UNIVERSITY OF CALIFORNIA

Merced

GPU Rasterization Methods for Path Planning and Multi-Agent Navigation

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical Engineering and Computer Science

by

Renato Farias

2020

ABSTRACT OF THE DISSERTATION

GPU Rasterization Methods
for Path Planning
and Multi-Agent Navigation

by

Renato Farias
Doctor of Philosophy in Electrical Engineering and Computer Science
University of California, Merced, 2020
Professor Marcelo Kallmann, Chair

In this dissertation I present new GPU-based approaches for addressing path planning and multi-agent navigation problems. The proposed methods rely on GPU rasterization techniques to construct navigation structures which allow us to address these problems in novel ways.

There are three main contributions described in this document.

The first is a new method for computing Shortest Path Maps (SPMs) for generic 2D polygonal environments. By making use of GPU shaders an approach is presented to implement the continuous Dijkstra's wavefront propagation method, resulting in an SPM representation in a GPU's buffer which can efficiently give a globally optimal shortest path between any point in the environment and the considered source point. The proposed shader-based approach also allows several extensions to be incorporated: multiple source points, multiple source segments, and the incorporation of weights that can alter the wavefront propagation

in order to model velocity changes at vertices. These extensions allow SPMs to address a large range of real-world situations.

The second contribution addresses the global coordination of multiple agents flowing from source to sink edges in a polygonal environment. The same GPU-based SPM methods are extended to compute a Continuous Max Flow in the input environment, which can be used to guide agents through the environment from source edges to sink edges, leading to a flow representation stored in the frame buffer of the GPU. A method for extracting flow lanes respecting clearance constraints is also presented, achieving the maximum possible number of lanes to route agents across an environment without ever creating bottlenecks.

In order to address decentralized autonomous agents, the third contribution presents a new method for dynamically detecting and representing in SPMs regions where agents are bottlenecked. The incorporation of weighted barriers are proposed to model the corresponding time delays in corridors of the SPMs, in order to provide agents with alternative paths avoiding bottlenecks. In this way, a novel type of SPM is defined, providing optimal solutions from weights which reflect dynamic delays in the corridors of the environments.

The methods proposed in this dissertation present novel approaches for addressing optimal paths and agent distribution in planar environments. Given the continuous development of high-performance GPUs, the proposed methods have the potential to open new avenues for the development of efficient navigation algorithms and representations.

The dissertation of Renato Farias is approved.

Approved: _____

Professor Sungjin Im

Approved: _____

Professor Stefano Carpin

Approved: _____

Professor Marcelo Kallmann, Committee Chair

University of California, Merced

2020

To my dad, mom, & sis

TABLE OF CONTENTS

1 Introduction **1**

2 Literature Review **4**

2.1 Discrete Search Methods 4

2.2 Multi-Agent Path Planning 5

2.3 Distance Fields on Meshes 6

2.4 GPU Methods 7

2.5 Shader Programming 8

3 Optimal Path Maps on the GPU **9**

3.1 Introduction 9

3.2 Related Work 11

3.2.1 Contributions 12

3.3 Multi-Source Optimal Path Maps 13

3.3.1 Method Description 15

3.4 Segment Sources 22

3.5 Vertex Weights 24

3.6 Results and Discussion 29

3.6.1 Benchmarks 31

3.6.2 Discussion 36

3.7 Conclusions 38

4 Planar Max Flow Maps and Determination of Lanes with Clearance **42**

4.1 Introduction 42

4.2 Related Work 44

4.2.1 Multi-Agent Path Planning 45

4.2.2 Use of Flow Algorithms in Multi-Agent Path Planning 45
4.2.3 Continuous Max Flows 46
4.2.4 Contributions 47
4.3 Definitions and Overview 48
4.4 Computing Max Flow Maps 54
4.4.1 Critical Graph 55
4.4.2 Main Algorithm 56
4.4.3 Lane Extraction from a Max Flow Map 58
4.5 Clearance-Based Max Flow Maps 59
4.5.1 Improved Lane Extraction using the Min Cut 61
4.6 Length Optimization of Flow Lanes 62
4.7 Results and Discussion 63
4.7.1 Efficiency for Flowing Agents 65
4.7.2 Additional Results 67
4.7.3 Limitations and Future Work 68
4.8 Conclusions 71

5 Multi-Agent Navigation with Shortest Path Maps and Adaptive Weighted Barriers 74
5.1 Introduction 74
5.2 Related Work 75
5.3 Problem Definition 75
5.4 Bottleneck Detection 76
5.5 Weighted Barrier 80
5.6 Results 80
5.7 Conclusions 82

6 Conclusions 84

6.1 Discussion 84
6.2 Future Work 85

LIST OF FIGURES

3.1 Example of a multi-source Optimal Path Map computed on a polygonal scene with obstacles. There are three source points in the upper half of the scene, and two line segment sources in the lower half. The contour lines represent equal distances to their closest source. Contour lines are directly extracted from a distance field which is stored in the Z-Buffer as a result of our method. The blue cylinders are agents and each has a polygonal line representing its shortest path to the closest source. 10

3.2 Top row: example steps for computing a single-source SPM. Bottom row: corresponding 3D perspective view of each step. 13

3.3 Example of a shadow area. The line segment e represents the side of an obstacle. The red point $\mathbf{p_g}$ is the generator, points $\mathbf{p_1}$ and $\mathbf{p_2}$ are the endpoints of e, and point $\mathbf{p_m}$ is the middle point of e. Vectors $\widehat{\mathbf{v}}_1$, $\widehat{\mathbf{v}}_2$, and $\widehat{\mathbf{g}}$ are the normalized vectors from $\mathbf{p_g}$ to $\mathbf{p_1}$, $\mathbf{p_g}$ to $\mathbf{p_2}$, and $\mathbf{p_g}$ to $\mathbf{p_m}$, respectively. Points $\mathbf{p_{1s}}$, $\mathbf{p_{2s}}$, and $\mathbf{p_{ms}}$ are calculated in the following way: $\mathbf{p_{1s}} = \mathbf{p_1} + c_{svf}\widehat{\mathbf{v}}_1$, $\mathbf{p_{2s}} = \mathbf{p_2} + c_{svf}\widehat{\mathbf{v}}_2$, and $\mathbf{p_{ms}} = \mathbf{p_m} + c_{svf}\widehat{\mathbf{g}}$. The three triangles are sufficient to cover the entire area behind the segment. Using less than three triangles may not result in a correct shadow if the generator is close to the segment because the area becomes wide and thin. Value 4 is used for constant c_{svf} such that shadows of any size can be handled given that our obstacle coordinates are normalized. . . 19

3.4 The circled points on the segment sources are the critical points, which are projections of obstacle vertices. 23

3.5 Line segment source examples. Left: SPM of two segment sources intersecting at the center. Right: Several paths from agents represented as blue triangles to their closest points in a segment source. In both cases the white contours represent the distance field from the sources. 23

3.6 Top: an agent on foot plots a constant-speed shortest path. Bottom: the top-left vertex of the long rectangular obstacle has its weight increased representing the possibility of using a bicycle to speed-up traversal time. That possibility leads to the fastest path. 25
3.7 If generators with different weights are not allowed to propagate, they may generate inconsistencies in the OPM. It is necessary to store an extra copy of the data array for each uniquely weighted vertex in the scene, otherwise situations such as the above arise. 27
3.8 The weight of the highlighted vertex is increased. Paths to regions with blue dashed lines pass through the "faster vertex". With vertex weights, ambiguity is introduced because while pixels in the same region still have the same parent, their paths after the parent may be different. This happens because if the goal is sufficiently distant it becomes advantageous to take a detour to pass through the weighted vertex and gain faster travel mode. 28
3.9 Resulting OPMs as the highlighted vertex has its weight increased. The region generated by the clipped cone at that vertex gradually bloats outwards until it reaches both sides of the map. 29
3.10 Simulation dynamically updating an OPM as its sources change from points (left) to growing line segments (right). 29
3.11 Simulation where agents attempt to reach moving trains represented by dynamic segment sources. As the trains move each agent has direct access to a shortest path to the closest train. 30
3.12 Evacuation simulation. Left: central area has 1 exit. Right: 3 exits. 31
3.13 Simulation with multiple vertex weights. 32
3.14 Agents are trapped when relying only on collision avoidance (top), but not when extracting directions from the SPM (bottom). 33

3.15 Top: Four groups of agents navigating toward their goal segments using RVO-based local behavior without any global path planning. Bottom: SPMs are applied to provide preferred velocity vectors aligned with shortest paths in order to guide the agents. 33
3.16 The x axis represents the number of obstacle vertices in the scene, and the y axis represents the computation time in seconds. 40
3.17 Single-source OPM results. 40
3.18 Multiple-source OPM results. 41

4.1 Example max flow map from a source edge to a sink edge. While this flow map has optimal flow capacity, path lanes are subsequently extracted taking into account the required agent clearance and then optimized in length. 43
4.2 Left: this max flow routes agents from source to sink edges and is illustrated with flow lines following the flow directions. The non-useful parts of the environment have directions with magnitude 0 and are shown as gray regions. Right: our equivalent max flow map includes non-null directions for the non-useful parts of the flow. While in this work our flow maps present directions as they are generated by our computational method, it would also be possible to design directions for the non-useful portions that lead agents to a useful portion of the flow. The *min cut* of the environment is represented by the 3 illustrated segments. The sum of their lengths is equal to the max flow value. 50
4.3 Shortest Path Map (SPM) example. Contour lines represent points equidistant to the SPM's source segment (highlighted bottom segment. Discs represent agents whose polygonal shortest paths are also shown. Each region of the SPM, denoted with a same color, shares the same vertex to be taken when reconstructing a shortest path to the source segment. 52

4.4 Overview of the main steps of our overall approach. (a) Max flow map of the input environment. (b) Lanes extracted from the flow map. (c) Optimized lanes. (d) Using lanes to guide agents from source to sink. 53
4.5 Shortest pats for obstacles of infinite (left) and zero (right) cost. When an obstacle has zero cost it means that the portion of the path passing through the obstacle does not add any amount to the total cost of the path. 54
4.6 Segments considered (left) in order to identify the shortest segments connecting pairs of obstacles and boundaries that compose our critical graph (right). . 55
4.7 Example steps to generate a max flow map. 57
4.8 Left: because of how the clearance-insensitive flow winds up in this example, a lane with clearance that simply follows it might close off a space in a way that a corridor is blocked and the maximum number of possible lanes in the environment is not extracted. Right: In this environment, when the max flow map is generated from the bottom boundary, the maximum number of lanes is correctly extracted. To ensure that generated flows always generate the maximum number of lanes the desired clearance is taken into account during the flow construction. 60
4.9 The *gates*, or shortest segments, between the boundary and obstacles, and between pairs of obstacles, are contracted by displacing vertices of the obstacles such that corridors will fit an exact number of lanes. Here, as an additional optimization, the convex hull of the obstacles is adjusted. Because gate contraction is only important to distribute flows in corridors, there is no need to contract the gate between P_{bot} and P_{top}. 61
4.10 Scenarios 1 (left), 2 (middle), and 3 (right). 65
4.11 Snapshots of simulations on scenario 3. Despite both having the same amount of space and 8 lanes to start with, the SPM_{snk} only permits 4 agents to reach the exit at a time, while the max flow map permits all 8 to do so. 66

4.12 The top row shows a clearance-insensitive max flow map missing one lane compared to the clearance-based max flow map in the bottom row. The right column shows the same lanes after a length optimization process. 68
4.13 In the top row, the clearance-insensitive max flow map misses two lanes compared to the clearance-based max flow map in the bottom row. The right column shows the same lanes after a length optimization process. 69
4.14 Examples generating the maximum number of lanes before (left) and after (right) length optimization. 72
4.15 In this environment lanes are inefficient in terms of length whether the map is generated from both P_{bot} or P_{top}. 73
4.16 A flow map with multiple sources and sinks. The red lines are source segments belonging to P_{src} and the blue lines are sink segments belonging to P_{snk}. Every segment on the boundary inbetween them is used in the SPM generation process and thus the map is created such that agents from any source may travel to any sink, preventing crossing lanes. 73

5.1 SPM computed from our simulation environment. The red segment on the right is the segment source. Agents emerge from the blue segment on the left and follow their shortest paths to the source. 77
5.2 (a) Between update intervals, candidate segments are created when chains of bottlenecked agents going from one side of the corridor to the other are identified. The candidate segments are rendered beneath the agents as magenta lines. (b) Out of the set of candidate lanes, a weighted barrier (cyan) is created and placed down the middle, modifying the SPM. Previous candidate segments are removed. 79
5.3 Examples of how the SPM changes when weighted barriers are added. SPM source segments are in red and weighted barriers in cyan. 81

5.4 A snapshot of the simulation running in two scenarios: a) using only the SPM and local collision avoidance, agents form a large bottleneck in the middle, b) applying the bottleneck detection process and placing weighted barriers, more agents prefer taking the side routes, reducing the severity of the bottleneck. . . 83

6.1 Multiple disconnected flow sources and sinks. 86

LIST OF TABLES

3.1 Average time in seconds to compute a single-source OPM on various maps (shown in Figure 3.17). P and V are the number of polygons and vertices. . . . 34

3.2 A comparison of GPU-based techniques: (1) dynamic search using an uniform grid [Kapadia et al. 2013], (2) dynamic search using a quad-tree [Garcia et al. 2014], (3) brute-force SPM [Wynters 2013], and (4) our method. The number of obstacles (O) and vertices (V) in the environment are included for the last two methods. Some hardware details were not specified in the papers. 35

3.3 Number of path queries per second. F is the number of faces (triangles) on each map and AVP is the average number of vertices in the paths computed. The last column shows the improvement obtained with OPMs. 37

4.1 The left-most column indicates the used method. S: shortest paths to sink using SPM_{snk}. L: lanes from the max flow map. O: optimized lanes from the max flow map. The simulations had the agents continuously spawn at the source whenever there was space for them, and then the agents moved towards the sink according to the used method. The simulated period was of 60 seconds. Columns *min*, *max*, and *avg* refer to the minimum, maximum, and average path/lane lengths computed for the scene, respectively, and *n* is the total number of agents that were able to reach the sink in the allotted time. The three scenarios are illustrated in Fig. 4.10. 64

ACKNOWLEDGMENTS

I want to thank first and foremost Professor Marcelo Kallmann, without whom this journey would not even have been possible. Being accepted as a PhD student at UC Merced has changed my life for the better in many ways, both personal and professional. Professor Kallmann's guidance was instrumental in my development and helped build the body of work of my PhD.

