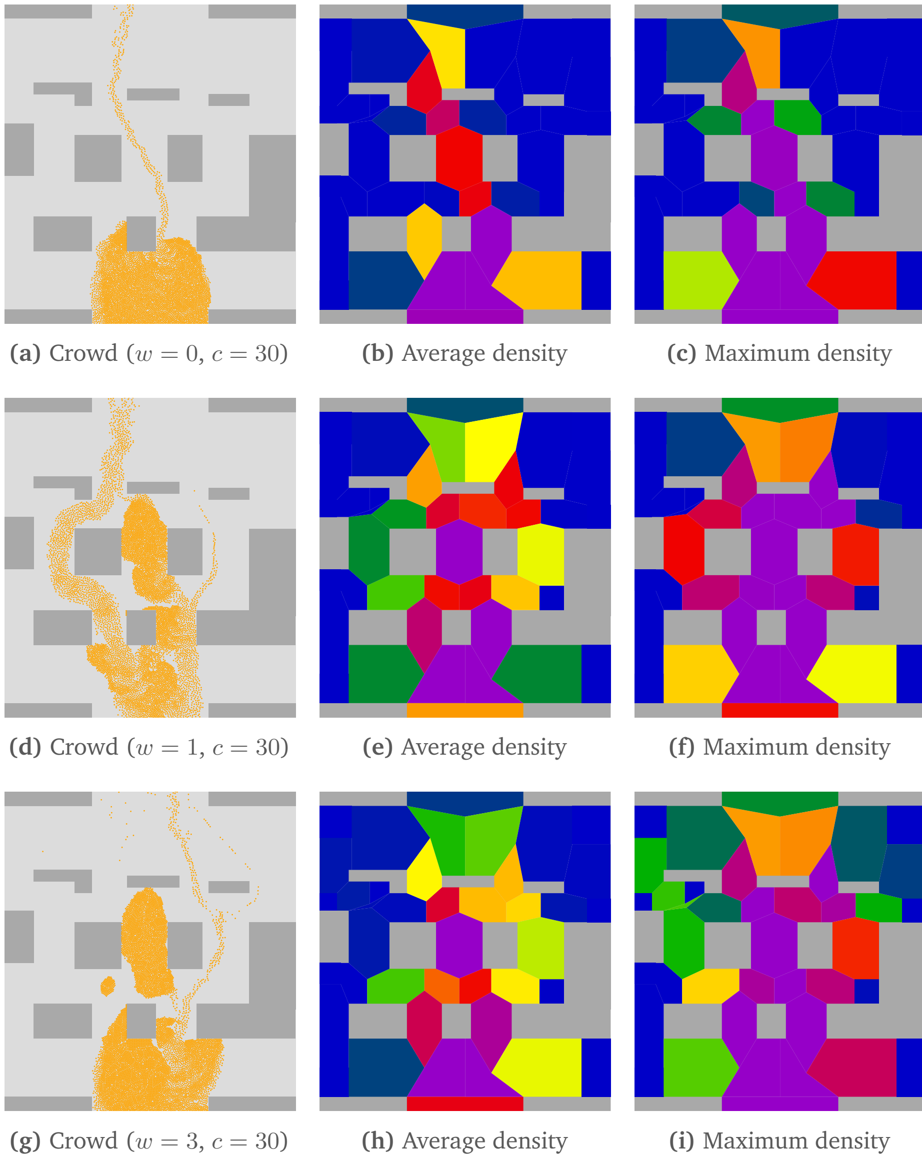


(a) Crowd ($w = 5$, $c = 15$) **(b)** Average density **(c)** Maximum density

(d) Crowd ($w = 10$, $c = 15$) **(e)** Average density **(f)** Maximum density

(g) Crowd ($w = 20$, $c = 15$) **(h)** Average density **(i)** Maximum density

Figure 9.11: Results of the density weight experiment from Section 9.6.3 for $w = 5$, $w = 10$, and $w = 20$, with 15 characters per second, and *without* re-planning.

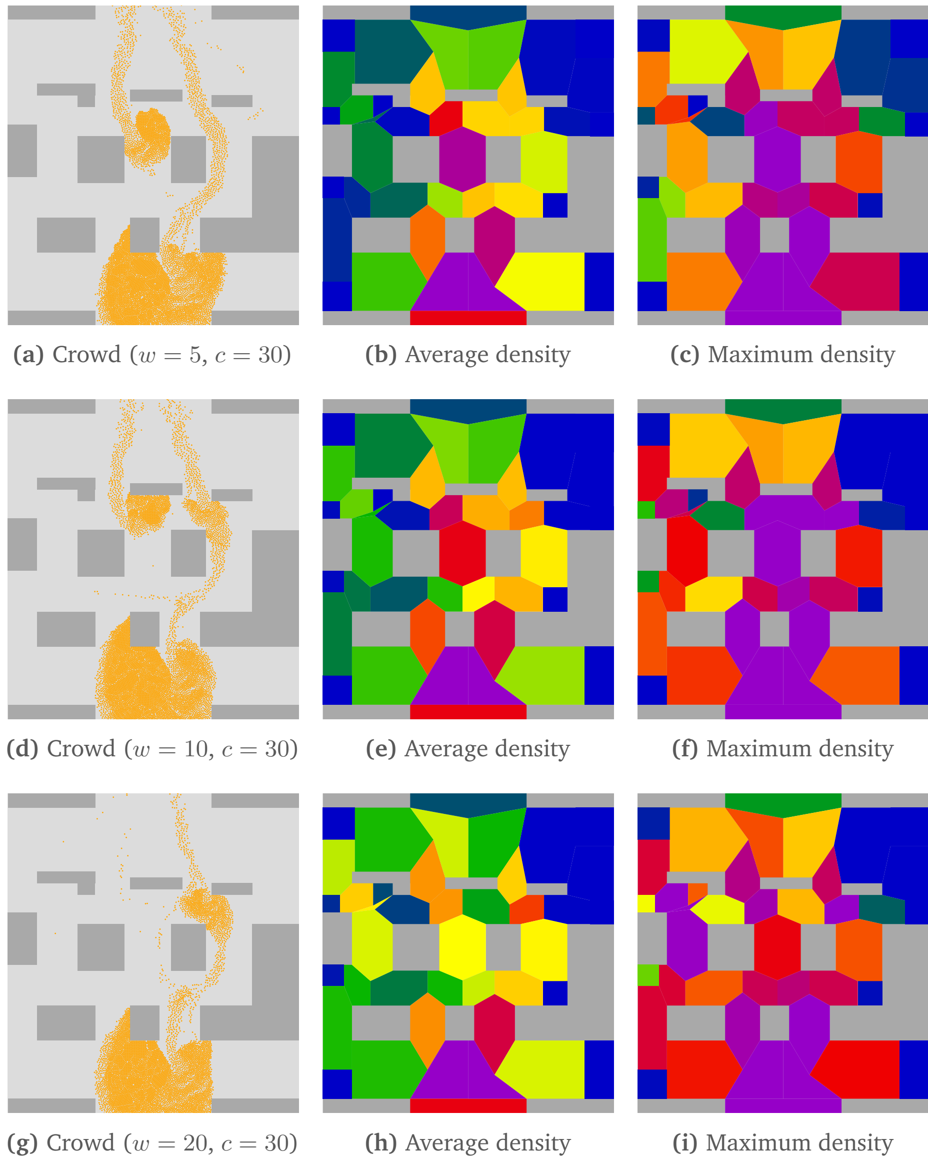


(a) Crowd ($w = 0$, $c = 30$) **(b)** Average density **(c)** Maximum density

(d) Crowd ($w = 1$, $c = 30$) **(e)** Average density **(f)** Maximum density

(g) Crowd ($w = 3$, $c = 30$) **(h)** Average density **(i)** Maximum density

Figure 9.12: Results of the density weight experiment from Section 9.6.3 for $w = 0$, $w = 1$, and $w = 3$, with 30 characters per second, and *without* re-planning.

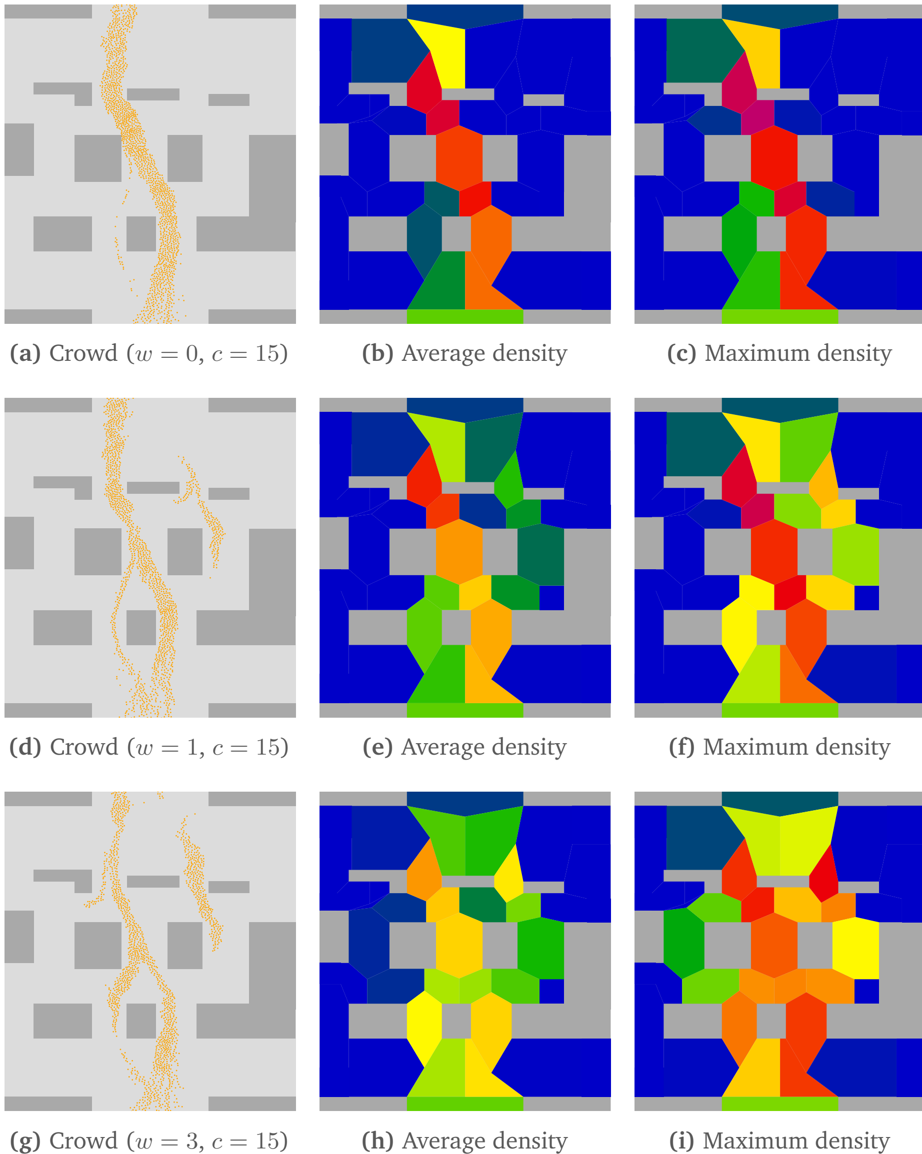


(a) Crowd ($w = 5$, $c = 30$) (b) Average density (c) Maximum density

(d) Crowd ($w = 10$, $c = 30$) (e) Average density (f) Maximum density

(g) Crowd ($w = 20$, $c = 30$) (h) Average density (i) Maximum density

Figure 9.13: Results of the density weight experiment from Section 9.6.3 for $w = 5$, $w = 10$, and $w = 20$, with 30 characters per second, and *without* re-planning.

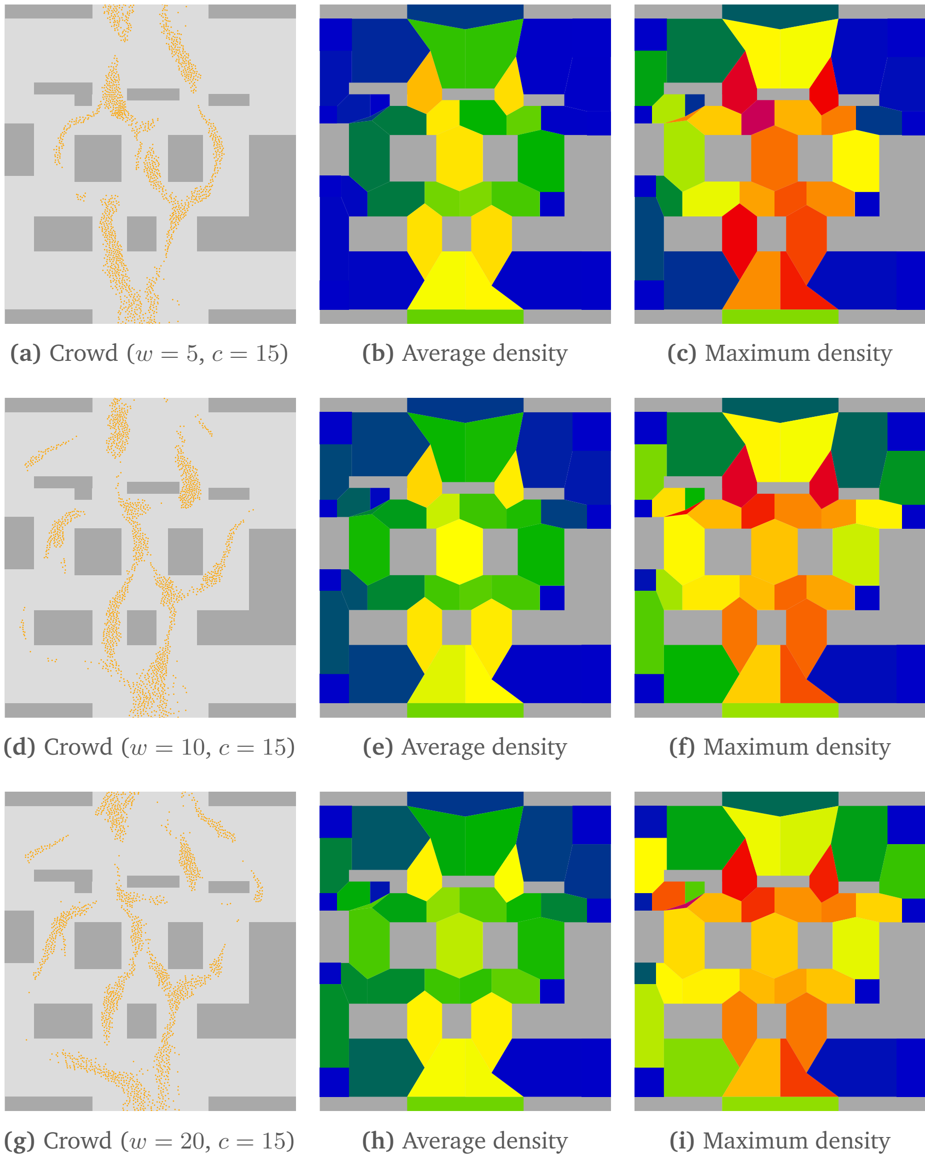


(a) Crowd ($w = 0$, $c = 15$) (b) Average density (c) Maximum density

(d) Crowd ($w = 1$, $c = 15$) (e) Average density (f) Maximum density

(g) Crowd ($w = 3$, $c = 15$) (h) Average density (i) Maximum density

Figure 9.14: Results of the density weight experiment from Section 9.6.4 for $w = 0$, $w = 1$, and $w = 3$, with 15 characters per second, and *with* re-planning.

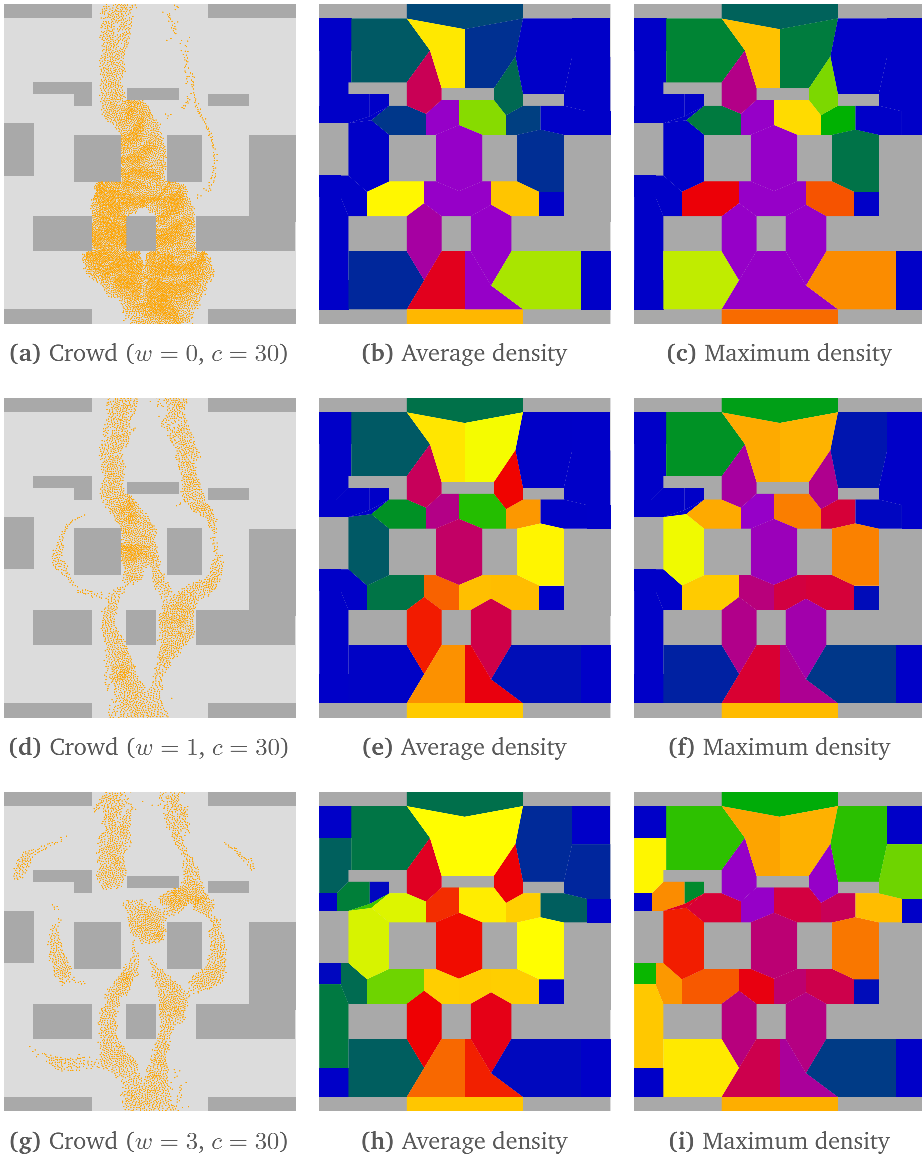


(a) Crowd ($w = 5$, $c = 15$) (b) Average density (c) Maximum density

(d) Crowd ($w = 10$, $c = 15$) (e) Average density (f) Maximum density

(g) Crowd ($w = 20$, $c = 15$) (h) Average density (i) Maximum density

Figure 9.15: Results of the density weight experiment from Section 9.6.4 for $w = 5$, $w = 10$, and $w = 20$, with 15 characters per second, and *with* re-planning.

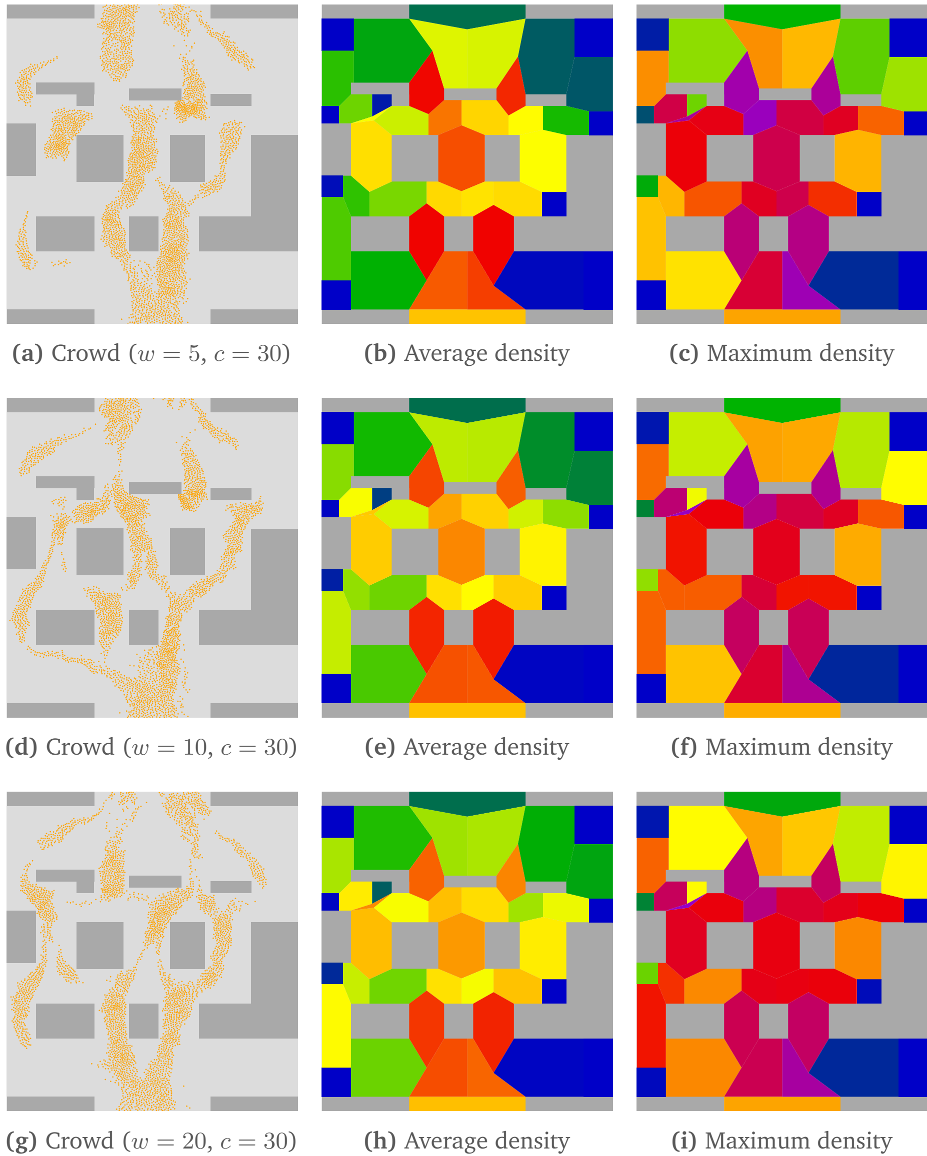


(a) Crowd ($w = 0$, $c = 30$) **(b)** Average density **(c)** Maximum density

(d) Crowd ($w = 1$, $c = 30$) **(e)** Average density **(f)** Maximum density

(g) Crowd ($w = 3$, $c = 30$) **(h)** Average density **(i)** Maximum density

Figure 9.16: Results of the density weight experiment from Section 9.6.4 for $w = 0$, $w = 1$, and $w = 3$, with 30 characters per second, and *with* re-planning.



(a) Crowd ($w = 5$, $c = 30$) (b) Average density (c) Maximum density

(d) Crowd ($w = 10$, $c = 30$) (e) Average density (f) Maximum density

(g) Crowd ($w = 20$, $c = 30$) (h) Average density (i) Maximum density

Figure 9.17: Results of the density weight experiment from Section 9.6.4 for $w = 5$, $w = 10$, and $w = 20$, with 30 characters per second, and *with* re-planning.

10 A Generic Crowd Simulation Framework

In this final chapter before the conclusion of this thesis, we propose a *multi-level framework* to generically describe crowd simulation systems, and we present our own implementation of this framework based on the ECM navigation mesh.

This chapter is based on the following publications:

- N.S. Jaklin, W.G. 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*, pages 11–14, 2013. [64]
- W. van Toll, N. Jaklin, and R. Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN / ICT.OPEN (ASCI track)*, 2015. [151]
- W. van Toll, N. Jaklin, and R. Geraerts. A generic multi-level framework for agent navigation, 2015. Poster at the 8th ACM SIGGRAPH International Conference on Motion in Games. [150]

10.1 Introduction

In Chapter 1 of this thesis, we have explained that path planning and crowd simulation are increasingly important problems in both computer games and serious gaming applications such as crowd management, evacuation studies, and safety training. In a crowd simulation, virtual characters need to autonomously find and traverse paths through a virtual environment. Characters should act in a realistic manner: their trajectories must be short and smooth, there should not be any collisions between characters, and (in many applications) the characters are expected to mimic human behavior. Furthermore, the simulation should be efficient even if the crowd is very large or dense.

In short, characters in the simulation need to perform multiple tasks that reach beyond a simple path planning algorithm. A crowd simulation system therefore requires multiple *levels* of planning, for which a *navigation mesh* is a useful representation of the environment. We have briefly touched upon this in Chapters 1 and 2 when we explained which subtasks are involved in a crowd simulation.

In this chapter, we describe this idea in more detail: we propose a generic *five-level hierarchy* for solving character navigation problems. We also present our own

algorithms and implementations of the three center levels, which concern the *geometric* aspects of path planning and crowd simulation.

Our crowd simulation software uses the ECM navigation mesh from Part II of this thesis, and we have used it for the experiments in many chapters of Parts II and III. The software is *modular* (algorithms can be combined freely to obtain various types of behavior) and *extensible* (new components can be added easily). We describe the architecture of this software, we show that it can simulate large crowds in real-time, and we provide various examples of how the software has been used in practice.

Compared to our original publications [64, 151], we omit discussions of the ECM because these have already been included in Chapters 4 and 5. We also provide more implementation details and a more thorough analysis of the software's performance.

The remainder of this chapter is structured as follows:

- Section 10.2 summarizes related work on crowd simulation frameworks.
- Section 10.3 describes our multi-level planning hierarchy and highlights the most important related work per level.
- Section 10.4 discusses the implementation of our crowd simulation software and explains how it was made to be modular, extensible, and efficient.
- In Section 10.5, we show that our software can simulate large crowds of heterogeneous characters in real-time, and we show how the software has been used in practice to perform simulations of real-world scenarios.
- Finally, Section 10.6 concludes the chapter and outlines future work.

10.2 Related Work

We refer the reader to Chapter 2 for general related work on crowd simulation and navigation meshes. For this chapter, we discuss various crowd simulation *frameworks* and programs, and we explain how they differ from our work.

Several commercial software frameworks exist for crowd simulation and analysis in serious applications. Examples of such frameworks include Legion [96], MassMotion [109], Pedestrian Dynamics [60], SimWalk [134], STEPS [138], and VisWalk [125]. One of these uses our ECM software as a 'black box' for the geometric aspects of the simulation, combined their own event-based system [60]. Most other simulation frameworks do not automatically compute an efficient navigation mesh; thus, they require more manual work.

In the entertainment industry, the Unity3D game engine has recently adopted the Recast software for the automatic generation of navigation meshes. We have

used the stand-alone version of Recast in our comparative study of Chapter 7. Golaem Crowd [37] started out as a similar package, but it has shifted its focus to high-quality plug-and-play crowds for entertainment applications. Massive [101] is a well-known software package that has been used for generating crowd behavior in many movies and games.

In the research community, SteerSuite [135] is used for simulating and evaluating algorithms for e.g. collision avoidance between characters. ADAPT [133] is a platform for developing agent behavior with an emphasis on animation. SimPed [22] and NOMAD [58] are models for passenger flows, based on real-world observations. They focus mostly on detailed behavioral models for each character. MomenTUMv2 [82] is a framework that focuses more on modeling of scenarios. The work closest to our own is Menge [21], a system in development that uses a multi-level hierarchy similar to ours. Menge has been combined with the theory of fundamental diagrams for the purpose of validation [11].

Our work differs in that it presents a more generic solution for the *geometric* aspects of path planning and crowd simulation. In terms of theory, we treat *route following* as a separate level for better flexibility. In terms of implementation, our ECM navigation mesh has many advantages, such as a small memory footprint, fast query times, independence of agent sizes, and support for dynamic environments. Also, we include specialized algorithms for route planning and route following in *weighted regions*, which we will explain further in Section 10.3.

10.3 Multi-Level Planning Hierarchy

We propose a generic framework for crowd simulation systems. The framework consists of five levels stacked in a 'hierarchy'. Figure 10.1 shows an overview of this hierarchy. Using a single character as an example, the levels can be summarized as follows:

- *High-level planning* uses AI techniques to translate a semantic action (e.g. 'go home') to one or more geometric queries ('find a path from position s to position g').
- *Global planning* computes an *indicative route*, i.e. a path from s to g that should be roughly followed.
- The following two levels update the character in every step of the simulation loop. *Path following* lets the character choose a *preferred velocity* such that it follows the indicative route. Note that a velocity is a 2D vector that encodes both *speed* and *direction*.
- *Local movement* uses the preferred velocity to compute a *new velocity* that deals with local situations, e.g. to prevent collisions with other characters or to maintain coherence in a social group of characters. The simulation then applies this velocity through time integration.

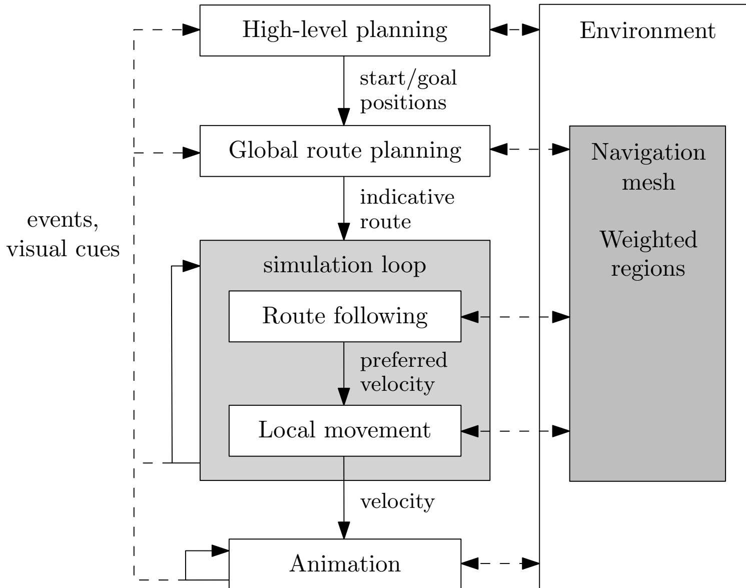


Figure 10.1: A five-level hierarchy for path planning and crowd simulation. Our research focuses on the three center levels, which concern the *geometric* aspects of planning. The environment can be represented by a *navigation mesh*, and it might contain *weighted regions*.

- Finally, *animation* handles the visual movement of the character, down to its 3D skeleton representation. The animation is typically updated at a higher framerate than the simulation itself.

10.3.1 High-Level Planning

At the top of the hierarchy, high-level planning translates the desired *semantic* behavior of a character to *geometric* path planning problems. First, a character'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. Each item of this list is of the form 'go from position A to position B and (possibly) perform an action at position B'. Of course, to let characters create such a plan automatically, the environment must be annotated with semantic data. In this example, such annotations would be the location of the train station in the environment, and the locations of ticket machines within the station.

High-level planning is a research topic of its own, involving artificial intelligence (AI) techniques such as STRIPS [28] and Hierarchical Task Networks [80]. Cognitive decision-making models have also been applied to a number of crowd simulations [116, 132, 165]. Throughout this thesis, we have focused on *geometric* planning (i.e. the center three levels of the hierarchy in Figure 10.1)

and not on these high-level tasks. Our ECM-based simulation software provides solutions for the geometric aspects of path planning and crowd simulation. It can be plugged into any other system that performs AI-related tasks, as long as this system produces specific start and goal positions for a character.

10.3.2 Global Route Planning

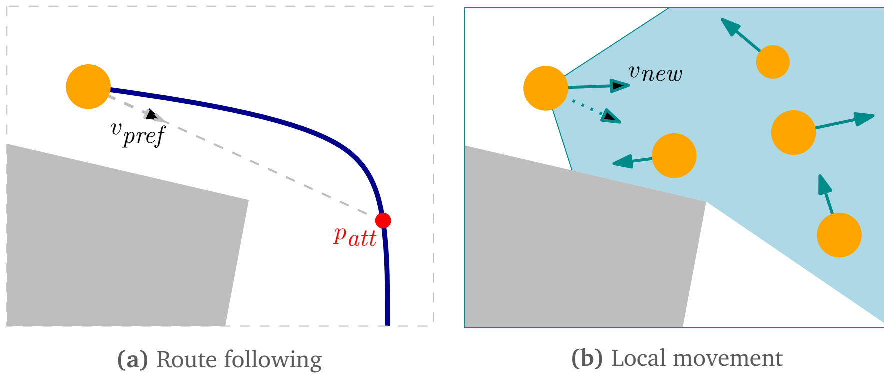
Next, global route planning uses the character's current position and goal position to compute a geometric route through the environment. In line with the rest of this thesis, we refer to the result as an *indicative route* because it is a preliminary indication of how the character should move. Having an indicative route that is followed roughly (instead of a path that is followed exactly) yields greater flexibility in the lower levels of the hierarchy.

An indicative route can be any curve through the free space. In practice, it is often a piecewise linear curve given by a sequence of bending points. Chapter 4 has explained how to compute indicative routes in the ECM: we first perform an A* search on the medial axis to obtain a sequence of ECM edges, from which we can then obtain various types of indicative routes, such as a route that stays on the left or right side of the walkable space, or a short route that keeps a preferred distance to obstacles.