I express my gratitude as well to committee members Professor Stefano Carpin and Professor Sungjin Im, for their time, guidance, and patience during my qualifying examination and my thesis defense.

I would also like to thank the Army Research Office who largely sponsored my thesis research under the grant number W911NF-17-1-0463.

Last but not least, I am grateful for my family's constant support from afar throughout this whole journey. I would not be here if it were not for them.

VITA

2014 Master Degree (Computer Graphics), Federal University of Rio de Janeiro, Brazil.

2011 B.S. (Computer Science), Federal Fluminense University, Brazil.

PUBLICATIONS

Ritesh Sharma, Renato Farias, and Marcelo Kallmann (2020). "Integrating Local Collision Avoidance with Shortest Path Maps." Eurographics poster paper.

Renato Farias and Marcelo Kallmann (2019). "Planar Max Flow Maps and Determination of Lanes with Clearance." Autonomous Robots (AURO).

Renato Farias and Marcelo Kallmann (2019). "Optimal Path Maps on the GPU." Transactions on Visualization and Computer Graphics (TVCG).

Renato Farias and Marcelo Kallmann (2018). "GPU-Based Max Flow Maps in the Plane." Robotics: Science and Systems (RSS).

Renato Farias and Marcelo Kallmann (2018). "Improved Shortest Path Maps with GPU Shaders." E-print arXiv:1805.08500.

Renato Farias (2014). "Point Cloud Rendering Using Jump Flooding." Masters disserta-

tion.

Renato Farias, Ricardo Farias, Ricardo Marroquim, Esteban Clua (2013). "Parallel Image Segmentation Using Reduction-Sweeps On Multicore Processors and GPUs." Conference on Graphics, Patterns, and Images (SIBGRAPI).

Renato Farias and Marcelo Kallmann. "Multi-Agent Navigation with Shortest Path Maps and Adaptive Weighted Barriers." To be submitted.

CHAPTER 1

Introduction

My work has focused on developing new GPU-based rasterization techniques for path planning and multi-agent navigation. Global navigation often depends on efficient path planning which is thus crucial in various applications from motion planning for robots to autonomous agents in virtual environments. Because of this, approaches for planning paths among obstacles have been extensively explored in diverse fields such as Artificial Intelligence (AI), Robotics, and Computer Animation.

While the problem has been extensively explored and many approaches have been introduced in recent years for computing paths among obstacles, efficiency of computing collision-free paths without global optimality guarantees has been the main focus. This reflects the fact that computing optimal paths efficiently is not a trivial task. No notable recent advancement has been achieved on practical methods for computing globally optimal shortest paths, which in regular planar environments are also known as Euclidean shortest paths.

One way of computing Euclidean shortest paths is by constructing a visibility graph of the environment and then running graph search on it. Unfortunately in the worst case the number of edges in the visibility graph is $\Theta(n^2)$, where n is the number of vertices describing the obstacles, which can significantly slow down path queries based on search algorithms running on the graph. Furthermore, each path query requires a new search.

Euclidean shortest paths can however be computed in optimal $\mathcal{O}(n \log n)$ time with Shortest Path Maps (SPMs). SPMs are constructed with respect to a "source point", and like Voronoi diagrams, SPMs partition the space into regions. Whereas regions in Voronoi diagrams share the same closest site, regions in SPMs share the same parent points along the shortest path to the source, which means that an SPM encodes shortest paths between a specified source and *all* other points in a particular planar environment.

While SPMs have been studied in Computational Geometry for several years, they have not been popular in practical applications. This is because their computation involves several complex steps, even when considering non-optimal construction algorithms. We have developed a GPU-based method that greatly simplifies the process while also introducing several novelties, as discussed in Chapter 3.

Beyond point-to-point path planning, another important class of problems relates to multi-agent navigation. The problem of optimally deploying multiple agents traversing a polygonal environment has important applications in many areas, for example, to control multiple robots in warehouses, to coordinate autonomous cars across narrow streets and to evaluate evacuation scenarios. While optimality can be defined by taking into account different variables such as energy, time, or distance travelled, in all cases the problem is difficult to be solved in a planar domain and is usually addressed in a discrete representation of the environment.

One main challenge in multi-agent navigation is to generate trajectories minimizing bottlenecks in generic polygonal environments with many obstacles. While many methods utilize a global path planner to generate goal points for agents, the avoidance or correction of bottlenecks is usually done with local collision avoidance heuristics. The alternative is to address the problem as an expensive global planning problem, which is the approach taken in the area of multi-agent path planning. We present two methods for dealing with this problem.

First, we present the approach detailed in Chapter 4 for taking into account the maximum flow capacity of a given polygonal environment to generate a system of bottleneck-free trajectories which do not require local collision avoidance. As a result we are able to generate trajectories of maximum flow from source to destination edges across a generic set of polygonal obstacles, enabling the deployment of large numbers of agents optimally with respect to the maximum flow capacity of the environment and guaranteeing no bottlenecks are formed.

In Chapter 5 we detail another approach where agents utilize an SPM to navigate towards the goal and our method dynamically detects bottlenecks and modifies the SPM with "weight barriers" in order alleviate the bottlenecks by making some agents prefer alternative paths. This reduces the impact of bottlenecks, generates agent behavior that is more realistic, and avoids the length inefficiency problem of max flow lanes.

CHAPTER 2

Literature Review

Our work is related to various different areas, from path planning and GPU computing to the computation of distance fields. The literature review below is organized according to these related areas.

2.1 Discrete Search Methods

Researchers in AI usually approach path planning with discrete search methods on grid-based environments, sometimes making use of hierarchical representations. Several advancements on discrete search methods have been proposed such as heuristic search, dynamic replanning, anytime planning [31], etc.; however, with few attempts to approximate Euclidean shortest paths. Probably the only exception is the work on "any-angle path planning" [42], which significantly improves the computed paths on grids with respect to getting close to a global optimal. However, still not guaranteeing to achieve paths with global optimality. The difficulty is that grid-based search leads to distance metrics that accumulate distances between centers of adjacent cells, and global optimality requires visibility computations along arbitrarily long straight line segments in any orientation. Nevertheless grid-based methods are robust and simple to be implemented, and thus very popular in many applications.

In Computer Animation, while several approaches have been introduced in recent years for

efficiently computing paths among obstacles, the state-of-the-art has focused mostly on the efficiency of computing collision-free paths. For instance, recent work has addressed new definitions of navigation meshes [16][44][25] but mostly addressing contributions related to speed of computation and computing paths with clearance. Given the complexity of the problem and the high computational cost of the simple approaches to it, global optimality is simply not addressed.

One way to compute globally-optimal Euclidean shortest paths is to first build the visibility graph of the environment [65] [45] [54] and then run a graph search algorithm on it [43] [8]. Previous work [30] has presented specific cases where the problem can be solved with greedy $\mathcal{O}(n \log n)$ time algorithms without explicitly building the entire visibility graph. However, a visibility graph can have $\Theta(n^2)$ nodes, where n is the number of vertices describing the environment, making it expensive to be computed, updated and queried. In addition, a new graph search is needed for each path query. It is therefore difficult to develop efficient methods based on visibility graphs.

2.2 Multi-Agent Path Planning

Related to planning paths for agents from their initial positions to target positions, and an important problem for a variety of applications, is the problem of Multi-Agent Path Planning (MAPP).

The vast majority of MAPP approaches developed to date are based on grid representations. In this case, the problem consists of finding trajectories for agents from given initial cells to given target cells in the environment grid, while avoiding cells marked as obstacles [35]. A popular approach to this problem is to plan paths individually for each agent and subsequently solve all conflicts. Different strategies for solving conflicts exist. For example, one approach is to recursively solve conflicts between pairs of agents [51]. Another is to have agents to follow paths which may have conflicts, and to let each agent address the

conflicts reactively in response to the local environment and nearby agents [62]. A number of variations and extensions exist, such as integration with roadmaps for scalability to higher dimensional problems including articulated structures [10]. While finding optimal solutions for several versions of the multi-agent path finding problem is known to be NP-hard [73, 57], unlabeled variations can be solved in polynomial time [71, 53].

2.3 Distance Fields on Meshes

Computing distance fields is a problem closely related to computing SPMs. While these methods do not represent the boundaries of a SPM decomposition, many of the methods could be extended to do so. Previous work including methods for computing distance fields are numerous and we include here just an overview of the area.

A popular method to compute distance fields is to rely on window propagation on meshes. The approach of Mitchell et al. [41] propagates front windows in unfolded triangles while solving front events during propagation, taking $\mathcal{O}(n^2 \log n)$ time to compute geodesics. It is also possible to perform window propagation without handling all events [7] [68], reaching $\mathcal{O}(n^2)$ time but in practice processing a high amount of windows. Window pruning techniques have also been investigated to improve practical running times [49].

Among other related methods, previous work has already addressed multiple types of sources, for instance when computing geodesic Voronoi diagrams with multiple sources [48] as well as polyline-sourced Voronoi diagrams [34] [69]. Such concepts however have not been implemented in the context of shortest path maps.

Mitchell et al. [41] present an algorithm to solve this discrete geodesic problem on arbitrary polyhedral surfaces using the continuous Dijkstra technique. Surazhsky et al. [56] build upon this method using a parametrization of the distance function over the edges of the mesh.

Qin et al. [49] achieve faster geodesic computation via a window pruning strategy, whereas Torchelsen et al. [60] focus on the creation of lower resolution meshes by identifying regions on the input mesh that can be unfolded with minimal area distortion. Lipman et al. [32] present a new distance metric called "biharmonic distance" based on the biharmonic differential operator.

2.4 GPU Methods

Previous work has investigated computing geodesic distances in parallel on GPUs [70][64] as well as rasterization-based GPU techniques for related applications such as Voronoi diagrams [23]. Although we also employ rasterization techniques to accumulate distances, our approach introduces the significant insight of placing clipped primitives at accumulated heights in order to compute a SPM. Furthermore, we employ new techniques taking advantage of modern programmable shaders such that it is no longer necessary to discretize geometry for rasterization, and instead we rely on specialized fragment shaders that directly fill in pixels during front expansions and without introducing errors from geometry discretization.

GPU methods have also been explored for path planning from grid-based searches, for example by performing multiple short-range searches in parallel [21], by parallelizing expansions per-pixel on uniform grids [26] and based on a quad-tree scheme [15]. However, grid-based approaches do not address global optimality in the Euclidean sense. We nevertheless compare reported times from some of these works with our approach (Table 3.2) and show that in addition to global optimality our method is also faster in most cases. The method based on the quad-tree scheme [15] can be faster but since it searches on a coarse space the produced paths can be very distant from an optimal.

GPU methods for performing grid-based search have also been explored, including a GPU implementation of randomized A* search called R*GPU [21], where instead of one state

being explored at a time, multiple short-range searches are performed in parallel based on a graph of sparsely placed states connected by edges and guided by a heuristic function; and parallelized wavefront expansion approaches with expansions per-pixel on uniform grids [26] and based on a quad-tree scheme [15].

Additional GPU methods have also been proposed to compute related structures. Delaunay triangulations have been computed in the GPU by mapping sites to a texture, computing the Voronoi diagram of the sites, deriving from it a triangulation that approximates the Delaunay triangulation, then repairing and flipping edges to obtain the final Delaunay triangulation [47]. Local collision avoidance has also been addressed in the GPU by applying the notion of velocity obstacles and computing the motion for many agents in parallel using a discrete optimization method [18]. Finallly, GPU methods for computing shortest path trees on graphs have also been proposed based on Dijkstra's algorithm and contraction hierarchies (CH) [9]; the graph, distance labels, and CH search space are copied to the GPU, where each vertex of the graph search can then be processed in parallel in its own thread.

2.5 Shader Programming

With respect to previous work in shader programming techniques, our work mainly relies on the use of shadow volumes. For a thorough overview of real-time shadow algorithms, we refer the reader to [11]. Algorithms for rendering hard shadows in real time can be roughly sorted into three categories: shadow mapping [66, 29, 33], alias-free shadow maps [1, 24, 52], and shadow volumes [14, 20]. The method used in our work relies on the generation of shadow volumes with a variation of the *z-fail* approach [6, 4, 12]. This approach is known to be slower than methods using shadow maps, due to higher overdraw and the need for near and far clipping geometry [28]; however, it does not suffer from (quite severe) resolution aliasing artifacts, and it is not camera-dependent, allowing in a single pass to generate shadows in every scene directions.

CHAPTER 3

Optimal Path Maps on the GPU

3.1 Introduction

This chapter presents our GPU-based method for the computation of optimal path maps which allow for the efficient extraction of optimal trajectories for agents in virtual environments. While several algorithms exist for computing shortest path maps, available methods are either too complex for practical use or too expensive for real-time applications. The proposed GPU computation approach greatly simplifies the process of building SPMs, allowing them to be easily computed with rasterization procedures triggered from OpenGL shaders without any pre-computation needed.

We call our maps Optimal Path Maps (OPMs) because they contain all of the functionality of SPMs and in addition address important extensions: maps with multiple sources of different types and maps representing velocity changes at vertices. See Figures 3.1 and 3.6 for examples.