As explained in Chapter 3, it is common to let the A* search algorithm compute a *shortest* path through the graph, but other options are possible. For instance, Chapter 9 has used crowd density information to compute (expected) *fastest* paths, and Geraerts and Schager have used *visibility* information to plan stealthy routes along which a character is not seen by others [34]. Linear programming techniques can solve global route planning queries for multiple characters in the same crowd, which simulates global coordination between groups of characters [75].

Another option is that the environment contains *weighted regions*. These are regions annotated with particular surface types (such as 'road' or 'grass') or psychological properties (such as 'attractive' or 'unsafe') that characters can translate to a personalized weight or traversal cost. For instance, this can be used to model characters that prefer to stay on the sidewalk, but that may cross the road or move through muddy terrain if required. Planning optimal global routes in such environments is computationally difficult, but provably good approximating techniques exist [63]. Figure 10.3 shows an example.

Navigation in weighted regions has proven to be considerably more complex than navigation in environments that are simply subdivided into 'free space' and 'obstacle space'. These are generally treated as two separate problems that need to be solved with separate data structures and algorithms. We refer interested readers to the PhD thesis of Jaklin [61] which provides various solutions for character navigation in weighted regions. Some of this work is also included in our ECM-based crowd simulation framework (Section 10.4), but our representations of the free space (i.e. the navigation mesh) and the weighted regions are detached.



(a) Route following

(b) Local movement

Figure 10.2: **(a)** At the route following level, a character (shown in orange) computes an attraction point p_{att} (red) on its indicative route (dark blue). This leads to a preferred velocity v_{pref} (black). **(b)** At the local movement level, a character computes a velocity v_{new} (shown in black) that is close to v_{pref} (dotted black) and that avoids collisions with neighboring characters.

10.3.3 Route Following

Route following ensures that a character follows its indicative route π_{ind} smoothly during the simulation. The goal of this level is to compute a *preferred velocity* v_{pref} for the character in *each time step* of the simulation.

Many researchers and frameworks do not treat route following as a separate level. However, we believe that route following is crucial for the following reasons:

- The indicative route is generally not smooth. One could smoothen it beforehand, but route following lets a character follow a smooth version of its route on the fly, based on its current position in the environment.
- Algorithms for collision avoidance (as described in the next subsection) typically require a preferred velocity as input. Unless the character can walk towards its goal in a straight line, we need an algorithm that can compute an appropriate 'sub-goal' and a corresponding walking direction at any moment during the simulation.
- Due to the presence of other characters, a character is often not located exactly on π_{ind}. A route following algorithm can define how the character should gradually move back onto the desired route.
- The virtual environment may contain *weighted regions* that are less or more attractive to traverse. A route following algorithm can take this into account, e.g. to let a character cut corners based on its personal preferences.

Two recent algorithms for route following are based on *attraction points*: in each simulation step, they choose an attraction point p_{att} on the indicative route towards

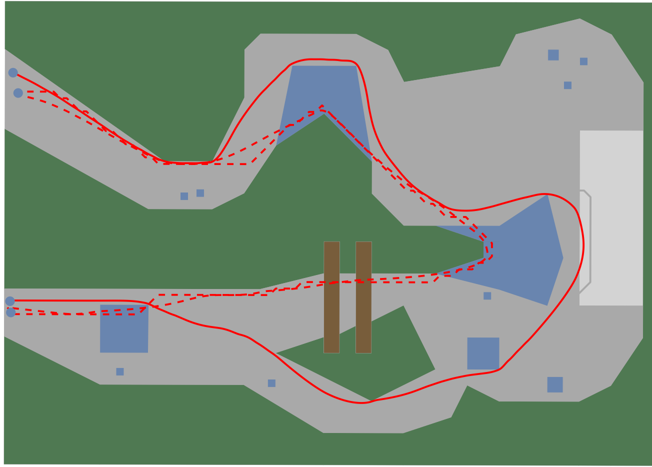


Figure 10.3: Route planning and route following in weighted regions. Each character can use its own personal region costs. The environment is a forest (green) with tree obstacles (black), puddles (blue), fallen trees (brown), and a spot with a panoramic view (light gray). For two characters (adult and child), we compute an indicative route (solid and dashed black, respectively) by using a grid that uses personalized region weights. The smoothed paths (solid and dashed red) are computed using the MIRAN method [62].

which the character wants to move. The preferred velocity is then computed as the vector that takes the character to p_{att} at its preferred walking speed. Figure 10.2a summarizes the concept of route following. It is comparable to one of the 'steering behaviors' described by Reynolds [126], who was among the first to acknowledge route following as a separate process.

The first algorithm, the *Indicative Route Method (IRM)* [74], defines p_{att} as the farthest point along π_{ind} that lies inside the largest obstacle-free disk containing the character's position. When more free space is available, p_{att} lies farther along the route and the amount of smoothing increases.

The successor of IRM, called *Modified and Indicative Routes and Navigation (MIRAN)* [62], defines a set of *candidate* attraction points along π_{ind} and chooses p_{att} as the *best* candidate according to the character's personal region preferences. In other words, the amount of smoothing and route shortening depends on the local region costs for that particular character. A user-controlled parameter determines how far along the route the candidate attraction points are allowed to lie which controls how closely the character will follow the route. Figure 10.3 shows an example of the results produced by the MIRAN method.

10.3.4 Local Movement

At the local movement level, the character might temporarily deviate from its route to resolve local collisions with other characters.

In early *collision-avoidance* algorithms, characters exerted attractive and repulsive *forces*, and physical laws of motion yielded new velocities for each character [46, 126]. A disadvantage of these models is that they are inherently reactive, instead of letting the characters actively choose how to move based on the movement of other characters.

Hence, more recent algorithms are based on *velocity selection* [9, 76, 105]. These algorithms let a character pick the best speed and direction from a range of options, based on a cost function. The cost of a candidate velocity is based on the difference to the character's preferred velocity v_{pref}, and on the predicted collisions with other characters based on their current movement. Thus, the character attempts to choose a velocity v_{new} that avoids collisions while being similar to v_{pref}.

Next to collision avoidance, local movement can also include other types of actions in which the character adapts its velocity in response to a local situation. For example, in the *Stream* model [36], a character adjusts its movement to match the average velocity of all neighboring characters that are moving in a similar direction. This improves the flow in dense crowds. Also, local rules can be used to simulate the behavior of small *social groups* of characters [77, 87].

Algorithms for local movement are usually based on a small number of neighboring characters within the character's *field of view*. Finding these neighboring characters is a computationally expensive step of the simulation loop. A data structure such as a kd-tree is suitable for answering nearest-neighbor queries [7]. Since the distribution of characters in the environment is constantly changing, this query structure is typically rebuilt in each simulation step. Our own software uses *nanoflann*, a high-performance implementation of a kd-tree [107].

When all characters have computed a new velocity, their positions are updated using time integration [74] and the next simulation step begins.

10.3.5 Animation

Finally, the animation level produces visual output by animating and translating the character's 3D model in the environment [6]. This is relatively simple if prerecorded 3D motion clips are available. Producing smooth and physically correct animations without requiring such data is an active research topic that is outside the scope of this thesis.

Note that the animation and the simulation usually have different framerates. Crowd simulations often use a fixed timestep of 0.1 seconds (i.e. 10 frames per second) [74], whereas smooth animation requires a much higher framerate, up to 60 or more frames per second in real-time gaming applications. We therefore assume that the animation level uses a separate loop. Whenever a *simulation* step finishes and all agent positions have been updated (in the simulation model only), the *animation* layer is notified and visually brings the agents to their new positions. Interpolation can be used to account for the differences in framerate.

10.3.6 Communication Between Levels

It is important to note that the planning process of our hierarchy is not purely serial. Events in the lower levels may cause a character to reconsider its global route. For instance, when an agent has reached its goal position, it returns to the global planning or high-level planning level to determine its next action. A character could also be triggered to perform a new action when a particular animation has finished. Another example is *re-planning*: if a part of the environment turns out to be too crowded, or if a section of the indicative route is unexpectedly blocked by a dynamic obstacle, an agent may choose to reconsider its route and take a detour, as discussed in Chapter 8. In Figure 10.1, this communication between levels is indicated by dotted arrows that point from lower to higher levels.

10.4 Implementation Details

In this section, we describe our crowd simulation software based on the ECM navigation mesh. The software has been used throughout various chapters of this thesis. It implements the three center levels of the generic framework from Section 10.3. Thus, it can be applied to the *geometric* planning problems induced by a *high-level* planner, and its results can be sent to e.g. a game engine to add a sophisticated *animation* component.

The ECM software has been written in platform-independent C++. It uses components of the Boost library [14] for Boolean geometry operations and for computing Voronoi diagrams. Optional dependencies include Vroni (for computing the ECM in an alternative way; see Section 4.4) and OpenGL (for visualization in our own demo projects). For this thesis, the code has been compiled using Visual Studio 2013, but it has proven to compile and run successfully on Linux as well.

10.4.1 Environment Files

We have created an XML file format (with the extension '.env') that can be used to specify 2D environments and multi-layered environments (MLEs). Such an ENV file describes a set of layers. Each layer can contain *walkable areas* (polygons on which characters can walk), *obstacles* (non-walkable polygons that overrule walkable areas), and *openings* (walkable polygons that overrule obstacles). This combination of elements makes the environment easy to define. An example is shown in Figure 10.4.

To compute the ECM of a single layer, our software first computes the *obstacle space* $\mathcal{E}_{obs}$ as $(WA^C \cup OB) \setminus OP$, where WA, OB, and OP respectively denote the unions of all walkable areas, obstacles, and openings, and where WA^C is the complement of WA within a sufficiently large bounding box. We have implemented this conversion by using the Boost library [14]. Next, $\mathcal{E}_{obs}$ is translated to a representation required by the ECM construction algorithm of choice. Section 4.4 describes the various construction algorithms that we have implemented.

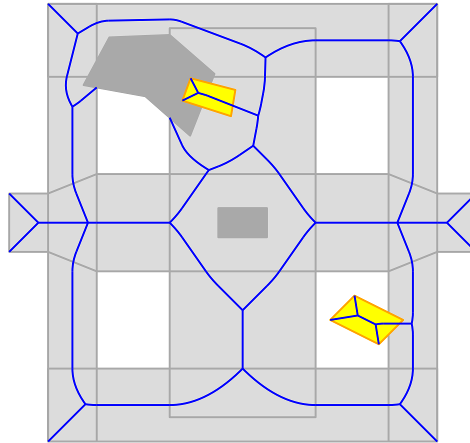


Figure 10.4: In an ENV file, users can define the walkable space of a layer in terms of walkable areas (light gray), obstacles (dark gray), and openings (yellow). The medial axis of the resulting free space $\mathcal{E}_{free}$ is shown in blue.

An MLE can contain *connections* that connect the free space of two layers. In an ENV file, a connection between two layers L_i and L_j is specified once in L_i with a reference to the layer ID of L_j, and once in L_j with a reference to L_i.

Furthermore, each layer can contain *weighted regions*: polygons with a certain label (e.g. 'grass' or 'road') to which agents can associate a personal weight. As explained in Section 10.3, the weighted regions are not included in the ECM; they are treated as a separate 'semantic layer' and are represented using a different data structure.

More specifically, we triangulate the weighted regions of each layer, as well as their complement within the layer's bounding box. The latter corresponds to the areas for which no region has been specified; we assign the 'default' region label to these areas. A layer without weighted regions therefore automatically receives two triangles with the default type.

The personal weights associated to regions are stored in separate *character settings* XML files that map region labels to numeric cost values. These settings files also contain other character parameters such as the radius, the preferred walking speed, and the methods and settings that the character uses for e.g. route planning, route following, and collision avoidance. Each set of character settings is called a *character profile* and is labelled with a name. Characters of different profiles can coexist in a simulation.

10.4.2 Algorithms of the Planning Hierarchy

The framework from Section 10.3 is *modular*: each level can be handled using various algorithms that can be combined arbitrarily. For the three center levels,

we have implemented and integrated several algorithms, in such a way that programmers can easily add new implementations.

For *global planning*, we have implemented A* search [45] on the medial axis. Characters can use this to compute shortest paths on the medial axis while ignoring edges that are too narrow for them to pass through. The resulting path can be converted to various types of indicative routes, such as short paths with a preferred amount of clearance, paths that follow the medial axis, or paths that stay on the left or right side of the free space. We have discussed these algorithms in Chapter 4. The re-planning algorithm from Chapter 8 and the density-based planning algorithm from Chapter 9 are also included.

When the environment contains weighted regions, characters can perform A* search on a grid that discretizes region information into cells of 1×1 meter. An improved path planning method for weighted regions has also been developed [63], but it has not yet been integrated for crowds in which different characters have different region preferences.

For *route following*, our framework includes both IRM [74] and MIRAN [62], which we have described in Section 10.3. At the *local movement* level, we have implemented two recent velocity-based collision avoidance algorithms by Moussaïd et al. [105] and Karamouzas et al. [76]. We have also included the popular ORCA collision-avoidance library [9]. The method by Moussaïd et al. is our default choice because it has yielded the most convincing results in practical scenarios, but users can easily switch between methods.

The *Stream* algorithm [36] is a method for local movement that can improve the crowd flow in dense scenarios. In our implementation, it lies between route following and collision avoidance: *Stream* updates the *preferred* velocity of a character, so it changes the input of the collision-avoidance method that follows.

Finally, we have included the option to simulate social groups of characters. The *Social Groups and Navigation (SGN)* method [87] applies partly to the local movement level: it uses forces to maintain group coherence. It also affects *global* planning: when a character loses track of the other members in its group, the character re-plans its path. More information can be found in the original publication of this method.

10.4.3 Simulation Loop

The main simulation loop of our software consists of fixed steps, or *frames*, that simulate 0.1 seconds. Users of our API (Section 10.4.6) can decide at what frequency this simulation loop is executed. For instance, calling the step function 10 times per second results in a *real-time* simulation, assuming that the work in one frame can be performed within 100 ms. Alternatively, if the step function is called as soon as the previous frame has finished, then the simulation runs *as fast as possible*.

Previous work has indicated that a step size of 0.1 seconds yields a simulation of sufficient precision [74, 76]. A coarser step size may cause characters to 'overshoot' their goals or collide with other characters too often, whereas a finer step size does not noticeably improve the precision.

We use a *fixed* step size to make the simulation deterministic. By contrast, in e.g. visualization loops, one can easily lower the framerate whenever the individual frames take too much time to compute. In our simulations, a variable step size based on performance would mean that the behavior of characters becomes less detailed when the simulation becomes computationally heavier. Therefore, we use a fixed step size, accepting the fact that computationally expensive simulations may not run in real-time.

We subdivide the content of a single simulation frame into multiple *substeps* that are processed one by one. Examples of substeps include computing the retraction of each character in the ECM (Section 4.3.2), computing a preferred velocity for each character, computing new velocities that avoid local collisions, updating the positions of all characters, and updating crowd density information in the ECM edges (Chapter 9).

Sequentially handling the *substeps* instead of the *characters* ensures that all characters use the same information. Instead, if the first character would perform all of the actions mentioned above before the second character gets its turn, different characters would be using different information, and the *order* in which the characters are stored would affect the results.

We also treat *dynamic updates* in a separate substep. If the ECM could change at any moment during a simulation frame, then the data used by characters (e.g. references to ECM cells) could become inconsistent. Therefore, all dynamic updates requested by the user are handled at a fixed point in the simulation loop, such that we can ensure that any references to ECM-related data are immediately updated.

10.4.4 Multithreading

Another advantage of our subdivision into substeps is that the computations *within a substep* can be computed for multiple characters in parallel. After all, the computations for one character do not depend on results computed for other characters within the same substep. In other words, the calculations within a substep are completely independent for the individual characters.

Thus, we can speed up the simulation by adding *multithreading* to each substep. We have implemented this by using basic OpenMP instructions [115] to automatically divide characters over multiple threads. Whenever a substep concerns social groups rather than individual characters, we parallelize over the groups instead. In Section 10.5, we will demonstrate how multithreading improves the performance of the simulation.

The only simulation substep in which the actions per character are *not* entirely independent (i.e. not thread-safe) is the substep that updates the *crowd densities* associated to ECM edges. To prevent characters from modifying the same density value at the same time, we use OpenMP to define critical regions in the code.

10.4.5 Demo Projects

Our ECM software contains two interactive *demo projects* that use OpenGL to visualize components such as the environment, the ECM, and the crowd.

The first demo project computes the ECM of an input environment. Users can inspect the result, add and remove obstacles that influence the ECM (using the local algorithms from Chapter 6 if desired), save the updated environment to a new ENV file, save the ECM to a file, and test the visibility algorithms described in Section 4.3. The second demo project can perform crowd simulations. Users can interactively pause and resume the simulation, add or remove characters, change the goals and personal properties of characters, and add or remove dynamic obstacles to affect the ECM (Chapter 6) and the global paths of characters (Chapter 8).

In both demo projects, the visual content of the demo window can be exported in vector format to the Ipe drawing tool [18] at any point in time. We have used this functionality to produce many of the figures in this thesis.

10.4.6 API

We have created an *API* that provides basic entry functions for e.g. loading an environment, computing or loading an ECM, intializing a simulation, adding a character with a particular profile, removing a character, and running a single simulation frame. This last API function fills an array of wrapper objects (C structs) that contain the new positions and orientations of each character, and it returns a pointer to this array. If an external program defines the exact same wrapper object, both programs can use the same data.

We have used our API to connect our crowd simulation software to the *Unity3D* game engine [158]. By linking characters in the simulation to animated 3D models in Unity, we can visualize moving crowds in 3D in real-time. Figure 10.5 shows an example of this combination.

The API is under active development and will be further improved in the future. We also plan to create an API for *environment modelling*. This could for instance be used by a visual editor to let users draw an environment in an intuitive way.

10.5 Experiments and Results

This section shows how our implementation performs in practice. In a number of environments, we analyze the computation time required by the various substeps of the simulation loop, both with and without using multithreading techniques.

Figure 10.5: Example of a crowd simulation that uses our software in combination with Unity3D. This simulation was created for the supplementary video of the *Stream* publication [36]. A 2D version of this environment also occurred in Chapter 7.

We use these results to analyze how many characters can currently be simulated in real-time. Our experiments use one CPU core and no parallel threads, except in Section 10.5.3 where we analyze how multithreading can improve the efficiency.

10.5.1 Experiment Setup

We performed our experiment in various environments from Chapter 4 and Chapter 5 of this thesis. In particular, we have included *City* (2D), *BigCity* (multi-layered), and *Zelda8x8* (2D) because they are large enough to support simulations of very large crowds, i.e. they are good choices for measuring scalability.

Settings. At the beginning of the simulation, we added N characters with (uniformly sampled) random start and goal positions. We repeated the experiment for various values of N. Whenever a character reached its goal, it received a new random goal position.

As our ECM framework includes many different algorithms that can be combined, there are many possible simulation settings. These settings are even allowed to vary between characters; for example, characters can have individual sizes, speeds, and collision-avoidance strategies. In this section, we give all characters the same settings such that the running times per substep are easier to interpret.

In line with real-life measurements [160], we gave each character a radius of 0.24 m and a preferred walking speed of 1.4 m/s. For global route planning, characters used the ECM to compute short indicative routes with a preferred clearance of 4 m. They used the density-based planning algorithm from Chapter 9 with a density weight $w = 10$, a viewing distance $d_D = 500$, and a re-planning

period of 10 seconds. For route following, characters used the Indicative Route Method (IRM) [74] because it currently has a more robust implementation than MIRAN. For local movement, they used vision-based collision avoidance [105] as well as the *Stream* method for improved coordination. We have not included any simulation components related to social groups [87] or dynamic updates (Chapter 6), for simplicity.

We acknowledge that many more settings are possible, and that each change in settings will lead to different behavior. However, the purpose of this experiment is to analyze the overall performance and scalability of our framework; we do not focus on behavior in this section. In future work, we intend to gain more insights into the effects of low-level simulation settings.

Measurements. We measured the average running time (in milliseconds) of each *substep* in the simulation, averaged over all frames. For example, we measured the total time spent on collision avoidance (which is one of the substeps), and we divided this by total number of frames that were simulated. We measure each substep separately to get a better insight in which substeps are the most computationally expensive.

The first seconds of the simulation are typically faster because characters are intially still evenly spread throughout the environment. After some time, the distribution of characters becomes more clustered, and more interesting challenges for collision avoidance occur. Therefore, we let the program 'warm up' for 100 simulation seconds to get a better image of its efficiency. More precisely, we performed our measurements between $T = 100$ s and $T = 400$ s, where T is the total simulated time (in seconds) that has passed.

Note that we measure the performance of our simulation in terms of 'milliseconds per frame', and not in 'frames per second' or 'CPU load'. We believe that this is the best reflection of the simulation's efficiency. Also, in real-time gaming applications, it is likely that only a certain percentage of the overall processing power can be allocated to crowd simulation. Performance results in milliseconds are easier to interpret for such applications as well.

10.5.2 Results

We will discuss the results for the *City* enviroment in detail. At the end of this section, we will briefly treat the other environments. For a visual impression, Figure 10.6 shows a simulation of $15{,}000$ characters in the *City* environment. We will now focus on the *performance* of the software.

Figure 10.7 shows how the average running time of each substep scales with the number of characters in the crowd. We have not included the performance of the *re-planning* step yet; we will analyze this component in Section 10.5.4. Before analyzing specific numbers, we will first discuss how each substep scales with the size of the crowd.

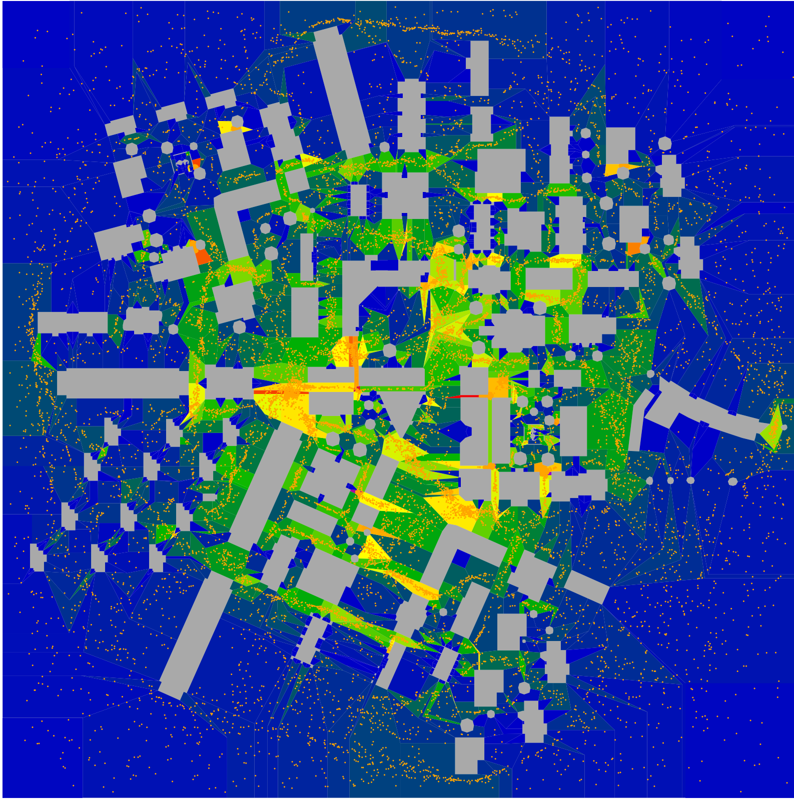


Figure 10.6: Still image of a crowd of 15,000 characters in the *City* environment. The background colors denote the crowd densities as described in Chapter 9. The characters have been enlarged for clarity; at their actual size, they do not intersect.

Figure 10.7a shows a close-up of the most efficient substeps. Most of these substeps take around constant time per character: computing the character's retraction in the ECM, computing a new attraction point and a preferred velocity, applying forces caused by (previously computed) collisions with other characters, updating the character's position, and updating ECM density information (Chapter 9) if the character has moved to an ECM cell of a different ECM edge. These substeps are independent of the other characters in the crowd, and their performance scales roughly linearly with the size of the crowd.

Nearest-neighbor structure. The remaining substeps are related to *neighbor relations* between characters and therefore take more time. Computing a kd-tree of N points (i.e. the centroids of characters) takes $\mathcal{O}(N \log N)$ time [7]. The *Stream* method and collision-avoidance method both perform nearest-neighbor queries in this kd-tree for each character. This takes around $\mathcal{O}(\log N)$ time per character because both methods search for a constant number of neighbors.

In our original publication [151], we used a *grid* for nearest-neighbor queries instead of a kd-tree. In crowded scenarios, many characters were located inside the same grid cell, which caused the running time to scale quadratically with the crowd size. Our new implementation based on kd-trees is much more scalable to large crowds.

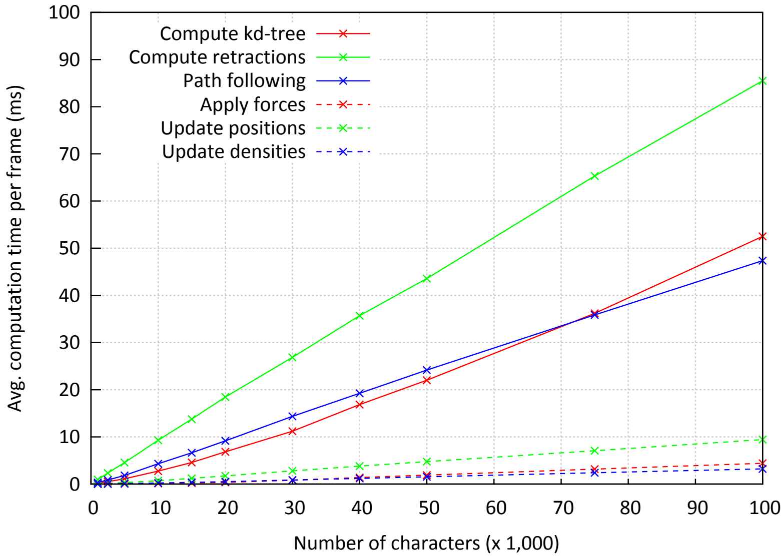
Collision avoidance. The substeps of the *Stream* method and the vision-based collision-avoidance (CA) method take more time, as shown in Figure 10.7b. Our original paper [151] did not yet include the *Stream* method. In this new analysis, we can see that *Stream* is a more efficient substep than the CA method, despite the fact that both methods perform nearest-neighbor queries. This indicates that nearest-neighbor queries are *not* the main source of inefficiency of the CA method.

Instead, the main issue is the CA method's use of *sampling*. The CA method evaluates a number of candidate walking directions for a character; for each candidate direction, it checks for potential future collisions with neighboring characters. Eventually, the method chooses the best candidate direction according to a cost function. The use of sampling adds a large constant factor to the running time. This constant factor appears to annihilate the (theoretically) super-linear trend of this substep, at least up to the crowd size that we have tested.

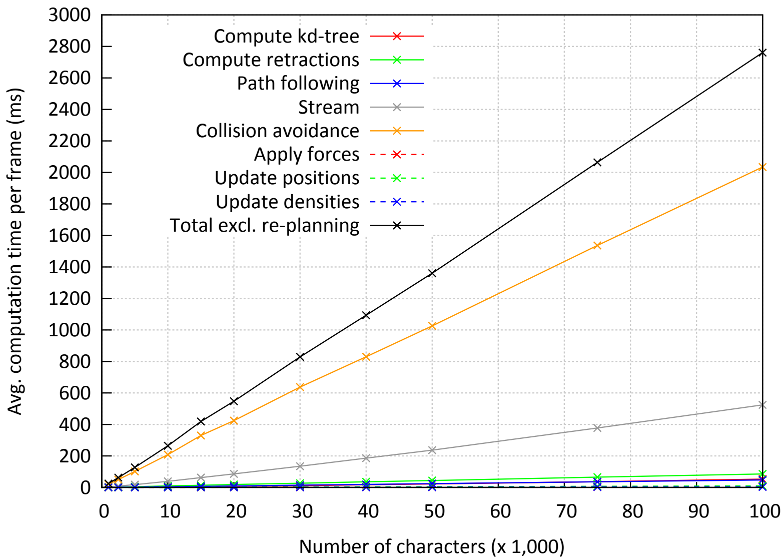
In the publication of this CA method, Moussaïd et al. [105] suggest to use a 'sufficiently large angular resolution' for this sampling scheme, but they do not propose a concrete value. In our implementation, the angle between two subsequent samples is $\frac{\pi}{40}$ radians (4.5 degrees), following the suggestion of a different CA method by Karamouzas and Overmars [76]. If we increase this value to $\frac{\pi}{10}$ radians (18 degrees) for testing purposes, the CA substep becomes considerably faster. We will refer to concrete running times in the next subsection. As always, sampling imposes a trade-off between efficiency and precision. Because it is currently unclear how the resolution affects the behavior of characters, we continue to use $\frac{\pi}{40}$ radians as the default value.

An alternative collision-avoidance method, ORCA [9], is based on linear programming rather than sampling. Preliminary experiments show that the ORCA implementation is about as fast as the *Stream* substep. However, we have noticed that ORCA does not yet combine well with our software in terms of character behavior. The main reason is that ORCA handles static obstacles in a different way. We intend to improve this in the near future. On the other hand, more research is required to evaluate which method yields the most 'human-like' behavior.

Total time. Figure 10.7b also displays the running time of all substeps combined, i.e. the average computation time of a complete frame. For example, simulating $20{,}000$ characters took 547 ms per frame on average. Interpolation between our measurements suggests that around $3{,}500$ characters can be simulated in real-time. However, note that we have used a *single CPU thread* in this experiment. We will now analyze how multi-threading can improve these results.



(a) Selected substeps



(b) All substeps + Total time

Figure 10.7: The performance of all simulation substeps in the *City* environment, using a single CPU thread. The horizontal axis denotes the number of characters in the simulation. The vertical axis denotes the average computation time of a substep (in milliseconds) throughout the entire simulation. Thus, each column of data points corresponds to one simulation run.

10.5.3 Multithreading

Next, we have performed the same experiment with 8 *parallel threads* in each substep. The running times are shown in Figure 10.8.

Figure 10.8a displays the same substeps as Figure 10.7a, and at the same scale. Using 8 parallel threads leads to a speed-up factor of more than 4 for various substeps, including the computation of retractions (from 85 to 19 ms at $100{,}000$ characters) and path following (from 47 to 10 ms). Unfortunately, *nanoflann* [107] cannot yet build a kd-tree on multiple parallel threads, and the corresponding substep is now relatively expensive. In theory, a multi-threaded implementation of this substep should be possible.

Figure 10.8b shows all substeps, with the y axis scaled by a factor of 4 compared to Figure 10.7b. We obtain a speed-up factor of 5 for *Stream* (from 524 to 100 ms at $100{,}000$ characters) and collision avoidance (from 2034 to 407 ms). In total, simulating $100{,}000$ characters becomes around 4.5 times as fast (from $2{,}761$ to 602 ms per frame). We can now simulate slightly more than $15{,}000$ characters in real-time with collision avoidance (CA) and *Stream* included.

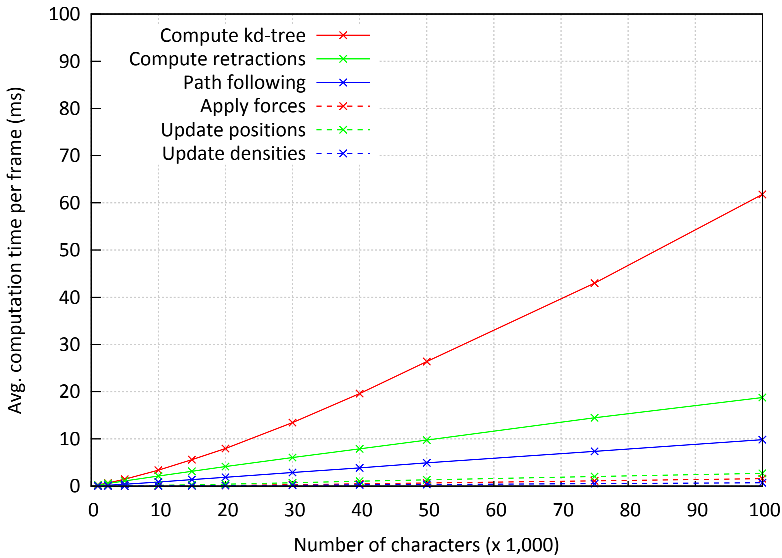
Again, improving the CA implementation can greatly improve the performance. For instance, if we increase the CA sample size to $\frac{\pi}{10}$ radians as described earlier, the CA substep takes 191 ms on average for $100{,}000$ characters, and we can simulate around $28{,}000$ characters in real-time.

The substeps that support multi-threading do not become 8 times as fast because the computations may not be equally fast for each character. Therefore, a thread may have to 'wait' until a new character is assigned to it, depending on how the threads are divided over the crowd. In future work, we intend to investigate more advanced multi-threading settings related to the dynamic scheduling of threads. Also, we intend to experiment with many more parallel threads on a machine with more CPU cores. For now, we have shown that simple multi-threading commands can already greatly improve the efficiency.

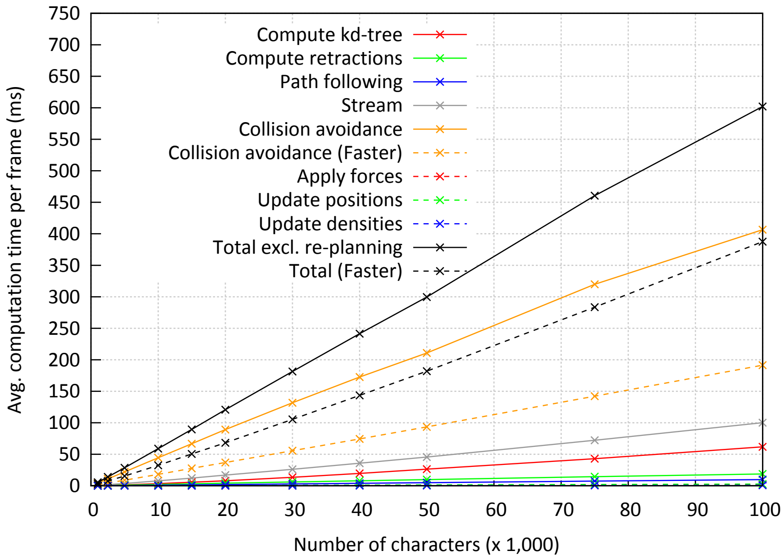
Given the rise of multi-core computers in recent years, we expect that simulations will be able to use many more parallel threads in the near future. In that respect, constructing the nearest-neighbor data structure on a single thread will eventually become the most expensive task. Thus, constructing this data structure in a parallel way will become a necessary improvement.