Our approach introduces several advantages. While most representations require a point localization technique in order to determine the region containing the query point, in the proposed approach point localization is reduced to a simple constant time grid buffer mapping. After this mapping, since every point in the OPM has direct access to its parent point along the shortest path to the closest source, agents have direct access to the next point to aim when executing their trajectories. In addition, if the entire shortest path is needed, it

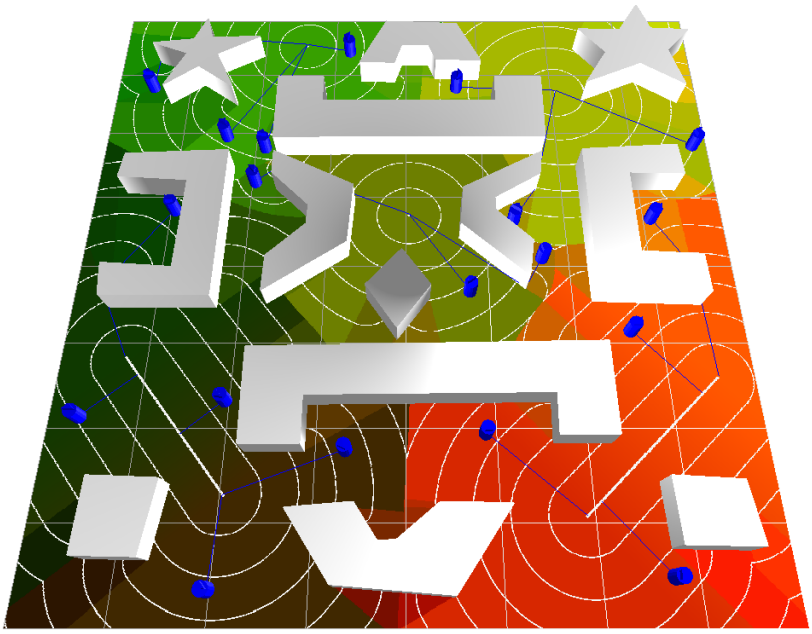


Figure 3.1: Example of a multi-source Optimal Path Map computed on a polygonal scene with obstacles. There are three source points in the upper half of the scene, and two line segment sources in the lower half. The contour lines represent equal distances to their closest source. Contour lines are directly extracted from a distance field which is stored in the Z-Buffer as a result of our method. The blue cylinders are agents and each has a polygonal line representing its shortest path to the closest source.

can be retrieved only in linear time with respect to the number of points in the shortest path.

Our approach is based on the idea of cone rasterization from sources and obstacle vertices. Unlike our initial work in this area [5], in the present method we do not require pre-computation of the shortest path tree of the environment and we also do not need to create any geometry for the rasterized cones. Instead we use dedicated fragment shaders to simply fill in the pixels that have direct line-of-sight to the vertices, improving computation speed and also eliminating errors that were introduced from discretizing cone geometry into triangles.

Our shaders operate on the original coordinates of the input vertices for all distance computations, therefore, when the buffer resolution is appropriate, our maps produce exact results not affected by the grid resolution. In addition, our new approach allows us to introduce a new type of map not addressed before: maps with weights at vertices, which allow accounting for speed changes at vertices, an interesting situation leading to new types of optimal path maps.

Our method can produce relatively complex dynamically-changing OPMs at real-time rates. Several examples and benchmarks are presented which demonstrate the various unique applications of OPMs and that in many cases our paths can be computed faster than competing approaches.

3.2 Related Work

The first proposed method based on Shortest Path Maps (SPMs) has worst-case time complexity of $\mathcal{O}(kn\log^2 n)$ [37], where k is a parameter called the "illumination depth", which is bounded above by the number of different obstacles touching a shortest path. Later, the first worst-case sub-quadratic algorithm for Euclidean shortest paths was proposed applying the continuous Dijkstra expansion, which naturally leads to the construction of SPMs [40]. The continuous Dijkstra technique simulates expanding wavefronts, which are the set of all points equally distant from a given source point. The expansion requires solving various events such as wavefront self-collisions forming hyperbolic boundaries. The result of the wavefront propagation is a spatial partition which is the SPM.

A nearly optimal algorithm for computing SPMs has been proposed taking optimal $\mathcal{O}(n\log n)$ time to preprocess the environment, allowing distance-to-source queries to be answered in $\mathcal{O}(\log n)$ time, and paths to be returned in $\mathcal{O}(\log n + k)$ time, where k is the number of turns along the path [22]. Unfortunately, these methods and all the known algorithms with good theoretical running times involve complex techniques and data structures that over-

burden their practical implementation in applications and prevent the development of useful extensions. In contrast, our GPU-based approach is relatively simple and is less affected by typical robustness difficulties encountered in many geometric computations for building spatial subdivisions relying only on floating point operations.

Alternative GPU approaches have also been explored in previous work. The first attempt to compute SPMs in GPU was designed to take advantage of the GPU's massive parallelization capabilities [67]. The method first pre-computes in CPU the visibility graph and the shortest path tree (SPT) of the environment. Afterwards, a brute-force but parallelized GPU computation is used to determine the closest SPT point to each pixel in order to produce a subdivision of the discrete screen space in SPM regions.

The idea of using shader rasterization as an efficient way to propagate wavefronts in the GPU was introduced in our previous work [5] and in this work we present a completely re-designed method incorporating several extensions and significantly improving the approach in multiple ways: 1) we eliminate the need to precompute the visibility graph and SPT, 2) in doing so we are able to address segment sources and speed changes at vertices, and 3) we no longer have to construct actual geometry for the rendered cones simulating wavefront expansions; instead we simply employ a dedicated fragment shader to directly fill in the relevant pixels, simplifying the process and most importantly eliminating error accumulation from cone discretization.

3.2.1 Contributions

The proposed method is the first SPM generation method to be implemented entirely with GPU shaders. It does not require any pre-computation, it addresses new capabilities not explored by previous navigation representations, and it enables multi-agent navigation based on paths with global optimality, a characteristic which has been neglected in simulated virtual environments developed to date. While advanced related methods in the geometry

processing area are available, they have not been applied to represent SPM boundaries or to represent paths with speed changes. To our knowledge our work has produced SPM diagrams of complexity not seen before in previous work.

3.3 Multi-Source Optimal Path Maps

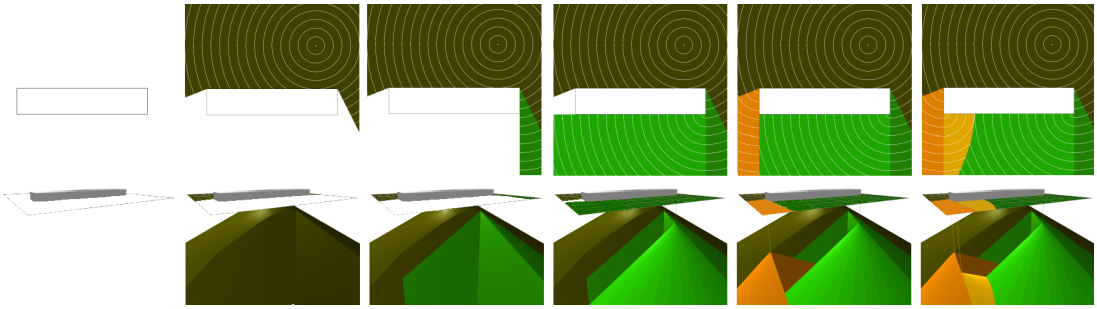


Figure 3.2: Top row: example steps for computing a single-source SPM. Bottom row: corresponding 3D perspective view of each step.

We first describe the base OPM case with multiple source points. Let n_s be source points $\{\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_{n_s}\}$ in the plane, such that $\mathbf{s}_i \in \mathcal{D}$, $i \in \{1, 2, ..., n_s\}$, and where $\mathcal{D} \subset \mathbb{R}^2$ defines a polygonal domain containing all sources. In all our examples $\mathcal{D}$ is a rectangular area delimiting the environment of interest, and the GPU framebuffer will be configured to entirely cover $\mathcal{D}$. A set of polygonal obstacles $\mathcal{O}$, with a total of n vertices, is also defined in $\mathcal{D}$ such that shortest paths will not cross any obstacles in $\mathcal{O}$.

Given source points the respective OPM will efficiently represent globally-shortest paths $\pi^*(\mathbf{p})$, which are optimal collision-free paths from any point $\mathbf{p} \in \mathcal{D} - \mathcal{O}$ to its closest source point $\mathbf{s}_i$, in the sense that $\mathbf{s}_i = min_j \lambda^*(\mathbf{p}, \mathbf{s}_j)$, where $\lambda^*(\mathbf{p}, \mathbf{s}_j)$ denotes the length of the shortest path $\pi^*(\mathbf{p}, \mathbf{s}_j)$, $j \in \{1, 2, ..., n_s\}$. Our OPM also efficiently represents the values of λ^* for all pixels of the framebuffer by storing them in a dedicated buffer created in the OpenGL pipeline. This representation gives us direct access to the distance field of the environment and allows us to easily draw the white isolines that can be seen in most of

the figures in this chapter. Depending on the situation source points can represent the start or the end point of a path. In most of the presented examples sources will represent goals to be reached by agents placed anywhere in the environment.

The plane represented by the framebuffer is located at $z = 0$. The basic idea of our method is to rasterize "clipped cones" with apices placed below source points and obstacle vertices, at the correct z heights, so that the final rendered result from an orthographic top-down view is the desired OPM (see Figure 3.2).

The process is implemented as follows. An array containing the n_s source points and n obstacle vertices is stored in the GPU. At each iteration one point or vertex is copied into a reserved position of a data array where it will be used to rasterize a clipped cone. The point or vertex that is selected to generate the clipped cone at each iteration is referred to as that iteration's "generator." Each point and vertex is processed once, such that the result is given after $n_s + n$ iterations.

Important to our approach is the fact that we do not actually need to create discretized geometry for representing and then drawing cones. Instead we simply fill in pixels that have direct line-of-sight to the generator, which is an equivalent operation. A cone apex is located below the generator relative to the $z = 0$ plane. The depth values of the affected pixels increase proportionally to their Euclidean distances to the apex, as with the slope of a cone. Because the depth is accumulated over iterations, it represents the distance back to the source point along the shortest path, λ^*. When all clipped cones are drawn at their respective heights, the GPU's depth test will maintain, for each pixel, the correct parent generator point, which is the immediate next point on the shortest path from that pixel to the closest source point. We say that a cone "loses" to another at a given pixel when its depth is greater, leading it to be discarded in favor of the "closer" cone.

3.3.1 Method Description

Given polygonal obstacles $\mathcal{O}$ with n vertices and n_s source points, $n_s \geq 1$, the total number of vertices to be processed is $n_{total} = n + n_s$. These vertices are stored in array DATAARRAY of size $n_{total} + 1$. The extra position is reserved for storing at each iteration the current generator that will be used for cone rasterization. By convention this is the first position in the array, DATAARRAY[0], and will be referred to as $\mathbf{g}_{cur}$. Once DATAARRAY is constructed, it is stored in the GPU as a Shader Storage Buffer Object. Each of the $n_{total} + 1$ positions in DATAARRAY stores:

• x, y : The original coordinates of the point or vertex in $\mathcal{D}$.

• STATUS : A flag that can be equal to SOURCE for sources, OBSTACLE for obstacle vertices, or EXPANDED for points or vertices which have already generated a cone.

• DISTANCE : The current known shortest path distance to the closest source point, λ^*. This will always be 0 for source points and is initially undetermined for obstacle vertices.

• PARENTID : Array index into DATAARRAY of the current parent point, which is the next point on the shortest path back to the closest source point. Since sources have no parent point, by convention they simply store their own index.

The framebuffer stores similar information for the pixels. For each pixel, its red and green components store the x and y coordinates of its parent point (equivalent to DATAARRAY[PARENTID].xy), its blue component stores λ^* (equivalent to DISTANCE), and its alpha channel stores either 0 if the pixel has yet to be reached by a cone or >0 otherwise. When the buffer is drawn, the color of each pixel is mapped in the following way: x is used as the red component, y is used as the green component, and the blue component is zeroed. Although this mapping is arbitrary, it allows to visualize the location of a region's parent from the red and green intensities.

The OPM generation consists of four steps which repeat n_{total} times such that each point and vertex is processed once. The steps are presented in Procedures 1-4. The hat notation

(e.g., $\widehat{\mathbf{n}}$) denotes unit vectors.

Step 1 is a search in DATAARRAY where the position with the smallest DISTANCE is copied into the reserved position of the array, index 0. Only points or vertices which have not yet generated a cone (STATUS $\neq$ EXPANDED) are considered in this search, and once one is chosen its status is updated to EXPANDED so that it cannot be processed again. The point that is chosen becomes $\mathbf{g}_{cur}$, the current generator. This step can be skipped in the first iteration of the algorithm as we can just start with one of the source points.

Procedure 1 Search Compute Shader

Input: DATAARRAY

1: int $generatorId \leftarrow -1$
2: float $generatorDist \leftarrow \infty$
3: **for** $\forall i, i \in 1, 2, ..., n_{total}$ **do**
4: **if** DATAARRAY[i].STATUS $\neq$ EXPANDED **then**
5: **if** DATAARRAY[i].STATUS = SOURCE **or** ($generatorId = -1$ **or** DATAARRAY[i].DISTANCE $<$ $generatorDist$) **then**
6: $generatorId \leftarrow i$
7: $generatorDist \leftarrow$ DATAARRAY[i].DISTANCE
8: **end if**
9: **end if**
10: **end for**
11: DATAARRAY[0] $\leftarrow$ DATAARRAY[$generatorId$]
12: DATAARRAY[$generatorId$].STATUS $\leftarrow$ EXPANDED

Step 2 is to generate a shadow area in order to solve visibility constraints. Using a geometry shader, we draw into a stencil buffer three triangles behind every obstacle line segment that is front-facing with respect to $\mathbf{g}_{cur}$, in a manner illustrated in Figure 3.3. Any pixel covered by one of these triangles is considered to be in shadow. The resulting buffer is used as a

stencil buffer in the next step. Three triangles is the minimum number of triangles needed to cover all possible shadow shapes. We use constant $c_{svf} > 0$, which stands for shadow vector factor, when computing the points that make up the triangles. This constant must be large enough to handle shadows of all sizes. Since our coordinates are OpenGL normalized coordinates in the $[-1, 1]$ range, a value of 4 is always enough. Note that limiting shadows to front-facing segments is merely for efficiency; generating triangles behind back-facing segments would not affect the shadow area.