10.5.4 Analysis of Real-Time Performance

Because our software simulates discrete frames of 0.1 s, the simulation runs in *real-time* if the computations in each frame take at most 100 milliseconds. Of course, this assumes that there are no other components that also require processing power, such as visualization. Assuming that the simulation runs only 'in memory' without visualization or user interaction, we can analyze how many characters can be simulated in real-time. (For applications that can only spend a maximum amount of time per second on crowd simulation, a similar analysis can be conducted.)



(a) Selected substeps



(b) All substeps + Total time

Figure 10.8: The performance of all simulation substeps in the *City* environment, using 8 parallel CPU threads. The data representation in this figure is the same as in Figure 10.7. The label 'Faster' refers to experiments with a lower sampling frequency in the collision-avoidance method.

With 8 parallel threads, we can simulate the movement of 10,000 characters in the *City* environment in around 59 ms per frame (as shown in Figure 10.8b). This means that there are 41 ms left for other tasks such as re-planning. In Section 4.5.2, we have shown that indicative routes in the *City* environment can be computed in 0.3 ms on average. Furthermore, Section 9.6.5 has shown that even the most complex indicative routes have an average computation time of less than 1.5 ms. Assume for simplicity that planning an arbitrary indicative route takes 1 ms. This implies that around 40 characters can re-plan their paths in a single simulation frame. If the re-planning queries of all 10,000 characters are divided over time, then each character can plan a new path every 25 seconds without losing real-time performance. Of course, these results are different for other environments because the performance of path planning depends heavily on the complexity of the ECM.

Other environments. Finally, Figure 10.9 compares the *total* average frame times for *City* to those for the *BigCity* and *Zelda8x8* environments, again using 8 parallel threads. In all three environments, the performance scales roughly linearly with the number of characters due to the sampling-based collision-avoidance substep. We have tested smaller environments as well, such as *Military* and *Zelda* from Chapter 4 and *Station* from Chapter 5, but these were not large enough to support as many characters as the other environments.

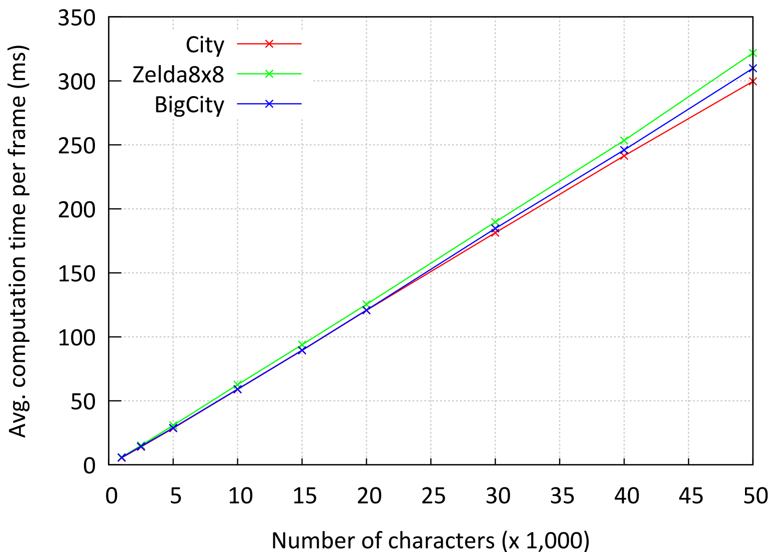


Figure 10.9: The average computation time per frame (in milliseconds) for various large environments from this thesis, using 8 parallel threads.

In conclusion, our software can currently simulate large crowds (over 10,000 characters) in real-time, and the majority of the computation time is spent on a

collision-avoidance method that can still be optimized. We emphasize that the program can simulate larger crowds as well, but not yet at real-time rates.

10.5.5 Real-World Simulations

Finally, we highlight a number of examples of how the framework has been used in practice to simulate real-world events.

The company Movares [106] has used our API for various simulations of real-life scenarios, such as virtual evacuations of the *Noord-Zuidlijn* metro stations in Amsterdam, and crowd flow simulations for the *Grand Départ* of the Tour de France in the city of Utrecht (2015). An impression of the latter project is shown in Figure 10.10.

The Pedestrian Dynamics crowd analysis software [60] has used the ECM framework in a similar way for multiple applications, such as crowd predictions for the King's Day ceremonies in Amsterdam (2014).

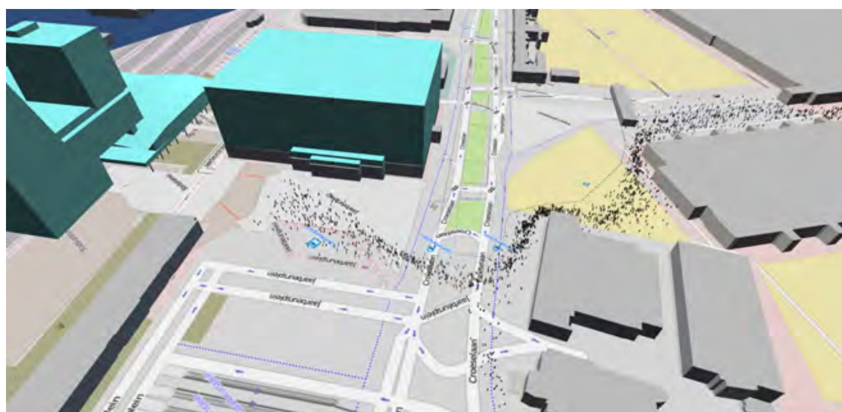


Figure 10.10: Our software has been used as a Unity plugin to simulate flows of spectators for the 2015 Tour de France *Grand Départ* in Utrecht. Simulations were performed using various lay-outs of fences to analyze which lay-out yielded the best and safest crowd flow.

10.6 Conclusions and Future Work

In this chapter, we have explained that a crowd simulation system involves many different types of algorithms. We have suggested a generic framework for crowd simulation that consists of five hierarchical *levels*: high-level planning, global route planning, route following, local movement, and animation. We have given examples of related work in each area.

Next, we have described the implementation of our crowd simulation software that uses the Explicit Corridor Map (ECM) navigation mesh. This software simulates the three center levels of the generic framework: it implements or integrates

various algorithms for route planning, route following, and local movement in a modular and extensible way. The program uses a carefully designed simulation loop and multi-threading techniques to obtain robust, deterministic, and efficient simulations. Our ECM software can be combined with other programs such as game engines.

Experiments show that our ECM framework can simulate large crowds in real-time using fixed frames that simulate 0.1 seconds. Collision avoidance is by far the most expensive component of the simulation; improving this algorithm and its implementation can greatly improve the results. On a consumer PC, we can currently simulate over $15{,}000$ characters in less than 0.1 s per frame in the *City* environment, excluding global route planning and re-planning. For smaller crowds, we have analyzed how many (re-)planning queries can be performed without losing real-time performance. Because the software benefits from multi-threading techniques, it is well-prepared for faster performance on (future) multi-core computers. We have also shown examples of how the industry has used our software to successfully simulate real-world events.

Discussion. In our current implementation, simulations tend to have minor problems in multi-layered environments (MLEs), particularly with respect to collision avoidance between characters. Characters cannot always recognize neighboring characters in other layers. We have yet to develop robust and efficient data structures and algorithms for *nearest-neighbor queries* in MLEs, such that collision avoidance in MLEs can work exactly as in 2D.

Another limitation of our simulations is that the local behavior of characters depends largely on their choice of indicative routes. For instance, if many characters try to follow short indicative routes that pass closely along corners of obstacles, then clusters of characters may form at these obstacle corners, and characters can get stuck. Local algorithms such as the *Stream* method can improve the crowd flow in some scenarios, but there is not one combination of algorithms and settings that always works well. To prevent such problems, characters should be willing to deviate farther from their indicative routes.

A related problem is that local and global planning are still largely detached in our implementation. In theory, our generic crowd simulation framework encourages communication from lower to higher levels; in practice, it is difficult to decide *when* this communication should take place. For example, if a cluster of characters is largely blocking the way for another character, it is unclear how 'bad' the situation should be to trigger a re-planning action.

Future work. The simulation companies that have used our software [60, 106] have shown that our implementation can be combined with methods for *high-level* planning. We would like to investigate more sophisticated AI techniques that can also efficiently model the individual knowledge and memory of characters. As explained in Chapters 8 and 9, this is particularly useful for environments with changing conditions such as dynamic obstacles or densities.

The discussion points mentioned earlier encourage future research on when and how the levels of our planning hierarchy should communicate in practice. The concept of strictly following an indicative route may be too limited to allow flexible behavior. Eventually, it might be that path following and collision avoidance make more sense as a single process in which all factors (e.g. the desired route, other characters, and weighted regions) are weighed in at once. Section 12.2 will discuss future work on the geometric planning levels in more detail.

Finally, the experiments of this chapter have focused on *efficiency* only. We have not yet analyzed the *behavior* or ‘realism’ of the simulated crowds. One reason for this is that the behavior of characters depends on many simulation settings; our software supports many combinations of algorithms, as well as detailed settings within each algorithm. Another reason is that it is largely unclear *how* the realism of a simulation can be measured in practice. In Chapter 12, we will argue that this is one of the most crucial topics for future work in our research field.

PART IV

Concluding Remarks

11 Summary and Conclusions

In a *crowd simulation*, virtual walking characters need to compute and traverse paths through a virtual environment while avoiding collisions. Simulations of large crowds occur increasingly often in computer games, in which real-time performance is required. Also, there is an increasing demand for crowd simulations of *real-world scenarios*. For example, crowd simulations can be used to predict dangerous situations during crowded events such as festivals, to estimate if a sports stadium can be evacuated within a certain amount of time, or to teach public safety personnel how to control crowds using a safe learning environment.

Thus, path planning and crowd simulation are important research topics. In Part I of this thesis, we have given an overview of these topics and related work. We have explained that a *navigation mesh* is an efficient representation of a virtual environment for the purpose of real-time path planning and crowd simulation. When planning a path in a navigation mesh, we actually compute a sequence of *regions* for the character to move through. Within these regions, the character can compute an *indicative route*, which it can then follow in real-time while avoiding other moving characters.

The rest of this thesis has investigated how to use navigation meshes to model and simulate *complex scenarios*. Examples of complex factors include characters with individual sizes and properties, multi-layered environments embedded in 3D, dynamic environments in which obstacles (dis)appear during the simulation, paths that need to be re-planned in real-time, or dense crowds in which characters should make intelligent decisions.

11.1 Navigation Meshes

Part II of this thesis revolved around *navigation meshes*: data structures that represent a virtual environment for path planning and crowd simulation.

In **Chapter 4**, we have presented the Explicit Corridor Map (ECM) navigation mesh for 2D environments with polygonal obstacles. The ECM is the *medial axis* of the free space $\mathcal{E}_{free}$, annotated with nearest-obstacle information at its bending points. For an environment with n obstacle vertices, the ECM requires $\mathcal{O}(n)$ storage and can be computed in $\mathcal{O}(n \log n)$ time based on any construction algorithm for the Voronoi diagram or the medial axis.

Because the ECM is a sparse graph, it is typically a more efficient representation of $\mathcal{E}_{free}$ than e.g. a grid. This enables more efficient path planning queries. Furthermore, the ECM allows for efficient fundamental operations such as point location, computing retractions [110], finding the nearest obstacle, and computing visibility information.

To compute a path for a character, we first find a path along the medial axis, which can then be converted to various types of *indicative routes* for the character to follow. Due to its clearance information, the ECM can be used to plan paths for disk-shaped characters of any radius. As such, the ECM is a navigation mesh that enables efficient path planning and simulation of heterogeneous crowds, i.e. crowds in which each character has its own properties and goals.

We have implemented the ECM based on robust software libraries for computing Voronoi diagrams. Our experiments show that the ECM can be constructed efficiently, and that it can be used to compute visibility polygons and indicative routes in real-time.

In many modern applications, the virtual environment is more complex and cannot be represented in 2D. For instance, imagine a multi-story building or a city with bridges and tunnels. However, the original 3D geometry is typically too detailed for the purpose of navigation.

In **Chapter 5**, we have defined a walkable environment (WE) as a set of polygons in $\mathbb{R}^3$ on which characters can walk, assuming a consistent direction of gravity $\vec{g}$ and a corresponding (virtual) ground plane P. Such a WE can be obtained from a raw 3D environment through a filtering process. Next, a *multi-layered environment* (MLE) is a WE that has been subdivided into *layers* such that any layer can be projected onto P without overlap. The layers are connected by *connections*: line segments whose vertices lie on the boundary of $\mathcal{E}_{free}$.

We have defined the medial axis (and therefore the ECM) for WEs and MLEs based on distances projected onto the ground plane P. To construct the medial axis of an MLE, we first compute the medial axis of all separate layers, while treating all connections as impassable obstacles. Next, we *open* the connections one by one. Opening a connection is analogous to deleting a line segment site from a Voronoi diagram, but with extra difficulties due to the multi-layered structure of the environment. For an MLE with n obstacle vertices and k connections, the medial axis has size $\mathcal{O}(n)$ and can be constructed in $\mathcal{O}(n \log n \log k)$ time. We expect that this construction time can be improved to $\mathcal{O}(n \log n)$ in the future.

By using multi-threading techniques, our implementation can construct the ECM in less than a second for moderately large MLEs, and within seconds for the most complex MLE in our test set. Because the multi-layered ECM locally has the same properties as the 2D ECM, many operations from Chapter 4 (including path planning and visibility queries) apply immediately to the multi-layered extension. The domain of MLEs presents interesting new problems, and we expect that other data structures for 2D environments can be extended to MLEs as well.

Chapter 6 investigated *dynamic* environments in which obstacles can appear or disappear during the simulation. For example, imagine a vehicle blocking a street, a bridge being destroyed, or a fence being added or removed. When such a dynamic event occurs, the navigation mesh should be updated, preferably in an efficient manner such that characters can respond to the event in real-time.

We have shown how the ECM can be updated *locally* when a convex polygonal obstacle is added or removed. These update operations are based on algorithms for inserting and deleting sites in a Voronoi diagram. In an environment of complexity n, a convex polygon of n' vertices can be inserted in $\mathcal{O}(\log n + n' + m_i)$ time, where $m_i \in \mathcal{O}(n)$ is the number of ECM cells that are visited during the insertion algorithm. A polygon can be deleted in $\mathcal{O}(\log n + m_d \log m_d)$ time, where $m_d \in \mathcal{O}(n)$ is the number of ECM cells around the obstacle to delete. In many cases, the obstacle only affects a small part of the ECM, and m_i and m_d are small numbers.

Our implementation and experiments show that we can update the ECM within milliseconds. This enables path planning and crowd simulation in dynamic 2D and multi-layered environments. However, the algorithms do not extend trivially to obstacles that intersect other geometry, and deleting an obstacle from a multi-layered environment may induce cases for which our current 2D algorithm does not work. We have sketched alternative approaches for future work.

Next to the ECM, various other types of navigation meshes have been developed by researchers worldwide over the past decade. So far, there has not yet been a standardized way of analyzing or comparing the quality of different navigation meshes. In **Chapter 7**, we have conducted the first *comparative study* of navigation meshes. We have presented definitions of environments and navigation meshes, as well as theoretical properties by which navigation meshes can be classified. Next, we have introduced various quantitative metrics that can measure the quality of a navigation mesh implementation: how accurately does it represent the free space $\mathcal{E}_{free}$, how efficient is its subdivision into regions, and how efficiently can it be constructed? Different application areas may assign different priorities to these metrics. Therefore, it is up to the user to decide which properties are the most relevant to the application at hand.

We have used these concepts to analyze and compare five state-of-the-art navigation meshes, along with a simple grid as a baseline. In our experiments, we have run all navigation mesh construction programs using the same hardware, input environments, and parameter settings. We believe that this work sets a new standard for the analysis and development of navigation meshes.

Our results confirm that navigation meshes are generally more efficient representations of $\mathcal{E}_{free}$ than grids. *Voxel-based* methods (which use approximations to obtain a walkable environment from raw 3D geometry) yield sufficient precision in many cases. However, these methods do not always preserve the environment's connectivity, they do not always scale well to large or complex environments, and they may depend on many parameter settings that need to be tweaked by

the user. By contrast, *exact* methods do not have these issues, but they currently require the environment to be pre-processed into 2D layers. A topic for future work therefore lies in robustly extracting $\mathcal{E}_{free}$ from 3D input *without* the disadvantages of voxel-based methods. Also, more metrics can still be developed, particularly in the areas of path planning efficiency, path length, and realism.

11.2 Path Planning and Crowd Simulation Algorithms

In Part III of this thesis, we have developed new algorithms for path planning and crowd simulation in navigation meshes. Although we have used the ECM from Part II as a guideline, we have described all concepts generically to make them applicable to other navigation meshes as well.

Chapter 8 considered *re-planning* in dynamic environments. When a dynamic obstacle has been added or removed (as described in Chapter 6), characters should check if their current paths are still valid and (if desired) optimal in the updated navigation mesh. When planning a new path, it is likely that information from the old path can be re-used to improve efficiency. Many existing re-planning algorithms require that each character remembers the search state from its previous query. This is too memory-intensive for applications with large crowds of characters.

We have presented *Optimal Dynamically Pruned A** (ODPA*), an extension of A* that prunes the search based on the old path and its relation to the dynamic event. Characters only need to remember their old paths and not the way in which these path were computed, which makes ODPA* suitable for crowd simulation.

Our experiments show that standard A* is faster in small graphs, but that ODPA* can outperform A* if the path is long and large parts can be re-used, such as in large and complex environments. Thus, ODPA* is an intuitive extension of A* that can improve the performance of real-time crowd simulations in large dynamic environments. Future challenges lie in handling multiple dynamic events in a single re-planning query, and in giving characters a sophisticated individual 'memory model' of the environment and its dynamic events.

It is common to let characters compute *short* paths through a navigation mesh. However, if a simulation contains many characters and the *crowd density* increases, this strategy may cause some areas to become overcrowded while other areas remain unused. In **Chapter 9**, we have shown how to annotate a navigation mesh with crowd density information that can be updated in real-time. The density in a navigation mesh region can be mapped to an expected walking speed in that region, using the theory of *fundamental diagrams* (empirically observed mappings from crowd density values to typical walking speeds).

Based on this density-annotated navigation mesh, we have presented a *density-based path planning* algorithm with which a character can plan a density-aware

path to its goal. Each character can use its own personal sensitivity to density-based delay. By regularly re-planning their paths, characters can respond to the most recent changes in density. If desired, re-planning can be made more efficient by letting characters only use density information up to a maximum path length.

Our experiments in the ECM show that density-based path planning per character, combined with periodic re-planning, can lead to *emergent* behavior in which the crowd automatically spreads among multiple routes. This can prevent characters from getting stuck in high-density regions; it can simulate dense crowd flows that were previously not possible. However, our algorithm is only one component in a larger system: the exact behavior of a crowd depends on many other simulation settings and implementation details. Furthermore, it is largely unclear how to evaluate the 'realism' of a crowd simulation in general.

In **Chapter 10**, we have suggested a hierarchical model for crowd simulations in general. The model consists of five levels: high-level planning, global route planning, route following, local movement, and animation.

Our ECM-based crowd simulation software models the three center levels, which concern the *geometric* aspects of path planning. We have implemented many different algorithms for route planning, route following, and local movement; these implementations can be freely combined in a modular way. By using multi-threading techniques and a careful subdivision of a simulation step into substeps, the software can simulate tens of thousands of characters in real-time.

The ECM software has been used throughout many chapters of this thesis. It has also been used successfully for real-time simulations of large crowds in real-world scenarios, e.g. to predict the crowd flow during the *Grand Départ* of the Tour de France in Utrecht in 2015, and to perform virtual evacuations of the *Noord-Zuidlijn* metro stations in Amsterdam.

Important topics for future work include analyzing the realism of a simulation, analyzing the global effects of low-level parameters, and making characters more intelligent without significantly harming the overall efficiency.

12 Discussion and Future Work

In this chapter, we highlight a number of important topics for future research, based on the results and limitations of our work up until now.

12.1 Navigation Meshes

We first look at topics related to Part II of this thesis.

12.1.1 Obtaining Walkable Environments

In Chapters 5 and 7, we have used the concepts of walkable environments (WEs) and multi-layered environments (MLEs) to represent the free space $\mathcal{E}_{free}$ of a 3D environment. In this thesis, we used WEs and MLEs as input, and we considered the problem of *automatically obtaining* them to be beyond the scope of the thesis.

To extract $\mathcal{E}_{free}$ from raw 3D geometry, *voxel-based* algorithms are currently the solution of choice. However, we have shown in Chapter 7 that such algorithms may lead to imprecise representations of $\mathcal{E}_{free}$, that they are often based on unintuitive parameters, and that they do not scale well to large environments.

We are therefore interested in developing *exact* algorithms for obtaining WEs from 3D geometry. Recent results are promising and numerically robust [123], but we have not yet achieved a real-time implementation that can compete with voxel-based algorithms in terms of efficiency. In the end, *hybrid* techniques may turn out to yield the best results. An example is NEOGEN [114] which uses voxels to obtain a raw subdivision into layers, followed by a higher-precision geometry reconstruction step per layer.

12.1.2 Obtaining and Using Multi-Layered Environments

A multi-layered environment (MLE) represents a walkable environment as a set of layers such that each layer can be handled in 2D. Both our own ECM and the NEOGEN navigation mesh [114] use this concept in their construction algorithm.

We expect that other path planning data structures and algorithms that are currently defined in 2D can also be extended to MLEs, based on the same projected distance function that we introduced for the medial axis. Examples of such data structures include the visibility graph [35], the constrained Delaunay triangulation, and the Local Clearance Triangulation [67]. A multi-layered version of the visibility graph could be used to plan *shortest paths* in multi-layered environments.

If a 2D construction algorithm assumes that $\mathcal{E}_{free}$ has been subdivided into polygonal regions, and it only uses these regions and their adjacency information, then it is likely that such an algorithm can be extended trivially to MLEs without affecting the running time. In Section 5.9, we have shown that this holds for computing indicative routes and visibility polygons.

Alternatively, if an algorithm is currently based on a 2D plane (using e.g. a plane sweep, incremental construction, or divide-and-conquer), it can most likely be extended to MLEs by treating all layers separately and then opening the connections one by one, such as in Chapter 5. In such cases, the running time of the algorithm may not be the same as in 2D. Important factors for the running time of the algorithm include the *number* of connections, as well as the *complexity* of the environment in the neighborhood of these connections.

It is therefore important to obtain MLEs with a convenient configuration of connections (although the definition of 'convenient' may differ per application). Hillebrand [54, 55] has shown that obtaining a *minimal number* of connections is NP-hard, but that heuristics can yield a small number of connections in practice. We are interested in developing fast algorithms for converting WEs to convenient MLEs based on other criteria, such as the width of connections or the distribution of connections throughout the environment.

Finally, we have seen that MLEs impose a number of extra difficulties in crowd simulations. In particular, dynamic deletions of obstacles do not always work in MLEs because theoretically difficult cases can occur, and collision avoidance between characters becomes less efficient due to more complex nearest-neighbor queries. We would like to perform a more thorough analysis of MLEs and their impact on the performance of a simulation.

12.1.3 Dynamic Environments and Robustness

In the near future, we expect that it will become more important to simulate dynamic environments that can change in arbitrary ways. We have shown that our algorithms from Chapter 6 are efficient, and we have explained how they can theoretically be extended to non-convex obstacles and intersecting geometry. However, it will be challenging to obtain a *robust* implementation that can handle all possible situations. This is particularly relevant for interactive applications in which the dynamic events are caused by the user and cannot be predicted beforehand. An example is a computer game in which the player can change the environment to create new routes in unpredictable ways.

Alternative techniques for updating the ECM include recomputing it from scratch in a parallel thread, or recomputing it only within a cleverly chosen bounding box. These techniques will most likely be less efficient than the purely local algorithms from Chapter 6, but they will be easier to implement robustly for intersecting geometry and multi-layered environments.

We would also like to investigate other types of dynamic geometry, such as elevators or moving platforms that are connected to different parts of the environment at different points in time.

12.2 Path Planning and Crowd Simulation Algorithms

Next, we discuss topics for future work that are related to Part III of this thesis.

12.2.1 Geometric Planning Levels

The crowd simulation framework described in Chapter 10 dinstinguishes between three levels of geometric planning: route planning, route following, and local movement. The underlying assumption is that the indicative routes are 'intelligent' enough to be traversable in combination with local collision avoidance. However, local methods may not be able to steer a character towards its goal if too many other characters are blocking the way. When this happens, the character should look for a different indicative route to follow.

Also, the distinction between global and local planning is not always evident. For instance, small obstacles such as street lanterns could be modelled as 'hard' obstacles in the navigation mesh that influence the high-level route choices of characters, but they could also be integrated into a local collision-avoidance method instead. Similarly, a dense cluster of characters could be inserted into the navigation mesh as a dynamic obstacle. Different choices lead to different behavior in the crowd.

A more specific potential problem of our current model is that characters always try to move towards an attraction point *on* the indicative route. Therefore, the crowd's behavior depends heavily on the type of indicative route that each individual character tries to follow.

We would like to investigate a more flexible approach. One option is to let characters also consider points to the left or right of the route at the path following level. Another idea is to not use precomputed indicative routes at all, but to choose attraction points on the fly by using the navigation mesh regions. Such approaches would also allow characters to locally navigate around dense areas or unattractive weighted regions. It might mean that path following and collision avoidance will eventually be merged into a single process.

12.2.2 Character Intelligence and Memory

In Chapters 8 and 9, we have discussed re-planning in environments with dynamically changing conditions (i.e. dynamic obstacles or changing densities). In both cases, a character's knowledge of the environment was modelled in a limited way.

If all characters use the same version of the navigation mesh, they always know about all dynamic events and the latest changes in density information.

A more realistic approach would be to let each character have its own *knowledge model* that encodes what the character currently knows about the environment. Ideally, each character should use its own version of the navigation mesh annotated with that character's personal information. Of course, this would significantly increase the memory usage of the simulation; as such, it would not be scalable to large crowds. It will be challenging to model detailed knowledge per character while still allowing tens of thousands of characters to be simulated.

In general, this thesis has used a simplified character representation to allow simulations of large crowds in real-time. As we add more knowledge to the crowd, each character gradually evolves from a moving disk into a complex artificial intelligence (AI) agent. Combining our algorithms with results from AI research will undoubtedly affect the computational efficiency of the simulation, but it can yield sophisticated models of characters that display more realistic behavior.

12.2.3 Validating Crowd Simulations

Next, *validating* the results of a crowd simulation is possibly the most important topic for future work. Crowd simulations are being used increasingly often in preparation of real-world events; Wijermans et al. [163] have given a rigorous overview of the field of *crowd management* and the ways in which this field can benefit from crowd simulations. However, it is still largely unclear to what extent the results of a crowd simulation correspond to real-world behavior. Thus, there is an increasing desire to measure this correspondence.

At a microscropic level, benchmark tools such as SteerBench [135] use various metrics to measure the quality of character behavior. Some of these metrics are based on logical ideas (e.g. 'pedestrians want to minimize the energy that they spend'), but it is not yet clear if the resulting numbers actually represent how well the simulation reflects real-world behavior.

Algorithms for local character behavior are often based on psychological concepts [46, 58, 105], and they often result in *emergent* crowd-wide behaviors that are also observed in real life (such as lane formation and stop-and-go waves). The ability to model such emergent phenomena is considered to be a desirable property [27, 46]. However, each model may still produce deadlocks or other undesired effects in particular scenarios; there is not a single model that guarantees appropriate behavior in all possible cases.

One could compare the trajectory of a simulated character to a path traversed in real life, but such a comparison would be highly scenario-specific. Another option is to detect and compare high-level patterns in *sets* of trajectories [159]. Also, as explained in Chapter 9, *fundamental diagrams* (i.e. empirically studied relations between crowd density and typical walking speeds) can be used to

compare simulations to real crowds at a macroscopic level [130, 167]. This idea has already been employed by researchers and developers of crowd simulation software [11]. However, fundamental diagrams are currently only well-studied for limited scenarios such as straight corridors and T-junctions [167], and the analysis depends heavily on the way in which crowd density is measured [167, 168].

We also stress that the output of a crowd simulation depends heavily on low-level parameters and implementation details. Our comparative study of *navigation meshes* (Chapter 7) was already influenced by such details, as well as by conceptual differences between algorithms. *Simulations* depend on even more of these factors, so analyzing and comparing their results will be even more challenging. It will be difficult to draw general conclusions on whether or not a particular simulation model produces realistic results.

12.3 Other Topics and Outlook

Of course, more directions for future research exist that reach further beyond the scope of this thesis. For instance, it would be interesting to combine walking characters with other simulated vehicles such as bicycles. Path planning under the *non-holonomic constraints* of vehicles has been studied intensively, as well as the various written and unwritten *traffic rules* that are involved when multiple vehicles are present. However, there is not yet one system in which many entities of different types interact realistically in real-time.

Also, this thesis has deliberately focused on characters that move along walkable surfaces. Some applications may feature AI-controlled characters that can perform more *complex actions* such as jumping, climbing, and crouching. These complex actions are tightly coupled with the field of 3D character animation; in other words, characters can no longer be simplified to disks with velocities.

For instance, jumping behavior can be included by annotating a navigation mesh semi-automatically [16], based on application-specific assumptions such as a maximum jump height or a fixed character shape. Other approaches use more traditional motion planning techniques for each individual character [98], but these techniques are too complex to be suitable for large crowds. A large future challenge is to *automatically* create data structures that can answer navigation-related queries in real-time for *crowds* of characters that can perform many actions. Any solution will undoubtedly include trade-offs between the overall performance and the amount of detail per character.

Eventually, the ideal hypothetical solution should be able to perform real-time simulations of many intelligent virtual characters that can perform complex tasks, combined with other entities such as vehicles, in large detailed environments that can change in unpredictable ways. Furthermore, the results of these simulations

should be provably realistic; they should reflect behavior observed in real life. To achieve such a goal, researchers will have to combine the expertise of many different communities.

Given the range of possible directions for future research, as well as the increasing relevance of simulations to society, we expect that crowd simulation and its related topics will remain important research areas for many decades.

Bibliography

[1] A. Aggarwal, L.J. Guibas, J. Saxe, and P.W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete & Computational Geometry*, 4:591–604, 1989.

[2] S. Aine and M. Likhachev. Anytime Truncated D*: Anytime replanning with truncation. In *Proceedings of the 6th International Symposium on Combinatorial Search*, pages 2–10, 2013.

[3] S. Ali, K. Nishino, D. Manocha, and M. Shah. *Modeling, Simulation and Visual Analysis of Crowds: A Multidisciplinary Perspective*. Springer, 2013.

[4] L. Arge, G.S. Brodal, and L. Georgiadis. Improved dynamic planar point location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–314, 2006.

[5] F. Aurenhammer, R. Klein, and D.T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific Publishing Company, 2013.

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

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[8] J.P. van den Berg, D. Ferguson, and J.J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proceedings of the 23rd IEEE International Conference on Robotics and Automation*, pages 2366–2371, 2006.

[9] J.P. van den Berg, S.J Guy, M.C. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *Proceedings of the 14th International Symposium on Robotics Research*, pages 3–19, 2011.

[10] G. Berseth, M. Kapadia, and P. Faloutsos. ACCLMesh: Curvature-based navigation mesh generation. In *Proceedings of the 8th ACM SIGGRAPH International Conference on Motion in Games*, pages 97–102, 2015.

[11] A. Best, S. Narang, S. Curtis, and D. Manocha. DenseSense: Interactive crowd simulation using density-dependent filters. In *Proceedings of the 13th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 97–102, 2014.

[12] H. Blum. A transformation for extracting new descriptors of shape. *Models for the Perception of Speech and Visual Form*, pages 362–380, 1967.

[13] R. Bonfiglioli, W. van Toll, and R. Geraerts. GPGPU-accelerated construction of high-resolution generalized Voronoi diagrams and navigation meshes. In *Proceedings of the 7th ACM SIGGRAPH International Conference on Motion in Games*, pages 24–30, 2014.

[14] Boost. The Boost C++ library. `http://www.boost.org/`, 2016.

[15] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004.

[16] S. Budde. Automatic generation of jump links in arbitrary 3D environments for navigation meshes. Master's thesis, Humboldt-Universität zu Berlin, 2013.

[17] CGAL. The Computational Geometry Algorithms Library. `http://www.cgal.org/`, 2016.

[18] O. Cheong. The Ipe extensible drawing editor. `http://ipe7.sourceforge.net/`, 2016.

[19] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80:1412–1434, 1992.

[20] F. Chin, J. Snoeyink, and C.A. Wang. Finding the medial axis of a simple polygon in linear time. *Discrete & Computational Geometry*, 21:405–420, 1999.

[21] S. Curtis, A. Best, and D. Manocha. Menge: A modular framework for simulating crowd movement. *Collective Dynamics*, 1(A1):1–40, 2016.

[22] W. Daamen. SimPed: A pedestrian simulation tool for large pedestrian areas. In *Proceedings of the EuroSIW Conference*, 2002.

[23] W. Daamen. *Modelling passenger flows in public transport facilities*. PhD thesis, Delft University of Technology, 2004.