Step 3 draws a clipped cone with the generator $\mathbf{g}_{cur}$ directly above its apex along the z axis. As previously stated, we do not actually create geometry for the cone but instead simply run a fragment shader over every pixel on the screen. The pixels that are not in shadow have direct line-of-sight to $\mathbf{g}_{cur}$, so they calculate their Euclidean distance to $\mathbf{g}_{cur}$ and add it to $\mathbf{g}_{cur}$'s accumulated distance, DISTANCE. If this sum is smaller than the current DISTANCE of the pixel (from the cone of a previous $\mathbf{g}_{cur}$), then its DISTANCE is updated and its PARENTID is set to $\mathbf{g}_{cur}$'s index.

Finally, step 4 is to update the DISTANCE of all vertices visible from the current generator, in a way similar to step 3. Each vertex not in shadow calculates its distance to $\mathbf{g}_{cur}$ plus $\mathbf{g}_{cur}$'s DISTANCE, and if that sum is smaller than its previous DISTANCE it stores the new DISTANCE and $\mathbf{g}_{cur}$'s index in its PARENTID. The reason steps 3 and 4 are separate is because step 3 is updating the framebuffer, while step 4 is updating the DATAARRAY. The end of this step is a synchronization point in our GPU implementation.

After all points and vertices have been processed, which means n_{total} iterations of steps 1-4, the result in the framebuffer will be the desired OPM. Examples of OPMs with a single source point are shown in Figure 3.17 and with multiple source points are shown in Figure 3.18.

Procedure 2 Shadow Area Geometry Shader

Input: DATAARRAY

Input: $\mathbf{g_{cur}}$ {Current generator point/vertex}

Input: e {One of the sides of a scene obstacle}

1: vec4 $\mathbf{p_1} \leftarrow$ first endpoint of e

2: vec4 $\mathbf{p_2} \leftarrow$ second endpoint of e

3: vec4 $\mathbf{p_m} \leftarrow (\mathbf{p_1}+\mathbf{p_2})/2$

4: vec4 $\mathbf{p_g} \leftarrow$ project and normalize vec4($\mathbf{g_{cur}}.xy$, 0, 0)

5: float $dx \leftarrow \mathbf{p_2}.x - \mathbf{p_1}.x$

6: float $dy \leftarrow \mathbf{p_2}.y - \mathbf{p_1}.y$

7: vec4 $\widehat{\mathbf{g}} \leftarrow$ normalize($\mathbf{p_m} - \mathbf{p_g}$)

8: vec4 $\widehat{\mathbf{n}} \leftarrow$ normalize(vec4($dy, -dx, 0, 0$))

9: float $d \leftarrow$ dot($\widehat{\mathbf{g}}$, $\widehat{\mathbf{n}}$)

10: **if** $d < 0.1$ **then**

11: vec4 $\widehat{\mathbf{v}}_1 \leftarrow$ normalize($\mathbf{p_1} - \mathbf{p_g}$)

12: vec4 $\widehat{\mathbf{v}}_2 \leftarrow$ normalize($\mathbf{p_2} - \mathbf{p_g}$)

13: vec4 $\mathbf{p_{1s}} \leftarrow \mathbf{p_1} + c_{svf}\widehat{\mathbf{v}}_1$

14: vec4 $\mathbf{p_{2s}} \leftarrow \mathbf{p_2} + c_{svf}\widehat{\mathbf{v}}_2$

15: vec4 $\mathbf{p_{ms}} \leftarrow \mathbf{p_m} + c_{svf}\widehat{\mathbf{g}}$

16: EmitPrimitive($\mathbf{p_1}$, $\mathbf{p_{ms}}$, $\mathbf{p_{1s}}$)

17: EmitPrimitive($\mathbf{p_2}$, $\mathbf{p_{2s}}$, $\mathbf{p_{ms}}$)

18: EmitPrimitive($\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_{ms}}$)

19: **end if**

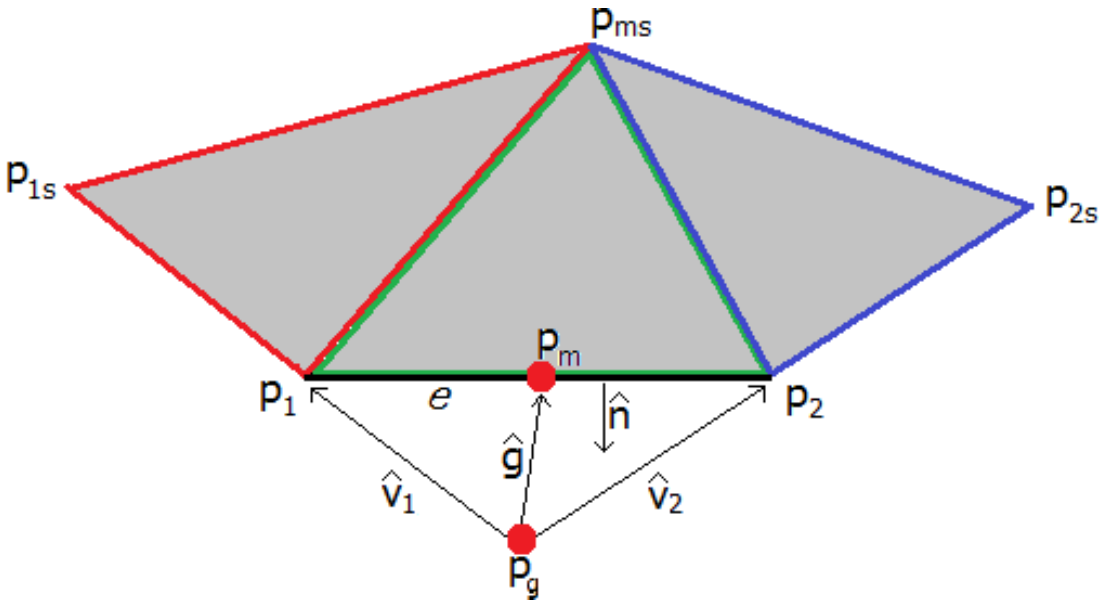


Figure 3.3: Example of a shadow area. The line segment e represents the side of an obstacle. The red point $\mathbf{p_g}$ is the generator, points $\mathbf{p_1}$ and $\mathbf{p_2}$ are the endpoints of e, and point $\mathbf{p_m}$ is the middle point of e. Vectors $\widehat{\mathbf{v}}_1$, $\widehat{\mathbf{v}}_2$, and $\widehat{\mathbf{g}}$ are the normalized vectors from $\mathbf{p_g}$ to $\mathbf{p_1}$, $\mathbf{p_g}$ to $\mathbf{p_2}$, and $\mathbf{p_g}$ to $\mathbf{p_m}$, respectively. Points $\mathbf{p_{1s}}$, $\mathbf{p_{2s}}$, and $\mathbf{p_{ms}}$ are calculated in the following way: $\mathbf{p_{1s}} = \mathbf{p_1} + c_{svf}\widehat{\mathbf{v}}_1$, $\mathbf{p_{2s}} = \mathbf{p_2} + c_{svf}\widehat{\mathbf{v}}_2$, and $\mathbf{p_{ms}} = \mathbf{p_m} + c_{svf}\widehat{\mathbf{g}}$. The three triangles are sufficient to cover the entire area behind the segment. Using less than three triangles may not result in a correct shadow if the generator is close to the segment because the area becomes wide and thin. Value 4 is used for constant c_{svf} such that shadows of any size can be handled given that our obstacle coordinates are normalized.

Procedure 3 Cone Fragment Shader

Input: DATAARRAY

Input: $\mathbf{g_{cur}}$ {Current generator point/vertex}

Input: $fragCoord$ {xy coordinates of the pixel}

Output: vec4 $fragValue$

1: bool $inShadow \leftarrow$ is the pixel in shadow or not?
2: vec4 $currentValue \leftarrow$ what's currently stored in this pixel {Texture fetch}
3: vec4 $fragValue \leftarrow currentValue$ {If nothing else, pass the current value on}
4: **if** $inShadow$ = false **then**
5: vec2 $\mathbf{p} \leftarrow$ normalize $fragCoord$
6: vec2 $\mathbf{p_g} \leftarrow$ project and normalize $\mathbf{g_{cur}}.xy$
7: float $newDist \leftarrow$ distance($\mathbf{p}$, $\mathbf{p_g}$) + $\mathbf{g_{cur}}$.DISTANCE
8: **if** there is no currently stored distance in the pixel **or** $newDist < currentValue.z$ **then**
9: $fragValue \leftarrow$ vec4($\mathbf{g_{cur}}.xy, newDist, 1$)
10: **end if**
11: **end if**

Procedure 4 Distance Compute Shader

Input: DATAARRAY

Input: $\mathbf{g_{cur}}$ {Current generator point/vertex}

1: int $id \leftarrow$ index of the vertex to be updated
2: bool $inShadow \leftarrow$ is the vertex in shadow or not?
3: **if** $inShadow$ = false **then**
4: vec2 $\mathbf{p} \leftarrow$ project and normalize DATAARRAY$[id].xy$
5: vec2 $\mathbf{p_g} \leftarrow$ project and normalize $\mathbf{g_{cur}}.xy$
6: float $newDist \leftarrow$ distance($\mathbf{p}$, $\mathbf{p_g}$) + $\mathbf{g_{cur}}$.DISTANCE
7: **if** there is no currently stored distance in DATAARRAY$[id]$ **or** $newDist <$ DATAARRAY$[id]$.DISTANCE **then**
8: DATAARRAY$[id]$.DISTANCE $\leftarrow newDist$
9: DATAARRAY$[id]$.PARENTID $\leftarrow$ $\mathbf{g_{cur}}$'s original index
10: **end if**
11: **end if**

3.4 Segment Sources

Line segment sources are one natural extension to our method, and are interesting as sources for what they can represent. Many navigation goals in real-world scenarios are not single points but segments, such as the finish line of a race, the thresholds of doorways or hallways, or the boundary of a coastline. Many of these cases appear when planning evacuation routes from buildings. Being able to compute OPMs with segments as sources allows us to maintain global optimality in these practical situations.

Consider that we now have n_l line segment sources $\{\mathbf{l}_1, \mathbf{l}_2, ..., \mathbf{l}_{n_l}\}$, such that $\mathbf{l}_i$, $i \in \{1, 2, ..., n_l\}$, consists of two endpoints $\in \mathcal{D}$. The OPM will then efficiently represent globally-shortest paths $\pi^*(\mathbf{p})$, which are now optimal collision-free paths from any point $\mathbf{p} \in \mathcal{D} - \mathcal{O}$ to the closest reachable point on a segment source $\mathbf{l}_i$.

Every line segment $\mathbf{l}_i$ can have n_{c_i} *critical points*, $n_{c_i} \geq 0$. A critical point denotes a point on the segment onto which at least one obstacle vertex projects. The obstacle vertex must have direct line-of-sight to the segment. Critical points are where the visibility of the scene changes with respect to the segment and are useful because in practice every path that passes through the corresponding obstacle vertex will have its shortest path reach the line segment on that critical point. See Figure 3.4. For each $\mathbf{l}_i$, first the two endpoints of the segment create two entries in DATAARRAY which are treated identically to source points. Then, $n_{c_i} + 1$ further entries are created, where n_{c_i} is equal to the number of critical points segment $\mathbf{l}_i$ possesses. Every one of these entries stores two pairs of xy coordinates rather than just one, with STATUS set to SOURCESEGMENT, to represent the sub-segments of $\mathbf{l}_i$. If $n_{c_i} = 0$, then the two endpoints are simply used because the segment has no sub-segments. If $n_{c_i} > 0$, then every adjacent pair of points, including both endpoints and critical points, will create an entry in DATAARRAY.

The distance calculation of the OPM generation process is different when the generator's

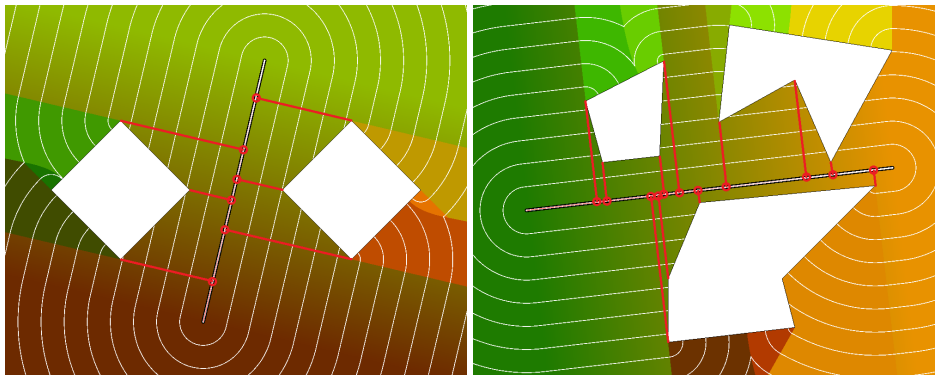


Figure 3.4: The circled points on the segment sources are the critical points, which are projections of obstacle vertices.

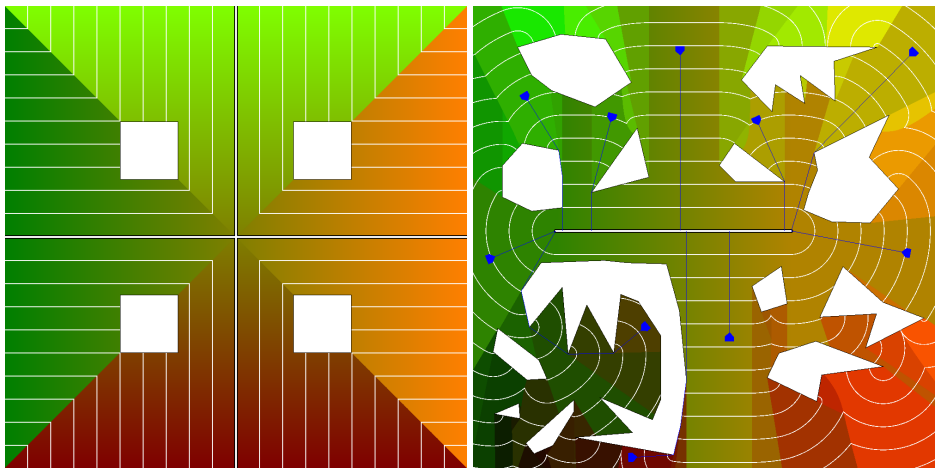


Figure 3.5: Line segment source examples. Left: SPM of two segment sources intersecting at the center. Right: Several paths from agents represented as blue triangles to their closest points in a segment source. In both cases the white contours represent the distance field from the sources.