[24] L. Deusdado, A.R. Fernandes, and O. Belo. Path planning for complex 3D multilevel environments. *Proceedings of the 24th Spring Conference on Computer Graphics*, pages 187–194, 2008.

[25] O. Devillers. On deletion in Delaunay triangulations. In *Proc. 15th Annual ECM Symposium on Computational Geometry*, pages 181–188, 1999.

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

[27] D.C. Duives, W. Daamen, and S.P. Hoogendoorn. State-of-the-art crowd motion simulation models. *Transportation Research Part C: Emerging Technologies*, 37:193–209, 2013.

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

[29] G. Flötteröd and G. Lämmel. Bidirectional pedestrian fundamental diagram. *Transportation Research Part B*, 71:194–212, 2015.

[30] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[31] J.J. Fruin. *Pedestrian Planning and Design*. Metropolitan association of urban designers and environmental planners, 1971.

[32] F.M. García, M. Kapadia, and N.M. Badler. GPU-based dynamic search on adaptive resolution grids. In *Proceedings of the 31st IEEE International Conference on Robotics and Automation*, pages 1631–1638, 2014.

[33] R. Geraerts. Planning short paths with clearance using Explicit Corridors. In *Proceedings of the 27th IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.

[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*, 2010.

[35] S.K. Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

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

[37] Golaem. `http://www.golaem.com/`, 2016.

[38] I. Gowda, D. Kirkpatrick, D. Lee, and A. Naamad. Dynamic Voronoi diagrams. *IEEE Transactions on Information Theory*, 29(5):724–731, 1983.

[39] P.J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978.

[40] B.D. Greenshields. A study of traffic capacity. In *Proceedings of the 14th Annual Meeting of the Highway Research Board*, pages 448–477, 1935.

[41] D.H. Hale and G.M. Youngblood. Dynamic updating of navigation meshes in response to changes in a game world. In *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*, pages 427–432, 2009.

[42] D.H. Hale and G.M. Youngblood. Full 3D spacial decomposition for the generation of navigation meshes. In *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 143–147, 2009.

[43] D.H. Hale, G.M. Youngblood, and P.N. Dixit. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 173–178, 2008.

[44] D. Harabor and A. Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the 52th AAAI Conference on Artificial Intelligence*, pages 1114–1119, 2011.

[45] 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.

[46] D. Helbing and P. Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, 1995.

[47] D. Helbing and P. Mukerji. Crowd disasters as systemic failures: analysis of the Love Parade disaster. *EPJ Data Science*, 1(7):1–40, 2012.

[48] M. Held. VRONI and ArcVRONI: Software for and applications of Voronoi diagrams in science and engineering. In *Proceedings of the 8th International Symposium on Voronoi Diagrams in Science and Engineering*, pages 3–12, 2011.

[49] M. Held and S. Huber. Topology-oriented incremental computation of Voronoi diagrams of circular arcs and straight-line segments. *Computer-Aided Design*, 41(5):327–338, 2009.

[50] C. Hernández, J.A. Baier, and R. Asin. Making A* run faster than D*-Lite for path-planning in partially known terrain. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling*, pages 504–508, 2014.

[51] C. Hernández, X. Sun, S. Koenig, and P. Meseguer. Tree Adaptive A*. In *Proceedings of the 10th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 123–130, 2011.

[52] C. Hernández, T. Uras, S. Koenig, J.A. Baier, X. Sun, and P. Meseguer. Reusing cost-minimal paths for goal-directed navigation in partially known

terrains. *Autonomous Agents and Multi-Agent Systems*, 29(5):850–895, 2015.

[53] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

[54] A. Hillebrand, J.M. van den Akker, R. Geraerts, and J.A. Hoogeveen. Performing multicut on walkable environments. In *Proceedings of the 10th International Conference on Combinatorial Optimization and Applications*, pages 311–325, 2016.

[55] A. Hillebrand, J.M. van den Akker, R. Geraerts, and J.A. Hoogeveen. Separating a walkable environment into layers. In *Proceedings of the 9th ACM SIGGRAPH International Conference on Motion in Games*, pages 101–106, 2016.

[56] M. Höcker, V. Berkhahn, A. Kneidl, A. Borrmann, and W. Klein. Graph-based approaches for simulating pedestrian dynamics in building models. In *eWork and eBusiness in Architecture, Engineering and Construction*, pages 389–394, 2010.

[57] K.E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. *International Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[58] S.P. Hoogendoorn. Microscopic simulation of pedestrian flows. In *Proceedings of the 82nd Annual Meeting at the Transportation Research Board*, pages 1–11, 2003.

[59] G.A. Hyslop and E.A. Lamagna. Error free incremental construction of Voronoi diagrams in the plane. In *ACM/SIGAPP Symposium on Applied Computing*, pages 388–396, 1993.

[60] Incontrol Simulation Solutions. Pedestrian Dynamics. `http://www.pedestrian-dynamics.com/`, 2015.

[61] N.S. Jaklin. *On Weighted Regions and Social Crowds: Autonomous-agent Navigation in Virtual Worlds*. PhD thesis, Utrecht University, 2016.

[62] N.S. Jaklin, A.F. Cook IV, and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds*, 24(3):285–295, 2013.

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

[64] N.S. Jaklin, W.G. 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*, pages 11–14, 2013.

[65] J. Janer, R. Geraerts, W.G. van Toll, and J. Bonada. Talking soundscapes: Automatizing voice transformations for crowd simulation. In *Proceedings of the AES 49th International Conference on Audio for Games*, pages 1–7, 2013.

[66] M. Kallmann. Navigation queries from triangular meshes. In *Proceedings of the 3rd International Conference on Motion in Games*, pages 230–241, 2010.

[67] M. Kallmann. Dynamic and robust Local Clearance Triangulations. *ACM Transactions on Graphics*, 33(5), 2014.

[68] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained Delaunay triangulations. *Geometric Modeling for Scientific Visualization*, pages 241–257, 2003.

[69] M. Kallmann and M. Kapadia. *Geometric and Discrete Path Planning for Interactive Virtual Worlds*. Morgan & Claypool Publishers, 2016.

[70] M. Kallmann and M. Matarić. Motion planning using dynamic roadmaps. In *Proceedings of the 21st IEEE International Conference on Robotics and Automation*, pages 4399–4404, 2004.

[71] M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano, and N.I. Badler. Multi-domain real-time planning in dynamic environments. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 115–124, 2013.

[72] M. Kapadia, N. Pelechano, J. Allbeck, and N.I. Badler. *Virtual Crowds: Steps Toward Behavioral Realism*. Morgan & Claypool Publishers, 2015.

[73] I. Karamouzas, J. Bakker, and M.H. Overmars. Density constraints for crowd simulation. In *Proceedings of the 1st International IEEE Consumer Electronics Society's Games Innovations Conference*, pages 160–168, 2009.

[74] I. Karamouzas, R. Geraerts, and M.H. 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.

[75] I. Karamouzas, R. Geraerts, and A.F. van der Stappen. Spacetime group motion planning. In *Proceedings of the 10th International Workshop on the Algorithmic Foundations of Robotics*, pages 227–243, 2012.

[76] I. Karamouzas and M.H. Overmars. A velocity-based approach for simulating human collision avoidance. In *Proceedings of the 10th International Conference on Intelligent Virtual Agents*, pages 180–186, 2010.

[77] I. Karamouzas and M.H. Overmars. Simulating and evaluating the local behavior of small pedestrian groups. *IEEE Transactions on Visualization and Computer Graphics*, 13:394–406, 2012.

[78] M.I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Proceedings of the 1st International Symposium on Voronoi Diagrams in Science and Engineering*, pages 51–62, 2004.

[79] L.E. Kavraki, P. Švestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[80] J. Kelly, A. Botea, and S. Koenig. Offline planning with Hierarchical Task Networks in video games. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 60–65, 2008.

[81] E. Khramtcova and E. Papadopoulou. Linear-time algorithms for the farthest-segment Voronoi diagram and related tree structures. In *Proceedings of the 26th International Symposium on Algorithms and Computation*, pages 404–414, 2015.

[82] P.M. Kielar, D.H. Biedermann, and A. Borrmann. MomenTUMv2: A modular, extensible, and generic agent-based pedestrian behavior simulation framework. Technical Report TUM-I1643, Technische Universität München, Institut Für Informatik, 2016.

[83] D.G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proceedings of the IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 18–27, 1979.

[84] A. Kneidl and A. Borrmann. How do pedestrians find their way? Results of an experimental study with students compared to simulation results. In *Proceedings of the International Conference on Emergency Evacuation of People from Buildings*, 2011.

[85] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the 18th National Conference of Artificial Intelligence*, pages 476–483, 2002.

[86] S. Koenig, M. Likhachev, and D. Furcy. Lifelong Planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[87] A. Kremyzas, N.S. Jaklin, and R. Geraerts. Towards social behavior in virtual-agent navigation. *Science China - Information Sciences*, 59(11):112102, 2016.

[88] J.J. Kuffner and S.M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings of the 17th IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[89] W.H.K. Lam, J.Y.S. Lee, and C.Y. Cheung. A generalised function for modeling bi-directional flow effects on indoor walkways in Hong Kong. *Transportation Research Part A*, 37:789–810, 2003.

[90] F. Lamarche. TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints. *Computer Graphics Forum*, 28(2):649–658, 2009.

[91] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[92] S.M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[93] D.T. Lee. Medial axis transformation of a planar shape. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):363–369, 1982.

[94] D.T. Lee and R.L. Drysdale III. Generalization of Voronoi diagrams in the plane. *SIAM Journal on Computing*, 10(1):73–87, 1981.

[95] W. Lee and R. Lawrence. Fast grid-based path finding for video games. In *Advances in Artificial Intelligence*, volume 7884 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2013.

[96] Legion. `http://www.legion.com/`, 2016.

[97] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An anytime, replanning algorithm. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, pages 262–271, 2005.

[98] T. Lopez, F. Lamarche, and T.-Y. Li. Space-time planning in changing environments: using dynamic objects for accessibility. *Computer Animation and Virtual Worlds*, 23:87–99, 2012.

[99] C. Loscos, D. Marchal, and A. Meyer. Intuitive crowd behaviour in dense urban environments using local laws. In *Proceedings of the 1st Theory and Practice of Computer Graphics Conference*, pages 122–129, 2003.

[100] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computing*, 32(2):108–120, 1983.

[101] Massive Software. `http://www.massivesoftware.com/`, 2016.

[102] M. Mononen. Recast Navigation. `https://github.com/memononen/recastnavigation/`, 2016.

[103] M.A. Mostafavi, C. Gold, and M. Dakowicz. Delete and insert operations in Voronoi/Delaunay methods and applications. *Computers and Geosciences*, 29(4):523–530, 2003.

[104] F. De Moura Pinto and C.M. Dal Sasso Freitas. Dynamic Voronoi diagram of complex sites. *The Visual Computer: International Journal of Computer Graphics*, 27(6–8):463–472, 2011.

[105] M. Moussaïd, D. Helbing, and G. Theraulaz. How simple rules determine pedestrian behavior and crowd disasters. *Proceedings of the National Academy of Sciences*, 108:6884–6888, 2011.

[106] Movares. `http://www.movares.nl/`, 2016.

[107] Nanoflann. `https://github.com/jlblancoc/nanoflann/`, 2016.

[108] R. Narain, A. Golas, S. Curtis, and M.C. Lin. Aggregate dynamics for dense crowd simulation. *ACM Transactions on Graphics*, 28:1–8, 2009.

[109] Oasys Software. MassMotion. `http://www.oasys-software.com/`, 2016.

[110] C. Ó'Dúnlaing and C.K. Yap. A 'retraction' method for planning the motion of a disc. *Journal of Algorithms*, 6(1):104–111, 1985.

[111] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. John Wiley and Sons, Ltd., 2nd edition, 2000.

[112] R. Oliva and N. Pelechano. Automatic generation of suboptimal navmeshes. In *Proceedings of the 4th International Conference on Motion in Games*, pages 328–339, 2011.

[113] R. Oliva and N. Pelechano. A generalized exact arbitrary clearance technique for navigation meshes. In *Proceedings of the 6th International Conference on Motion in Games*, pages 103–110, 2013.

[114] R. Oliva and N. Pelechano. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. *Computers & Graphics*, 37(5):403–412, 2013.

[115] OpenMP. The OpenMP C/C++ library for parallel programming. `http://openmp.org/`, 2016.

[116] S. Paris and S. Donikian. Activity-driven populace: A cognitive approach to crowd simulation. *IEEE Computer Graphics and Applications*, 29:34–43, 2009.

[117] S. Patil, J.P. van den Berg, S. Curtis, M.C. Lin, and D. Manocha. Directing crowd simulations using navigation fields. *IEEE Transactions on Visualization and Computer Graphics*, 17:244–254, 2010.

[118] N. Pelechano, J.M. Allbeck, and N.I. Badler. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 6th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 99–108, 2007.

[119] N. Pelechano, J.M. Allbeck, M. Kapadia, and N.I. Badler. *Simulating Heterogeneous Crowds with Interactive Behaviors*. CRC Press, 2016.

[120] J. Pettré, H. Grillon, and D. Thalmann. Crowds of moving objects: Navigation planning and simulation. In *Proceedings of the 24th IEEE International Conference on Robotics and Automation*, pages 3062–3067, 2007.

[121] 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*, pages 81–89, 2005.

[122] M. Phillips, A. Dornbush, S. Chitta, and M. Likhachev. Anytime incremental planning with E-Graphs. In *Proceedings of the 30th IEEE International Conference on Robotics and Automation*, pages 2436–2443, 2013.

[123] R.M. Polak. Extracting walkable areas from 3D environments. Master's thesis, Utrecht University, 2016.

[124] F. Preparata. The medial axis of a simple polygon. In *Mathematical Foundations of Computer Science*, volume 53, pages 443–450. Springer, 1977.

[125] PTV Group. VisWalk. `http://vision-traffic.ptvgroup.com/`, 2016.

[126] C. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of the Game Developers Conference*, pages 763–782, 1999.

[127] B.C. Ricks and P.K. Egbert. A whole surface approach to crowd simulation on arbitrary topologies. *IEEE Transactions on Visualization and Computer Graphics*, 20:159–171, 2014.

[128] T. Roos and H. Noltemeier. Dynamic Voronoi diagrams in motion planning. In *System Modelling and Optimization*, volume 180, pages 102–111. Springer, 1992.

[129] D. Schmalstieg and R.F. Tobler. Exploiting coherence in 2.5-D visibility computation. *Computers & Graphics*, 21:121–123, 1997.

[130] A. Seyfried, B. Steffen, W. Klingsch, T. Lippert, and M. Boltes. The fundamental diagram of pedestrian movement revisited - empirical results and modelling. In *Traffic and Granular Flow*, pages 305–314. Springer, 2005.

[131] M.I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[132] W. Shao and D. Terzopoulos. Autonomous pedestrians. *Graphical Models*, 69(5–6):246–274, 2007.

[133] A. Shoulson, N. Marshak, M. Kapadia, and N.I. Badler. Adapt: The agent development and prototyping testbed. In *Proceedings of the 17th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 9–18, 2013.

[134] SimWalk. `http://www.simwalk.com/`, 2016.

[135] 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*, pages 158–169, 2009.

[136] J. Snoeyink. Point location. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34. Chapman & Hall/CRC, 2nd edition, 2004.

[137] G. Snook. Simplified 3D movement and pathfinding using navigation meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[138] STEPS. `http://www.steps.mottmac.com/`, 2016.

[139] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[140] N. Sturtevant and S. Rabin. Canonical orderings on grids. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 683–689, 2016.

[141] A. Sud, R. Gayle, E. Andersen, S. Guy, M.C. Lin, and D. Manocha. Real-time navigation of independent agents using adaptive roadmaps. In *Proceedings of the 14th ACM Symposium on Virtual Reality Software and Technology*, pages 99–106, 2007.

[142] A. Sud, N. Govindaraju, and D. Manocha. Interactive computation of discrete generalized Voronoi diagrams using range culling. In *Proceedings of the 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, 2005.

[143] K. Sugihara and M. Iri. Construction of the Voronoi diagram for 'one million' generators in single-precision arithmetic. *Proceedings of the IEEE*, 80(9):1471–1484, 1992.

[144] X. Sun and S. Koenig. The fringe-saving A* algorithm - a feasibility study. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2391–2397, 2007.

[145] X. Sun, S. Koenig, and W. Yeoh. Generalized Adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 469–476, 2008.

[146] D. Thalmann and S.R. Musse. *Crowd Simulation*. Springer, 2nd edition, 2013.

[147] W. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. The Explicit Corridor Map: Using the medial axis for real-time path planning and crowd simulation. In *International Computational Geometry Multimedia Exposition*, pages 70:1–70:5, 2016.

[148] W. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. The Explicit Corridor Map: A medial axis-based navigation mesh for multi-layered environments. arXiv:1701.05141, 2017. In submission to a journal.

[149] W. van Toll and R. Geraerts. Dynamically Pruned A* for re-planning in navigation meshes. In *Proceedings of the 28th IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2051–2057, 2015.