STATUS is marked as SOURCESEGMENT. It is necessary to determine whether the point being updated is closer to one of the endpoints of the sub-segment, or somewhere inbetween. If it is closer to one of the endpoints, the distance is simply the distance to that endpoint. Otherwise, the distance is equal to the distance between the point and its projection on the sub-segment.

The described changes are sufficient to handle both segments and points as sources. Figure 3.5 shows additional examples of OPMs with line segment sources.

3.5 Vertex Weights

Another useful extension is to consider weights assigned to the vertices of the scene. A weight w on a vertex signifies that when an agent passes by the vertex its speed is changed according to w, implying that the distance calculation for that particular generator's cone will be altered by a certain multiplicative factor which is given by the value of w. This is the equivalent of changing the slope of the cone being rasterized, which is also equivalent to changing the speed at which that wavefront propagates.

As with segment sources, vertex weights allow our maps to represent practical scenarios that have not been explored previously. As an example, consider a virtual character that needs to arrive at a certain destination. One option is to walk directly there; another is to take a more roundabout path that at a certain point lets the character get on a bicycle or another vehicle, speeding up the traversal of the remaining distance. This scenario is illustrated in Figure 3.6. A shortest path map cannot answer which option is faster because it cannot represent the change in speed, but an optimal path map considering vertex weights can.

Given two points, $\mathbf{p}_i$ and $\mathbf{p}_j \in \mathcal{D} - \mathcal{O}$, the Euclidean distance between them, $d(\mathbf{p}_i,\mathbf{p}_j)$, and a weight w, $w > 0$, let the weighted distance be equal to $d_w(\mathbf{p}_i, \mathbf{p}_j) = d(\mathbf{p}_i, \mathbf{p}_j)/w$.

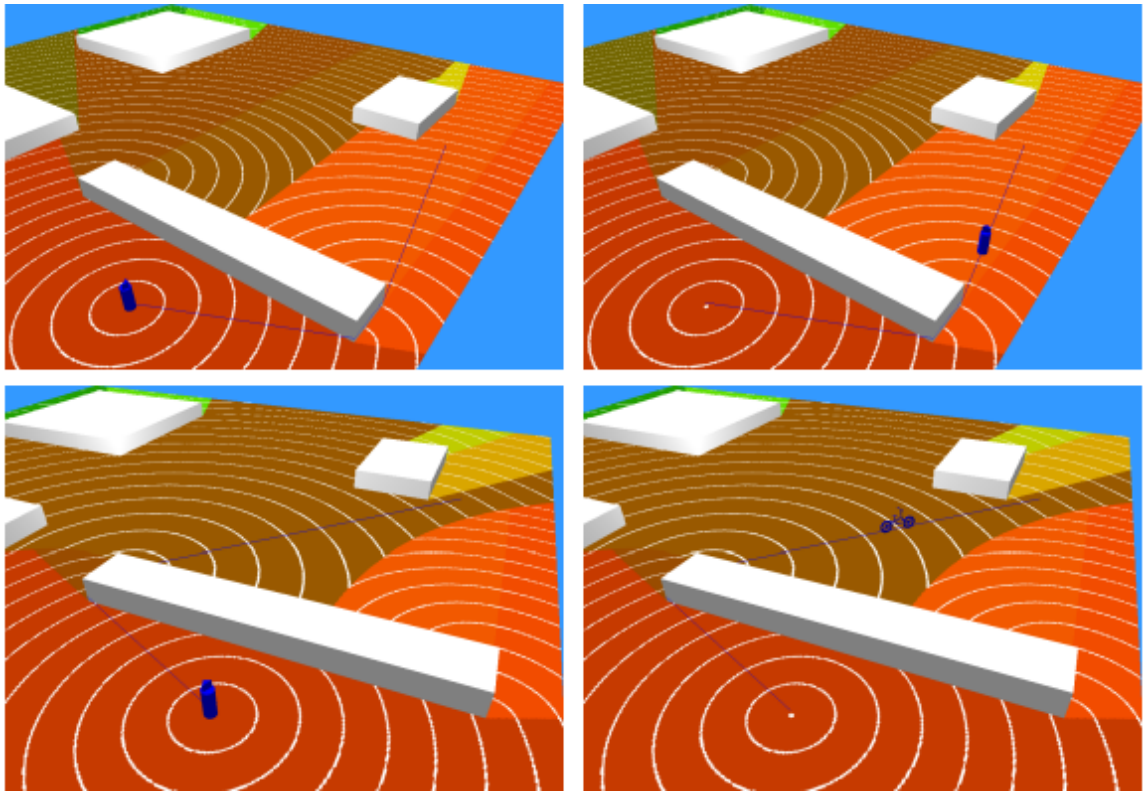


Figure 3.6: Top: an agent on foot plots a constant-speed shortest path. Bottom: the top-left vertex of the long rectangular obstacle has its weight increased representing the possibility of using a bicycle to speed-up traversal time. That possibility leads to the fastest path.

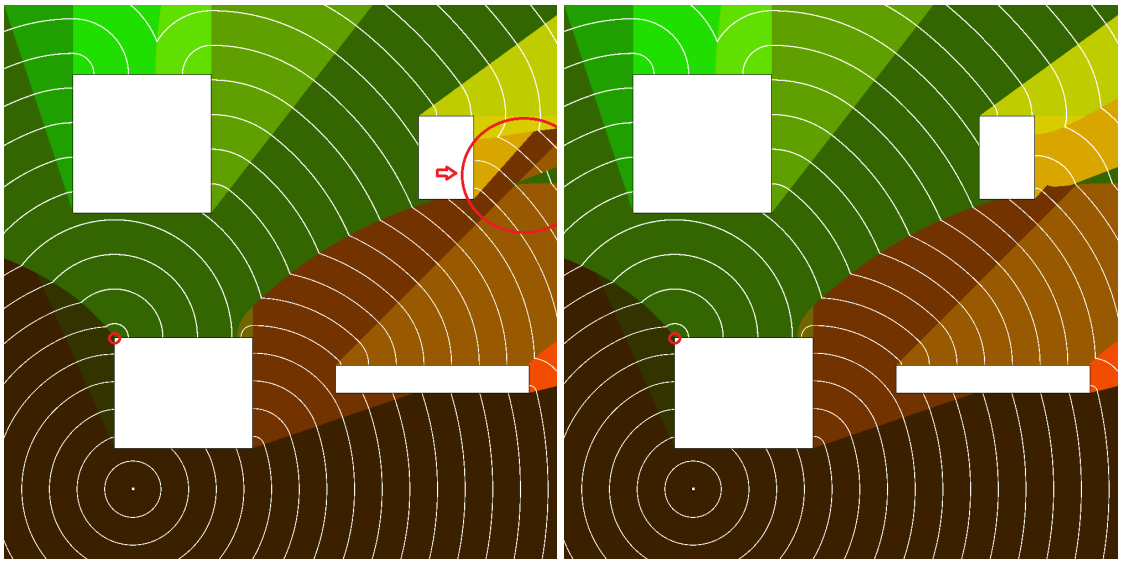
When all vertices have $w = 1$, a regular OPM is generated. When any vertex has $w \neq 1$, the OPM is altered. For example, if a vertex has $w = 2$, agents that pass through the vertex would move twice as fast. It represents the agent switching to a faster mode of travel.

If a generator with $w \neq 1$ becomes the parent of another vertex with $w' \neq 1$, then the weight stored in the vertex will be $max(w, w')$. This symbolizes the agent always preferring to stick with the fastest mode of travel that it comes across.

For every unique weight that exists in the scene, we must store an extra copy of each of the obstacle vertices in the data array. In a regular OPM it is impossible for a cone to lose to another in close distance but win over a long distance, so there is no need to propagate any but the closest cone for each vertex. In a weighted OPM this is however possible. A cone with a wider slope (higher w) may eventually poke out from under a cone with a narrower slope (lower w). This makes it necessary to propagate the closest cone for every unique weight, otherwise the resulting map may generate incorrect discontinuities.

In the example of Figure 3.7 the highlighted circled vertex (on the lower-left image quadrant) has a weight of 1.3, making it a more attractive option for optimal paths and thus distorting the OPM towards it. However, as can be seen in (a), it generates a discontinuity on the other side of the map (region highlighted with an arrow) because it was unable to propagate to the area behind the upper-right obstacle. By giving an extra space for the uniquely weighted vertex to propagate (b), an extra cone is drawn and the wavefronts line up correctly.

Interestingly the isolated green region, the one which is not adjacent to a generator vertex as indicated with the arrow in Figure 3.7, still appears in the correct version of the map. This indicates that if the agent is located in that region, the shortest path to the source first goes to the parent point of the green region, which is disconnected from the isolated region. Therefore OPMs with speed changes do not have anymore the property that each



(a) Incorrect (b) Correct

Figure 3.7: If generators with different weights are not allowed to propagate, they may generate inconsistencies in the OPM. It is necessary to store an extra copy of the data array for each uniquely weighted vertex in the scene, otherwise situations such as the above arise.

region associated with a parent generator is singly-connected.

The inclusion of vertex weights requires additional solutions for the correct visualization of the obtained OPMs. Consider the example shown in Figure 3.8. Although the selected goal points (red crosses) in both (a) and (b) have the same parent, the paths that they generate are not the same; one passes through the weighted vertex and the other does not. Here we use dashed lines to differentiate regions with paths altered by the weighted vertex. The inclusion of additional weighted vertices would require additional patterns in the visualization. Because weighted vertices change the speed at which an agent traverses, white isolines no longer denote equal distance to the source but rather equal time intervals. Figure 3.9 shows the resulting OPM when the weight of the same vertex as in Figure 3.7 is set to increasing values.

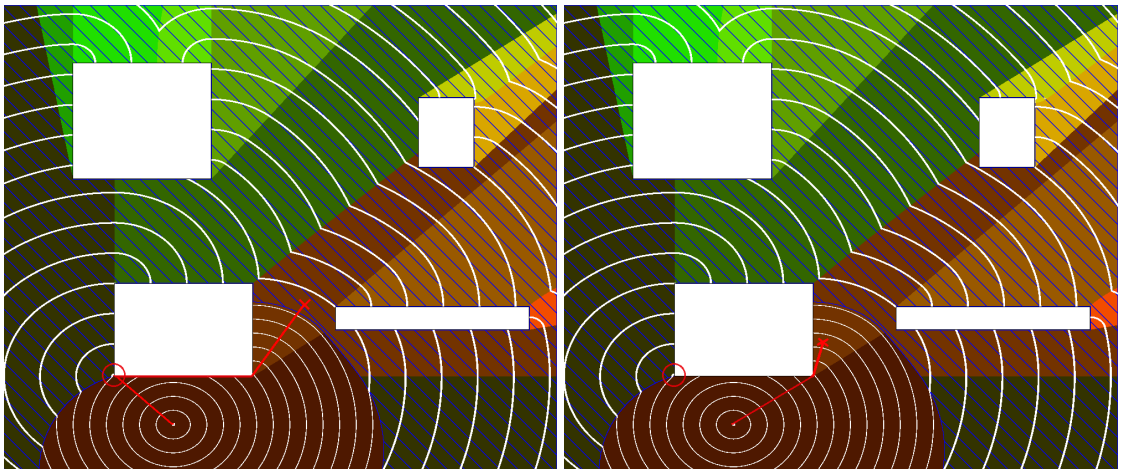
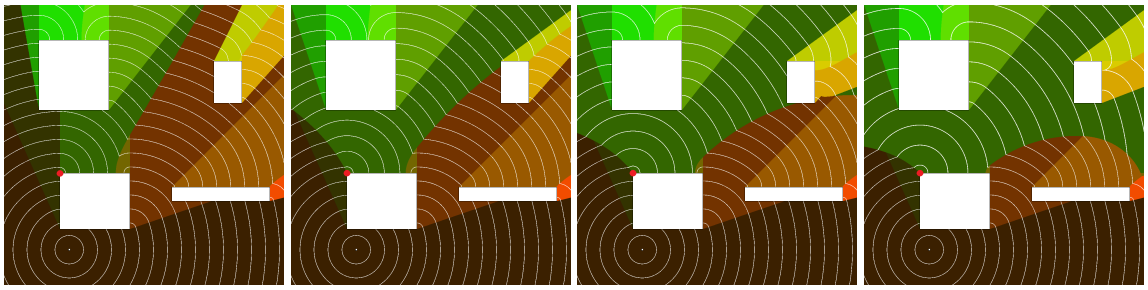


Figure 3.8: The weight of the highlighted vertex is increased. Paths to regions with blue dashed lines pass through the "faster vertex". With vertex weights, ambiguity is introduced because while pixels in the same region still have the same parent, their paths after the parent may be different. This happens because if the goal is sufficiently distant it becomes advantageous to take a detour to pass through the weighted vertex and gain faster travel mode.



(a) Weight 1 (b) Weight 1.1 (c) Weight 1.2 (d) Weight 1.3

Figure 3.9: Resulting OPMs as the highlighted vertex has its weight increased. The region generated by the clipped cone at that vertex gradually bloats outwards until it reaches both sides of the map.

3.6 Results and Discussion

We have produced several agent simulations taking advantage of the new capabilities introduced in this work.

Dynamically Changing Sources Figure 3.10 depicts the layout of a subway, with the sources symbolizing train doors which dynamically change from non-existent to points and then growing line segments as the doors open. The OPM is updated in real-time as this happens and the paths of the agents adjust accordingly.

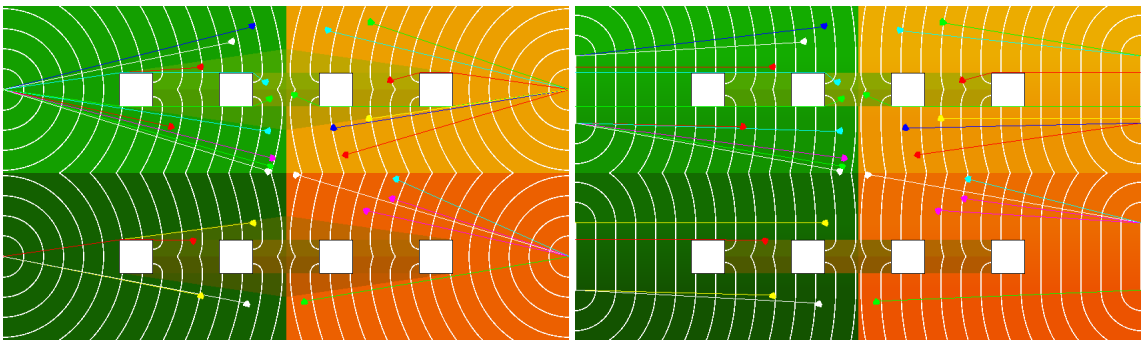


Figure 3.10: Simulation dynamically updating an OPM as its sources change from points (left) to growing line segments (right).

Moving Segment Sources We add motion to the segment sources in the simulation depicted in Figure 3.11. The segment sources represent dynamic goals (trains) that agents attempt to reach. The trains move either left or right on their tracks while the map is continuously updated.

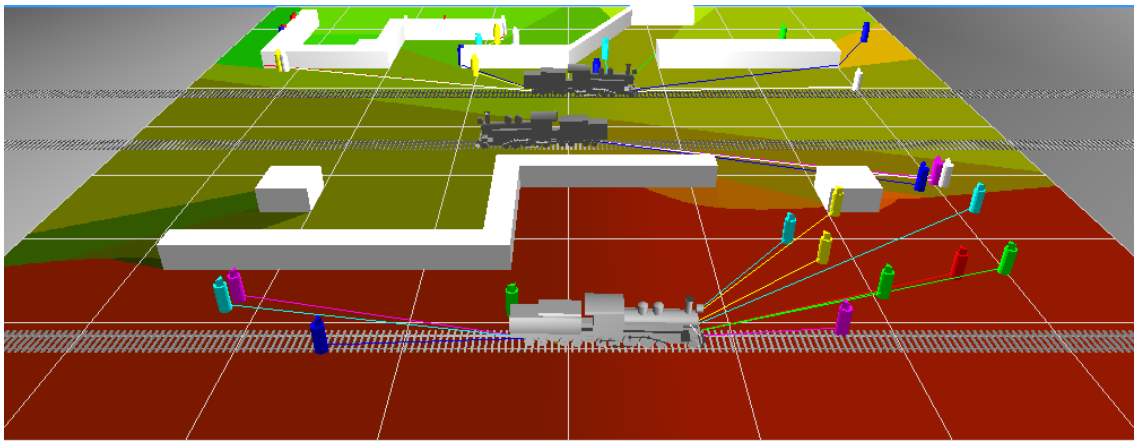


Figure 3.11: Simulation where agents attempt to reach moving trains represented by dynamic segment sources. As the trains move each agent has direct access to a shortest path to the closest train.

Evacuation Analysis An OPM is used in Figure 3.12 to calculate a distance field where a greener color indicates closer proximity to a source while a redder color indicates greater distance from a source. Sources are segments indicating road exits and the illustrated map is a region of the roads in the northwest area of Bodie, CA. Three segment sources represent the exits, one in the northwest area and two in the northeast area. By varying the number of passages leading out of the central area it is possible to visually analyze differences in evacuation distances and the OPM boundaries delimiting different directions towards closest exits. The encoded optimal paths are readily available for simulating autonomous agents.

Multiple Vertex Weights Multiple vertices in the environment in Figure 3.13 have had their weights increased due the availability of a faster transportation mode. Four agents with

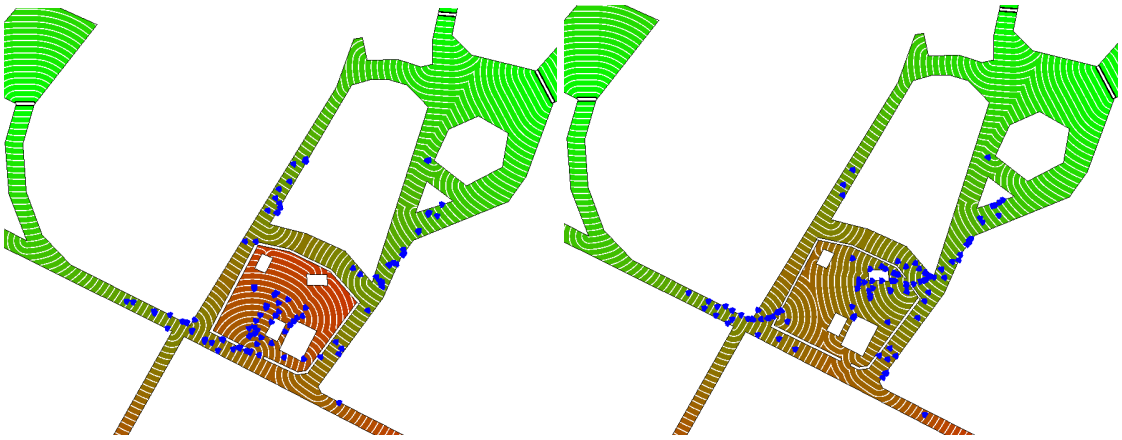


Figure 3.12: Evacuation simulation. Left: central area has 1 exit. Right: 3 exits.

identical start and end points navigate the environment, one at a time. Each time an agent passes through a weighted vertex it uses the transportation resource and the corresponding weight is reverted to regular, altering the OPM and resulting paths for subsequent agents.

Collision Avoidance Integration We have explored the approach of relying on SPMs with local collision avoidance for providing optimal paths for agents to navigate around obstacles towards their goals [50]. We have integrated our SPM implementation with the well-known Reciprocal Velocity Obstacle (RVO) [61] approach for local collision avoidance. As shown in Figures aa, when relying only on local behavior agents can easily become trapped, but this does not happen when agents are guided by the velocity vector determined by using the SPM of the environment.

3.6.1 Benchmarks

We evaluate the performance of our algorithm with several benchmarks where we use a framebuffer resolution of 1000x1000 on a Nvidia GeForce GTX 970 GPU and an Intel Core i7 3.40 GHz computer with 16GB of memory.

Table 3.1 shows average execution times for computing 100 single-source OPMs with random source points in $\mathcal{D} - \mathcal{O}$. The table shows times both with and without transferring the

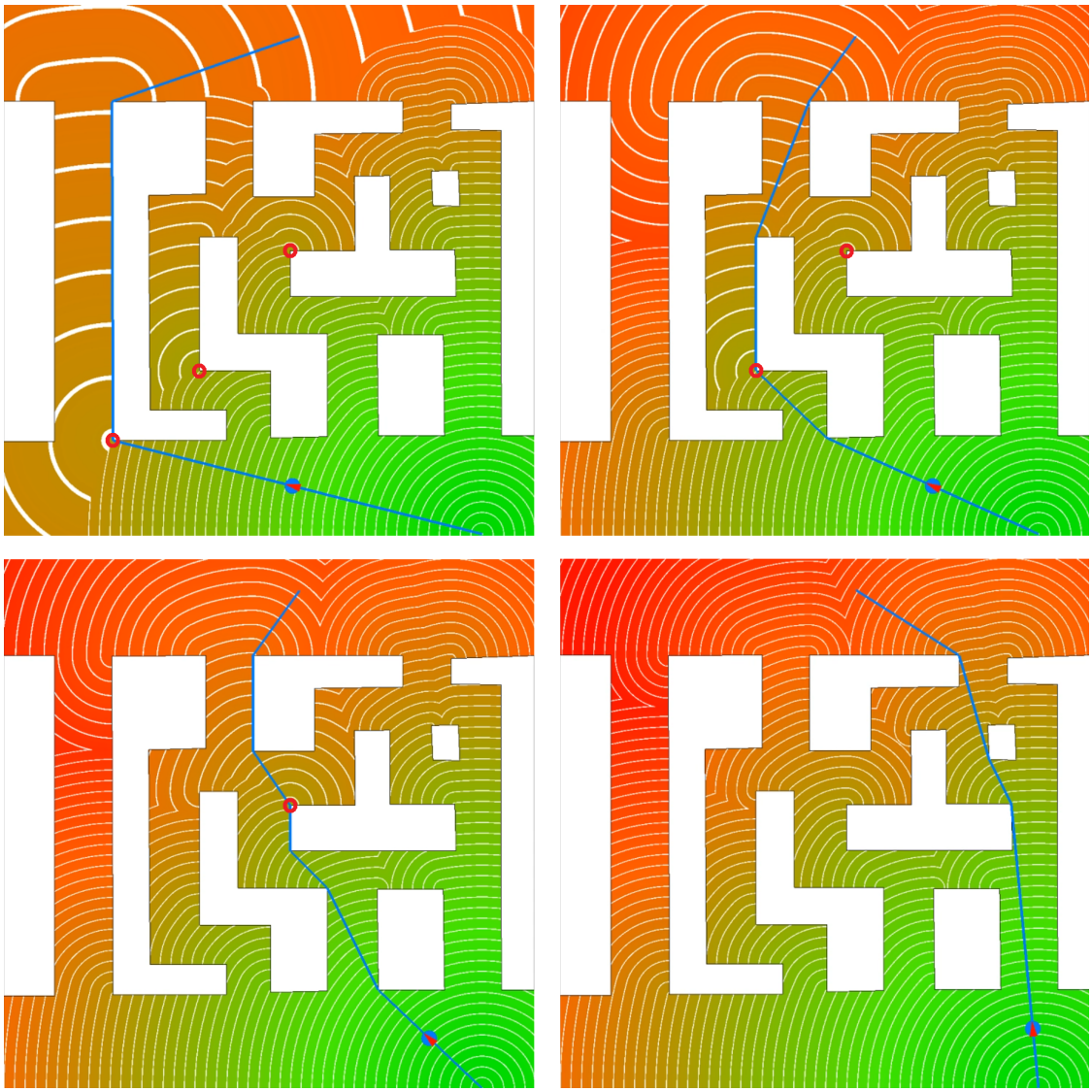


Figure 3.13: Simulation with multiple vertex weights.

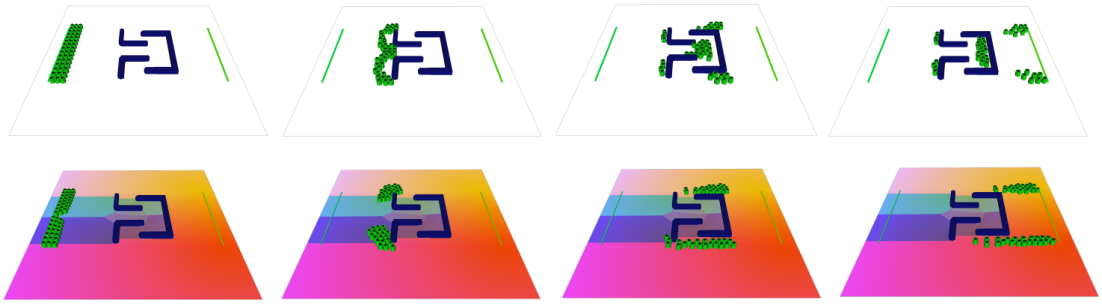


Figure 3.14: Agents are trapped when relying only on collision avoidance (top), but not when extracting directions from the SPM (bottom).

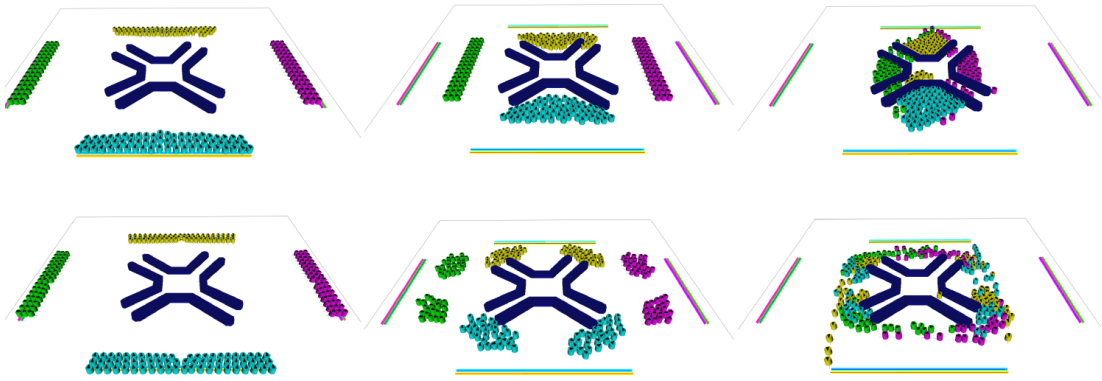


Figure 3.15: Top: Four groups of agents navigating toward their goal segments using RVO-based local behavior without any global path planning. Bottom: SPMs are applied to provide preferred velocity vectors aligned with shortest paths in order to guide the agents.

			Computation	Comp.+Transfer
Map name	**P**	**V**	**Time (s)**	**Time (s)**
Concave1	2	12	0.0011	0.0207
Concave2	13	96	0.0088	0.0465
Spiral	1	38	0.0022	0.0274
SpmEx1	3	15	0.0016	0.0215
SpmEx2	13	91	0.0100	0.0470
Profiling0	4	16	0.0014	0.0221
Profiling1	16	64	0.0054	0.0404
Profiling2	36	144	0.0456	0.0858
Profiling3	64	256	0.1251	0.1680
Profiling4	100	400	0.2701	0.3099
Profiling5	196	784	0.8564	0.8863
Profiling6	400	1600	2.7371	2.8070

Table 3.1: Average time in seconds to compute a single-source OPM on various maps (shown in Figure 3.17). P and V are the number of polygons and vertices.