[150] W. van Toll, N. Jaklin, and R. Geraerts. A generic multi-level framework for agent navigation, 2015. Poster at the 8th ACM SIGGRAPH International Conference on Motion in Games.

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

[152] W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th ACM SIGGRAPH International Conference on Motion in Games*, pages 91–100, 2016.

[153] W.G. van Toll, A.F. Cook IV, and R. Geraerts. Navigation meshes for realistic multi-layered environments. In *Proceedings of the 24th IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3526–3532, 2011.

[154] W.G. van Toll, A.F. Cook IV, and R. Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012.

[155] W.G. van Toll, A.F. Cook IV, and R. Geraerts. Real-time density-based crowd simulation. *Computer Animation and Virtual Worlds*, 23(1):59–69, 2012.

[156] P. Tozour. Building a near-optimal navigation mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[157] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. *ACM Transactions on Graphics*, 25:1160–1168, 2006.

[158] Unity3D Game Engine. `http://www.unity3d.com/`, 2016.

[159] H. Wang, J. Ondřej, and C. O'Sullivan. Path patterns: Analyzing and comparing real and simulated crowds. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 49–57, 2016.

[160] U. Weidmann. *Transporttechnik der Fussgänger - Transporttechnische Eigenschaften des Fussgängerverkehrs*. Literature Research 90, ETH Zürich, Institut für Verkehrsplanung, Transporttechnik, Strassen- und Eisenbahnbau, 1993. In German.

[161] R. Wein, J.P. van den Berg, and D. Halperin. The Visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications*, 36:66–87, 2007.

[162] E. Whiting, J. Battat, and S. Teller. Topology of urban environments: Graph construction from multi-building floor plan data. In *Proceedings of the 12th International Conference on Computer-Aided Architectural Design*, pages 115–128, 2007.

[163] N. Wijermans, C. Conrado, M. van Steen, C. Martella, and J. Li. A landscape of crowd-management support: An integrative approach. *Safety Science*, 86:142–164, 2016.

[164] S.A. Wilmarth, N.M. Amato, and P.F. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Proceedings of the 16th IEEE International Conference on Robotics and Automation*, pages 1024–1031, 1999.

[165] B. Yersin, J. Maïm, P. de Heras Ciechomski, S. Schertenleib, and D. Thalmann. Steering a virtual crowd based on a semantically augmented navigation graph. In *Proceedings of the 1st International Workshop on Crowd Simulation*, pages 169–178, 2005.

[166] B. Yersin, J. Maïm, F. Morini, and D. Thalmann. Real-time crowd motion planning: Scalable avoidance and group behavior. *The Visual Computer: International Journal of Computer Graphics*, 24(10):859–870, 2008.

[167] J. Zhang. *Pedestrian fundamental diagrams: Comparative analysis of experiments in different geometries*. PhD thesis, Forschungszentrum Jülich, 2012.

[168] M. van der Zwan. The impact of density measurement on the fundamental diagram. Master's thesis, Utrecht University, 2016.

Samenvatting

Een *crowdsimulatie* is een computerprogramma dat het gedrag van een menigte lopende mensen (karakters) simuleert. Zulke simulaties komen voor in computergames, waarin het gedrag van de karakters in real-time moet worden uitgerekend. Crowdsimulaties worden ook steeds belangrijker voor situaties in het echte leven: een simulatie kan bijvoorbeeld de drukte en veiligheid op een festival voorspellen, berekenen hoe snel een stadion geëvacueerd kan worden, of politiepersoneel opleiden om met grote menigtes om te gaan.

In dit proefschrift hebben we technieken bekeken die crowdsimulaties kunnen verbeteren. In deel I hebben we uitgelegd dat een *navigation mesh* de virtuele omgeving opdeelt in verbonden beloopbare vlakken, zodanig dat karakters er efficiënt routes mee kunnen uitrekenen naar hun individuele doellocaties. Elk karakter volgt in de simulatie een eigen route terwijl het obstakels en andere (bewegende) karakters in real-time ontwijkt.

In de rest van het proefschrift hebben we onderzocht hoe we navigation meshes kunnen uitbreiden en inzetten voor *complexe* scenario's, zoals gelaagde omgevingen in 3D, omgevingen die dynamisch veranderen, of drukke menigtes waarin karakters intelligente keuzes moeten maken.

Navigation Meshes

Deel II van het proefschrift draaide om *navigation meshes*.

In **hoofdstuk 4** hebben we de *Explicit Corridor Map* (ECM) beschreven, een navigation mesh gebaseerd op de *medial axis* (MA) en gerelateerd aan het Voronoi-diagram. Voor een 2D-omgeving met polygonale obstakels, met in totaal n obstakelpunten, heeft de ECM complexiteit $\mathcal{O}(n)$ en kan deze berekend worden in $\mathcal{O}(n \log n)$ tijd.

De ECM ondersteunt veel handelingen die nuttig zijn voor crowdsimulaties, waaronder het vinden van het dichtstbijzijnde obstakel voor een query-punt, het berekenen van zichtbaarheidsinformatie ('visibility polygons'), en het berekenen van routes voor cirkelvormige karakters van willekeurige grootte. Onze implementatie van de ECM kan deze operaties in real-time uitvoeren.

In **hoofdstuk 5** hebben we *gelaagde omgevingen* ('multi-layered environments') gepresenteerd. Een *beloopbare omgeving* is een verzameling beloopbare vlakken in 3D. Een gelaagde omgeving deelt zo'n omgeving op in *lagen* zodanig dat elke

laag in 2D gevisualiseerd kan worden. De lagen zijn verbonden via *connecties*: de lijnsegmenten waarlangs de beloopbare omgeving is opgesplitst. Een gelaagde omgeving is een nuttige vereenvoudiging van een 3D-omgeving voor padplanning en crowdsimulatie.

We hebben de MA (en daarmee de ECM) gedefinieerd voor gelaagde omgevingen, gebaseerd op geprojecteerde afstanden op het grondvlak. We hebben een algoritme beschreven dat de MA berekent door eerst alle connecties als gesloten obstakels te beschouwen en ze daarna één voor één te openen. Voor een gelaagde omgeving met k connecties en n obstakelpunten is de MA $\mathcal{O}(n)$ groot. Het constructie-algoritme kost $\mathcal{O}(n \log n \log k)$ tijd; deze looptijd kan mogelijk nog versneld worden tot $\mathcal{O}(n \log n)$.

Onze implementatie van dit algoritme kan de ECM snel uitrekenen, ook voor grote complexe omgevingen. Bovendien werken de meeste operaties uit hoofdstuk 4 automatisch ook in gelaagde omgevingen.

Dynamische omgevingen kunnen veranderen tijdens de simulatie, bijvoorbeeld als een voertuig een doorgang blokkeert of als een explosie een nieuwe route beschikbaar maakt. Dit heeft invloed op de paden die karakters kunnen volgen; daarom moet de navigation mesh worden bijgewerkt. In **hoofdstuk 6** hebben we algoritmen beschreven die de ECM efficiënt en lokaal bijwerken wanneer een obstakel dynamisch wordt toegevoegd of verwijderd. Onze implementatie kan deze dynamische updates uitvoeren binnen enkele millseconden.

Naast de ECM zijn in het afgelopen decennium meer navigation meshes ontwikkeld. Er bestond nog geen manier om de kwaliteit van navigation meshes te beoordelen of te vergelijken. In **hoofdstuk 7** hebben we de eerste *vergelijkende studie* van navigation meshes uitgevoerd.

We hebben theoretische eigenschappen beschreven waarmee we types navigation meshes kunnen classificeren, en metrieken die de kwaliteit van een mesh voor een bepaalde omgeving kunnen uitdrukken. Deze eigenschappen en metrieken vormen een nieuwe standaard voor (experimenteel) onderzoek op het gebied van navigation meshes.

Met deze componenten hebben we een theoretische en praktische vergelijking uitgevoerd van een aantal moderne navigation meshes. Een conclusie is dat *benaderende* algoritmen soms informatie verliezen en minder goed opschalen naar grotere omgevingen, maar dat *exacte* algoritmen voorbewerkte input verwachten en nog niet kunnen omgaan met willekeurige 3D-geometrie.

Algoritmen voor Padplanning en Crowdsimulatie

Deel III van dit proefschrift richtte zich op speciale algoritmen voor het berekenen van routes en het simuleren van menigtes.

In **hoofdstuk 8** hebben we padplanning in dynamische omgevingen bestudeerd. Nadat een obstakel is toegevoegd of verwijderd en de navigation mesh is aangepast, zoals beschreven in hoofdstuk 6, is de route van een karakter mogelijk ongeldig of niet meer optimaal. We hebben een algoritme ontwikkeld dat een pad in een navigation mesh efficiënt kan *herplannen* nadat een obstakel is toegevoegd of verwijderd. Dit algoritme, genaamd *Optimal Dynamically Pruned A** (ODPA*), maakt gebruik van het oude pad en de relatie tussen dat pad en de dynamische verandering. ODPA* voegt een aantal 'pruning'-regels toe aan het standaard A*-zoekalgoritme om het zoeken te versnellen in de context van herplanning. Omdat ODPA* geen extra geheugen gebruikt gedurende de simulatie, is het algoritme geschikt voor simulaties van grote menigtes.

Onze experimenten laten zien dat het standaard A*-algoritme sneller is in kleine omgevingen, maar dat ODPA* voor versnellingen kan zorgen bij lange complexe paden waarvan een groot deel kan worden hergebruikt. Het algoritme kan dus zorgen voor snellere simulaties van grote menigtes in complexe dynamische omgevingen.

In een navigation mesh is het gebruikelijk om karakters een *korte* route naar hun doel te laten berekenen. Bij grote menigtes kan dit leiden tot opstoppingen in gebieden die veel karakters tegelijk willen gebruiken, terwijl andere gebieden onbenut blijven. In **hoofdstuk 9** hebben we bestudeerd hoe karakters de *dichtheid* van de mensenmassa kunnen gebruiken bij het vinden van routes.

We annoteren een navigation mesh met de huidige dichtheid in elk gebied. De dichtheid in een gebied kan worden vertaald naar een verwachte loopsnelheid, gebaseerd op *fundamentele diagrammen* die het verband tussen dichtheid en typische loopsnelheid beschrijven. Door deze verwachte loopsnelheid te gebruiken bij het plannen, eventueel met extra hoge kosten voor vertraging, kunnen karakters routes berekenen die een voorkeur geven aan minder drukke gebieden.

Onze experimenten tonen aan dat padplanning gebaseerd op dichtheid kan leiden tot meer variatie: de menigte verspreidt zich automatisch over meedere routes. Met de juiste simulatie-instellingen kan dit bovendien de doorstroming in een omgeving helpen verbeteren.

In **hoofdstuk 10** hebben we een generiek raamwerk voor crowdsimulatie beschreven. Het raamwerk bestaat uit vijf niveaus; de middelste drie niveaus (globale padplanning, het volgen van routes, en lokaal gedrag) zijn *geometrisch* van aard en kunnen worden opgelost met behulp van een navigation mesh zoals de ECM.

Onze simulatiesoftware, gebaseerd op de ECM, is gebruikt voor de meeste experimenten in dit proefschrift. We hebben een aantal implementatiedetails van deze software beschreven, zoals een opdeling van elke simulatiestap in modulaire sub-stappen waarmee allerlei soorten algoritmen kunnen worden gecombineerd.

Onze experimenten laten zien dat de software grote menigtes in real-time kan simuleren. Het ECM-simulatieprogramma is met succes door andere onderzoekers en bedrijven gebruikt om scenario's uit de echte wereld te simuleren,

zoals de bezoekersstromen voor de Tour de France-start in Utrecht in 2015 en evacuatiestudies van metrostations voor de Noord-Zuidlijn in Amsterdam.

De datastructuren en algoritmen uit dit proefschrift kunnen ingezet worden om real-time crowdsimulaties in meerdere toepassingsgebieden te verbeteren. De meest opvallende onderwerpen voor vervolgonderzoek zijn het automatisch verkrijgen van een gelaagde omgeving uit onbewerkte 3D-geometrie op een exacte manier, het uitbreiden van karakters met meer kunstmatige intelligentie zonder dat te veel efficiëntie verloren gaat, en evalueren in hoeverre het gedrag in een simulatie overeenkomt met de werkelijkheid.

Acknowledgments

The work presented in this thesis has been made possible by many people. I will attempt to mention and thank them concisely, hoping not to skip any important names along the way.

I first met my daily supervisor ('co-promotor'), Roland Geraerts, at a course on crowd simulation back in 2010. It led to a small project under his supervision, followed by an MSc thesis project and eventually a PhD position. We have seen each other evolve throughout the years, and Roland has given me many opportunities to grow while keeping my personality in mind and leaving room for humor. In other words, *hartstikke mooi*.

My supervisor ('promotor'), Marc van Kreveld, was always available for discussions at all levels of detail. Marc has a keen eye for identifying the most important questions in a research project, and for developing sound definitions and proofs. Marc, thank you for sharing your knowledge and expertise.

Norman Jaklin and Arne Hillebrand have been wonderful office mates. I've had countless brainstorming sessions with them, and they have been great travel companions during conferences in e.g. Rome, Los Angeles, and San Francisco. Surprisingly, they also permitted (and even joined in on) my many voice impressions and 'sound noises' that delivered madness to room 4.17.

Angelos Kremyzas joined us as a scientific programmer for our ECM crowd simulation software, and we shared an office during the last few months before my thesis defense. With a great amount of effort, care, and laughter, Angelos has helped us convert my 'pet project' into a future-proof program.

Atlas Cook IV has been involved in our ECM-related papers from the start, and he joined me and Roland in my first conference abroad. I have learned a lot from his eye for detail and his incredibly positive attitude, be it from here or from across the globe.

Mihai Polak, together with Arne, has lent his assistance whenever things became three-dimensional, e.g. for pre-processing 3D environments into the input I required, and for computing coverage in 3D for Chapter 7 of this thesis.

For the theoretical results of Chapter 5, I've had helpful discussions with Maarten Löffler and Elena Khramtsova about multi-layered environments and Voronoi diagrams.

Many colleagues have also joined in numerous fun lunch conversations, social get-togethers, table soccer matches, and poker nights. Arne, thanks for the many table tennis duels and their minigames of ever-increasing intensity.

The comparative study of Chapter 7 has been a great opportunity to work with Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, and Julien Pettré. I'd like to thank these researchers for kindly sharing their source code, their virtual environments, and (above all) their critical and useful comments that have helped shape the corresponding paper. Thanks to Roy Triesscheijn for supplying a solid basis for this work as a student, and for continuing to help in his free time.

I was in the lucky position to work as a PhD student four days per week. This allowed me to spend most of my 'fifth days' with my former fellow UU student Jeroen van Knotsenburg as we explored the realm of app development mixed with other wonderfully nerdy activities. This combination would not have been possible without Jeroen's flexibility and patience, especially when my thesis was nearing completion.

At the same time, I was able to rediscover acting, music, and comedy. In these areas, Abel ten Berge was usually nearby to deliver support and inspiration in creative and hilarious ways. Jeroen and Abel, thanks for being the most logical choices for paranymphs imaginable.

In general, let me thank all other friends and family members who have offered advice and diversion at countless occasions. This includes Boni survivors Marnix, Tom, Mirjam, and Jan Anne, all $n + 1$ members of the 'For you an ask, for us a know' quiz team, everyone in the Super Awesome Ninjas improv group, and many more.

Finally, special thanks goes to my parents Martin and Annette, my sister Hester, and my brother-in-law Menno for understanding me like no other. I could probably spend a full-length chapter on explaining my gratitude to them.

Curriculum Vitae

Wouter Geert van Toll was born in Utrecht, The Netherlands in 1989. From 2000 to 2006, he went to the St. Bonifatiuscollege in Utrecht where he obtained his ‘gymnasium’ (secondary education) degree. He started studying computer science at Utrecht University in 2006, followed by the Game and Media Technology master program at the same university in 2009.

Wouter’s Master’s thesis, titled ‘A navigation mesh for efficient density-based crowd simulation in multi-layered environments’, was supervised by dr. Roland Geraerts. It was written in 2011 during an internship at Incontrol Simulation Solutions, and it led to publications that have been extended and rewritten for Chapters 5, 6, and 9 of this PhD thesis.

In 2012, Wouter started as a PhD student at Utrecht University under the supervision of dr. Roland Geraerts and prof. dr. Marc J. van Kreveld. He completed his thesis there in 2016, and he defended it in 2017.

Other publications. In addition to the articles incorporated into Parts II and III of this thesis, Wouter was involved in two other publications:

- R. Bonfiglioli, W. van Toll, and R. Geraerts. GPGPU-accelerated construction of high-resolution generalized Voronoi diagrams and navigation meshes. In *Proceedings of the 7th ACM SIGGRAPH International Conference on Motion in Games*, pages 24–30, 2014. [13]

- J. Janer, R. Geraerts, W.G. van Toll, and J. Bonada. Talking soundscapes: Automatizing voice transformations for crowd simulation. In *Proceedings of the AES 49th International Conference on Audio for Games*, pages 1–7, 2013. [65]