Method	CPU	GPU	O	V	Optimality	Time (s)
DS (1)	–	GF GT 650M	–	–	Average	32.93
DS (1)	–	GF GTX 680	–	–	Average	21.25
DS (2)	–	–	–	–	Average	14.12
DS (2)	–	–	–	–	No	0.04
SPM (3)	i7 2.66 GHz	GF GTX 580	64	256	Best	1.42
OPM (4)	i7 3.40 GHz	GF GTX 970	64	256	Best	0.13

Table 3.2: A comparison of GPU-based techniques: (1) dynamic search using an uniform grid [Kapadia et al. 2013], (2) dynamic search using a quad-tree [Garcia et al. 2014], (3) brute-force SPM [Wynters 2013], and (4) our method. The number of obstacles (O) and vertices (V) in the environment are included for the last two methods. Some hardware details were not specified in the papers.

resulting OPM back to the host memory.

Figure 3.16 charts out computation times on the Profiling maps. These maps are composed of uniform rows of square obstacles (see Figure 3.17) with large visible areas from all points in the map. This represents a worst-case scenario for our method because there are large areas visible from all vertices. Still we observe that the increase in computation time is not too distant from linear, given the parallel execution of the GPU rasterization operations.

Table 3.2 shows that our method is able to compute an OPM and return optimal paths faster than some previous GPU-based methods which are grid-based and thus non-optimal. For example, Kapadia et al. [26] gives times to plan paths on a grid environment with similar resolution to the buffer used in our benchmarks, 1024x1024, as follows: between 32.931 and 49.126 seconds for a GT 650M and between 21.246 and 30.778 seconds for a GTX 680. While it would be disingenuous to directly compare these numbers to our benchmarks, which used a newer GTX 970, we nevertheless believe that a new card will

not offer the significant speed up that would be required to match even the 2.80 second running time we achieved on our most complicated map. In a later work they employed a quad-tree to speed up the computation [15], but sacrificing optimality even more in the process.

We have also performed experiments against a window propagation method used for computing geodesic paths on meshes. Table 3.3 shows a comparison between the average number of path queries per second our method can answer compared to the CPU method of Xin and Wang [68] available in CGAL. For both algorithms, one million points were randomly generated on the map and then used as query points. As can be seen from the table, in all cases tested our OPMs were able to answer a significantly larger number of path queries per second. This follows from the fact that point location is a trivial constant time operation in OPMs, and after that, paths are constructed by simple concatenation of parent points from the query point. The faster query time basically follows from the grid-based representation of our method. We found it difficult to compare construction times. Our method was slower in computing our per-pixel SPM representation than the time taken by the CGAL method to compute their sequence tree. However, a sequence tree only represents paths to the vertices of the obstacles, and trivially querying the structure to construct path information for every pixel would lead to slower times.

3.6.2 Discussion

Although our method uses a framebuffer grid and thus samples the environment at the level of pixels, distances are calculated exactly using the original coordinates of the sources and obstacle vertices. This means that there is no accumulation of error introduced by the method when integrating lengths of solution paths. In practice, only the region borders formed by collision fronts are affected by the pixel approximation since they decide the first parent point to take when starting a shortest path to the closest source. After the first parent point is selected, all the next ones are determined only from floating point

			Queries per second		
Map name	**F**	**AVP**	**CGAL**	**OPM**	**Improv.**
Concave1	18	2.80	1,530,456	9,678,293	6.3x
Concave2	124	3.47	723,589	7,958,298	11.0x
Spiral	42	6.48	1,152,206	5,435,787	4.7x
SpmEx1	23	2.68	1,301,066	9,669,309	7.4x
SpmEx2	119	3.24	720,928	8,278,968	11.5x
Profiling0	26	2.58	1,150,880	10,511,710	9.1x
Profiling1	98	2.95	704,423	9,273,255	13.2x
Profiling2	218	3.29	457,435	8,164,065	17.8x
Profiling3	386	3.55	373,985	7,815,736	20.9x
Profiling4	602	3.90	232,336	7,267,917	31.3x
Profiling5	1178	4.57	153,881	6,075,666	39.5x
Profiling6	2402	5.54	92,909	5,011,099	53.9x

Table 3.3: Number of path queries per second. F is the number of faces (triangles) on each map and AVP is the average number of vertices in the paths computed. The last column shows the improvement obtained with OPMs.

computations with the input vertices. If needed, it is however still possible to guarantee the correct shortest path for a given query point on a pixel bordering two (or more) regions; in that case we can test to see which of the neighboring regions' parent points are in fact the closest to the query point, using their exact accumulated distances to the closest source.

A suitable framebuffer grid resolution is expected to be chosen guaranteeing that every grid pixel contains at most one source point or obstacle vertex, and no free space exists between adjacent pairs of obstacle grid pixels or adjacent shadow regions. Under these conditions our method will provide correct minimum shortest paths, with optimality guaranteed up to the precision of half one pixel diagonal length for point queries on pixels at the boundaries of the OPM regions.

Besides being resolution-sensitive the main limitation of our method is that it may only be suitable for real-time simulations in environments of moderate size. Our method is slower than state-of-the-art path finding solutions that focus on speed of computation instead of global optimality [25]. However, our performance times have the potential to increase over time given the rapid expansion of GPU-based computing hardware and techniques.

3.7 Conclusions

We have introduced in this chapter a novel shader-based GPU method for computing optimal path maps addressing multiple types of sources and weights at vertices representing speed changes. We also uncover the interesting property that speed changes may lead to maps with disconnected regions associated to a same parent generator, something that cannot happen in traditional SPMs. The achieved capabilities have clear practical applications and were not explored before in an optimal way. Our benchmarks show that our method outperforms comparable approaches in many cases.

Our approach opens new directions for incorporating navigation mapping techniques within

the graphics pipeline. Our maps can instantly guide agents in multi-agent simulations from GPU buffers storing distances to the closest target and the next point to aim for from any position in the environment.

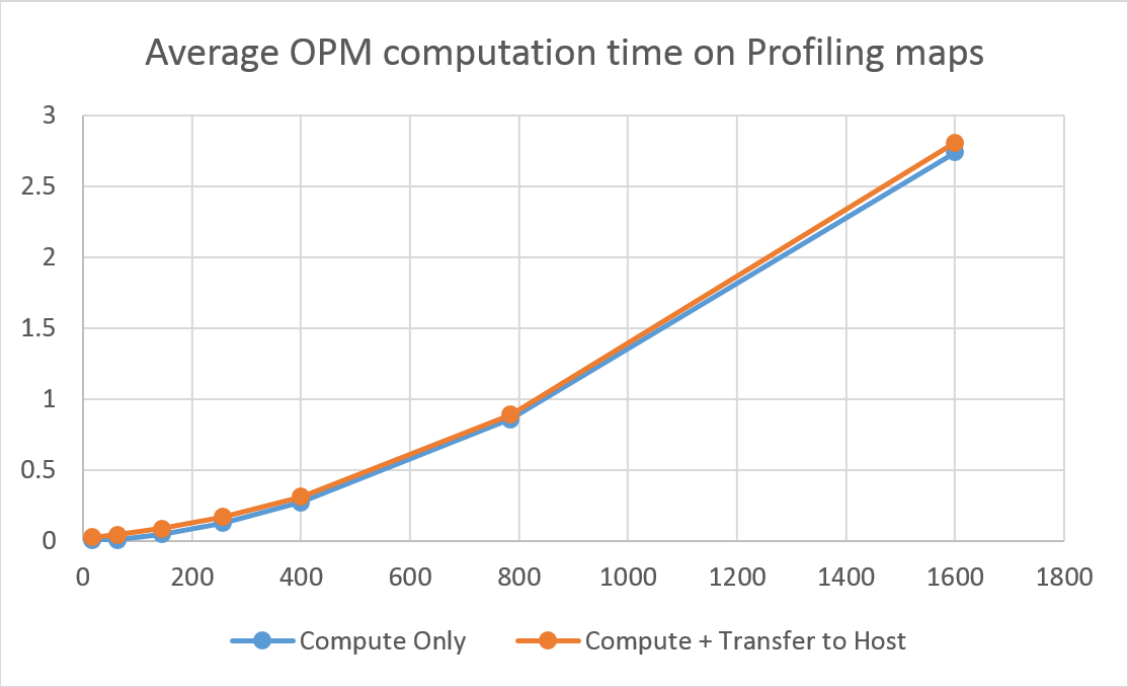
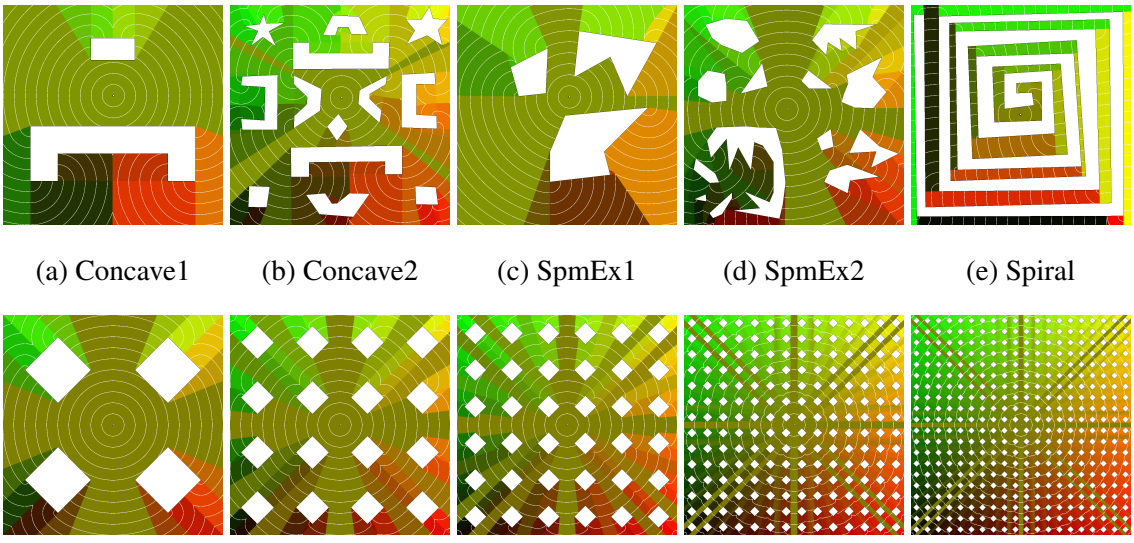


Figure 3.16: The x axis represents the number of obstacle vertices in the scene, and the y axis represents the computation time in seconds.



(a) Concave1 (b) Concave2 (c) SpmEx1 (d) SpmEx2 (e) Spiral

(f) Profiling0 (g) Profiling1 (h) Profiling2 (i) Profiling5 (j) Profiling6

Figure 3.17: Single-source OPM results.

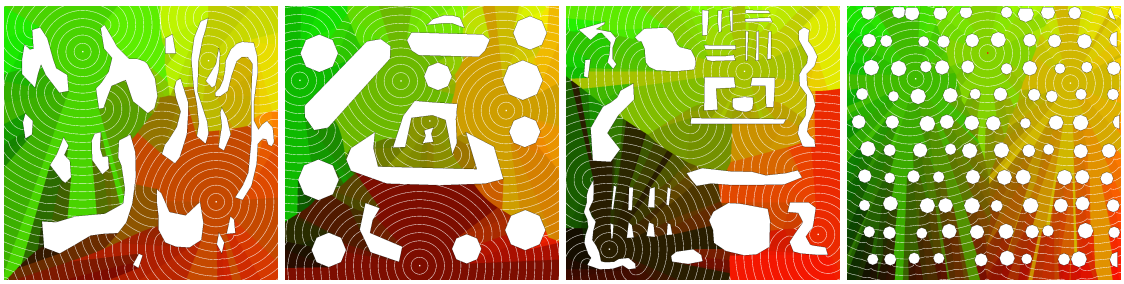


Figure 3.18: Multiple-source OPM results.

CHAPTER 4

Planar Max Flow Maps and Determination of Lanes with Clearance

4.1 Introduction

The problem of optimally deploying multiple agents traversing a polygonal environment has important applications in many areas, as for example, to control multiple robots in warehouses, to coordinate autonomous cars across narrow streets and to evaluate evacuation scenarios. While optimality can be defined by taking into account different parameters such as energy, time, or distance traveled, in all cases the problem is difficult to be solved in a planar domain and is usually addressed in a discrete representation of the environment.

In this chapter we present a method for agent deployment among obstacles based on computing the continuous maximum flow of a 2D environment. In this case we address computing solutions based on disjoint lanes which are optimal with respect to the maximum flow of agents traversing the environment. Our overall approach is based on GPU rasterization techniques which allow us to compute maximum flows from polygonal representations and to represent them in a *max flow map* discretized in a frame buffer in the GPU.

Additionally, we describe the *clearance-based max flow map*, which generates max flows specifically for achieving the maximum possible number of lanes with a given clearance. These maps achieve this property by incorporating during the flow generation a technique

suggested for generating *integer* max flows [39]. We therefore describe in this chapter two types of max flow maps: one insensitive to the clearance needed by agents, and another specific for a given clearance. We also include definitions for the types of flows that are represented in each case.

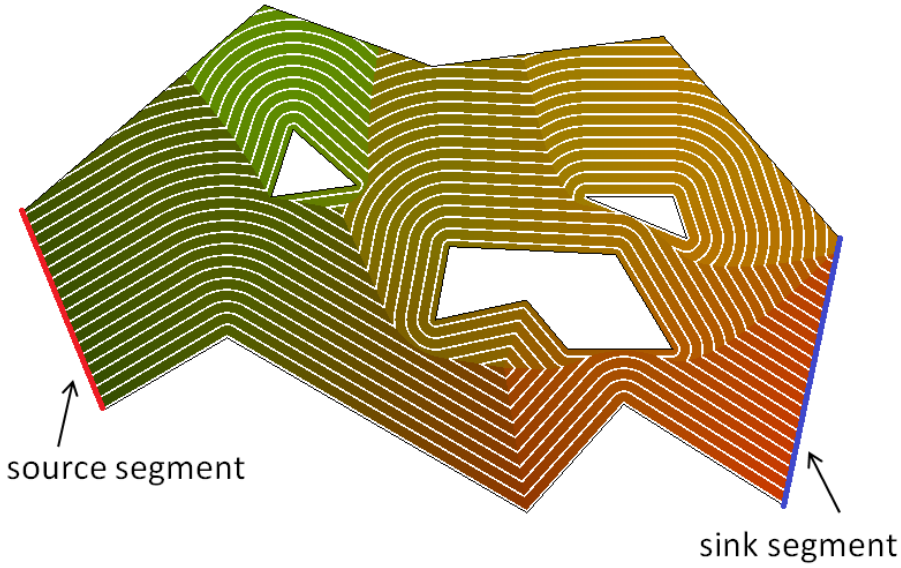


Figure 4.1: Example max flow map from a source edge to a sink edge. While this flow map has optimal flow capacity, path lanes are subsequently extracted taking into account the required agent clearance and then optimized in length.

Our methods produce bottleneck-free lanes which can be used to safely deploy and guide agents across a cluttered environment. If a large quantity of agents is deployed, the system of lanes will optimally guide all agents to reach the destination region. Here optimality is related to the maximum number of agents that can be deployed across the environment from a source polygonal entrance to a sink polygonal exit without creating bottlenecks. See Figure 4.1 for an example. Once a flow map is computed, lane trajectories are extracted according to the size of the agents, and optimized in length while keeping constant the maximum flow achieved by the system of trajectories.

As a result our methods are able to generate lanes of maximum flow from source to des-

tination edges among a generic set of polygonal obstacles. When the system of lanes is fully occupied by agents, no more agents can fit the environment without eventually creating bottlenecks. Agents can safely follow our computed lanes without having to employ any complex local behavior strategies in order to reach the destination region. In terms of length, our lanes are locally-optimal with respect to the total length of all lanes.

Simulations are also presented demonstrating the superior performance of our method in deploying large quantities of agents across environments with obstacles, when compared to having agents following their shortest paths to the destination.

4.2 Related Work

Our work develops a new approach to address multi-agent navigation in cluttered environments which is based on finding the maximum number of valid disjoint lanes that can be routed from an initial polygonal source segment to a polygonal target segment.

The addressed problem has some resemblance to the well-known k-disjoint shortest paths problem in graphs [3], which seeks to find the k pairwise disjoint shortest paths connecting given initial graph nodes to corresponding target graph nodes. Our formulation however is quite different in which it addresses a continuous polygonal environment, with polygonal edges considered as entrances and exits, and addressing required clearance constraints. Another difference is that there is no labeled target per agent, and instead agents can be routed to any point in the exit segment. Our proposed approach solves this problem by computing the continuous max flow of the input environment and then extracting paths from it. While our solution is of max flow, the total path length is optimized only locally.

The literature review below makes an overview of the more generic Multi-Agent Path Planning (MAPP) problem, analyzes previous uses of max flow algorithms in MAPP problems, and finally reviews the continuous max flow problem which is the approach taken in this

work.

4.2.1 Multi-Agent Path Planning

The Multi-Agent Path Planning (MAPP) problem is related to planning paths for agents from their initial positions to target positions. It is an important problem for a variety of applications and the problem has been extensively studied in different contexts, as discussed in Chapter 2.

One important characteristic of methods to solve the MAPP problem is that, in order to find a solution when one exists, time has to be discretized and the employed search procedure has to take into account the time component. Our approach however focuses on finding disjoint paths such that searching in the time component is not needed. While our approach can be seen to be less generic in the sense that it does not solve agent coordination to reach specific goal points, the proposed method addresses the spatial reasoning problem of maximizing the flow of agents across an environment, and is better suited for optimizing areas with high traffic of agents among obstacles. Once paths (or lanes) of maximum flow are computed, an arbitrary number of agents can be deployed by simply following the lanes in order to optimally traverse the environment.

4.2.2 Use of Flow Algorithms in Multi-Agent Path Planning

Discrete flow algorithms have clear applications to multi-agent path planning and the problem of computing maximum flows in a capacitated network graph has been an important problem in combinatorial optimization. The problem is commonly studied in textbooks and many polynomial-time algorithms exist. The connection between network flows and path planning has started to be investigated in a number of works; however, to date all previous works have been limited to investigations performed in discrete versions of the problem.

A Conflict-Based Min-Cost-Flow algorithm has been proposed to address the combined

target assignment and path finding problem, where a min-cost max-flow algorithm on a time-expanded network graph is used to assign all agents in a single team to targets [36]. Yu and Lavalle [72] study the problem of computing minimum last arrival time and minimum total distance solutions for multi-agent path planning on graphs. Their formulation relies on discrete multi-commodity flow algorithms which address the problem of flowing different types of commodities through a graph network. Heuristics for search-based algorithms that systematically explore the state space have also been proposed based on commodity flows [58, 27], and multi-agent path planning for goals that are permutation invariant has been addressed with graph network flows [71].

In the area of multi-agent simulation, crowd-flow graphs have been developed to distribute agents in an environment according to capacity information extracted from a harmonic field computed in the environment [2].

While these works clearly show that solving flow problems represents a powerful approach to address multi-agent path planning, previous work has used discrete max flow algorithms applied to a time-expanded representation. No previous work in multi-agent navigation has explored the use of a continuous flow formulation in order to directly generate lanes and at the same time address planar environments described by polygonal boundaries. Such an approach is important in order to reach optimality guarantees in the Euclidean sense, and furthermore, to take into account specific geometric constraints (such as agent size) without simplifications.

4.2.3 Continuous Max Flows

While the generalization of the maximum flow problem to a continuous domain is clearly interesting, its computation is not obvious. Strang [55] describes an extension of the max-flow min-cut theorem to continuous flows, showing that the maximum flow from sources to sinks in a planar domain is determined by the minimal cut, just like the discrete version

of the problem. This result opens a direction for computing max flows in continua.

Mitchell [38] addresses the problem of actually constructing the min cuts and max flows in a clever approach based on the computation of Shortest Path Maps (SPMs) [37]. While no implementations are presented, polynomial-time algorithms are given for varied max flow scenarios involving source edges and sink edges in simple polygons. Similar to the calculation of SPMs, a continuous Dijkstra paradigm forms the basis for the algorithms, but in a specific form which solves the so-called $0/1/\infty$-weighted regions problem. In this work we follow this approach in order to achieve our proposed flow maps.

While it is not straightforward to generate max flows and SPMs via CPU-based methods, our GPU-based techniques represent a practical approach to achieve implementations solving these problems. Our underlying GPU-based approach takes advantage of the built-in rasterization features of the OpenGL rendering pipeline in order to propagate costs during the construction of our maps. A complete description with additional details, extensions and benchmarks against other approaches for building SPMs was given in the previous chapter. We are not aware of any other implementations to compute maximum flows.

4.2.4 Contributions

This work proposes two main contributions. First, we introduce methods to compute maximum flow maps for polygonal domains relying on the insight of applying GPU rasterization techniques previously used for computing shortest path maps. We then address the new problem of extracting paths with clearance from max flows, presenting a specific flow construction method that takes into account clearance, and addressing methods for lane extraction and total length minimization.

4.3 Definitions and Overview

Let the input polygonal environment be delimited by a polygon $\mathcal{P}$ containing all obstacles of interest in its interior. The set of polygonal obstacles is denoted by $\mathcal{O}$. We are considering the situation where agents will enter $\mathcal{P}$ from given source edges P_{src} and exit the environment by crossing sink edges P_{snk}, while not colliding with any obstacles in $\mathcal{O}$. In our formulation P_{src} and P_{snk} are polygonal lines which are pieces of the boundary of the domain $\mathcal{P}$. We also consider that P_{src} and P_{snk} are connected polygonal lines.

Assuming P_{src} is left of P_{snk}, as in the example of Figure 4.1, there are two additional polygonal boundaries between P_{src} and P_{snk} which appear at the bottom and top of the domain. We call these additional polygonal lines as P_{bot} and P_{top}. In this case the concatenation of P_{src}, P_{bot}, P_{snk} and P_{top} completely covers the domain boundary in counter-clockwise order.

With source and sink edges defined it is possible to define the max flow problem in $\mathcal{P}$. A few variations on the definitions have been presented in the literature. Mitchell [38] defines the max flow problem as computing a vector field $\sigma : \mathcal{P} \rightarrow \mathbb{R}^2$ that maximizes $v = \int_{P_{snk}} \sigma \cdot \mathbf{n}\, ds$, subject to: div $\sigma = 0$ and $|\sigma| \leq c$ in $\mathcal{P}$.

In the above definition, vector $\mathbf{n}$ is the outward unit vector normal to P_{snk}, v is the value of the flow σ, and c is a capacity constraint function that can be defined to limit the magnitude of the vector field. Flow conservation comes from the divergence-free constraint, which also implies that flow-in equals flow-out. A variation of this definition includes the additional constraint $\sigma \cdot \mathbf{n} = 0$ on the boundary of obstacles [39]. There might be different maximum flows for one given environment.

For our navigation applications we have found that having directions at the sink edges orthogonal to the sink edges is not necessary for maximizing the number of outgoing agents that can exit the environment. It is enough that directions are outgoing, i.e., that $\sigma \cdot \mathbf{n} > 0$.

Considering a more generic setting, we would also only be interested in maximizing the flow with respect to outgoing directions at the sink edges which are reachable, by following the flow, from a point in a source edge. These adaptations are addressed by function f in our proposed Definition 1. This definition also specifies that the magnitude of the vector field is always 1 when it is defined, or 0 in regions where the flow is not useful.

Definition 1. (MAX FLOW.) *A max flow in $\mathcal{P}$ is a vector field $\sigma : \mathcal{P} \rightarrow \mathbb{R}^2$, that maximizes $\int_{P_{snk}} f(\mathbf{p})\, ds$, subject to:* $\operatorname{div} \sigma = 0$ *in $\mathcal{P}$ and $|\sigma| \in \{0, 1\}$ in $\mathcal{P}$, where:*

$$f(\mathbf{p}) = \begin{cases} 1, & \sigma \cdot \mathbf{n} > 0 \text{ and } \mathbf{p} \text{ is reachable from a source,} \\ 0, & \text{otherwise.} \end{cases}$$

Point $\mathbf{p}$ in Definition 1 is the point in a sink edge at which normal vector $\mathbf{n}$ is computed. Function f ensures that only the useful flow to route agents from source to sink is considered in the maximization. Function f will lead to a max flow insensitive to the outgoing angles at sink edges, and to the possibility that parts of the flow encode arbitrary directions not useful to routing agents from source to sink.

The computational methods proposed in this work are based on GPU-based rasterization techniques that will generate a max flow according to Definition 1, but will represent it in a discretized form, in a frame buffer grid in the GPU. The result will be a *max flow map*, as defined below.

Definition 2. (MAX FLOW MAP.) *A max flow map for $\mathcal{P}$ is a discrete vector field $v : G \rightarrow \mathbb{R}^2$, where G is a 2D grid covering $\mathcal{P}$, and:*

$$v(\mathbf{x}) = \sigma(\mathbf{x}), \forall \mathbf{x} : |\sigma(\mathbf{x})| = 1, \text{ where } \sigma \text{ is a max flow in } \mathcal{P}.$$

Definition 2 captures the types of flow that our computational method produces. Basically a max flow map represents a max flow and also allows for additional directions to exist in

parts of the environment that are not useful to route agents from source to sink. In both cases the flow value, or the *capacity* of the flow, for routing agents from source to sink is the same. The flow value can also be determined as the length of the polygonal min cut of the environment [38], which captures the narrowest space constraining the flow capacity. See Figure 4.2.

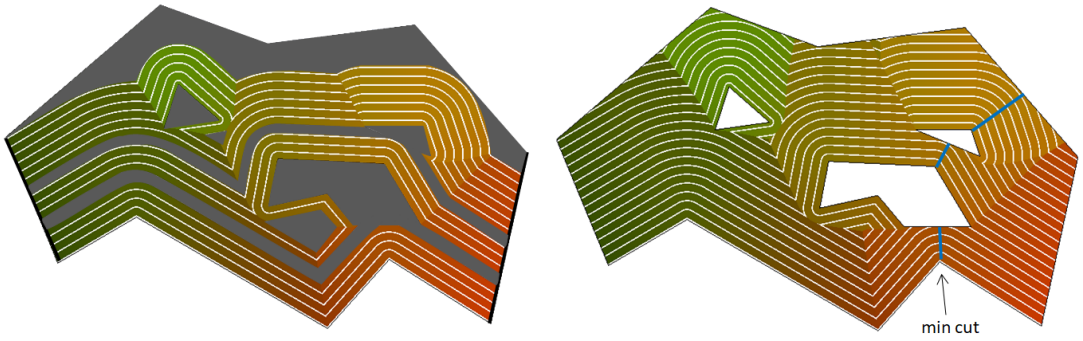


Figure 4.2: Left: this max flow routes agents from source to sink edges and is illustrated with flow lines following the flow directions. The non-useful parts of the environment have directions with magnitude 0 and are shown as gray regions. Right: our equivalent max flow map includes non-null directions for the non-useful parts of the flow. While in this work our flow maps present directions as they are generated by our computational method, it would also be possible to design directions for the non-useful portions that lead agents to a useful portion of the flow. The *min cut* of the environment is represented by the 3 illustrated segments. The sum of their lengths is equal to the max flow value.

Our flow map generation method requires the computation of Shortest Path Maps (SPMs) to accumulate the values of a max flow, and so we define SPMs next. A complete exposition of the method including additional details and extensions is available in our previous work [13], as was detailed in Chapter 3.

SPMs are structures constructed with respect to one or more source points or source segments, and that partition the space into regions that share the same sequence of points along the shortest collision-free path to the closest source. An SPM therefore encodes shortest

paths from *all* points in a given planar environment to the closest point in a source. For any given point $\mathbf{x}$, its shortest path $\pi(\mathbf{x})$ to the closest source is reconstructed by retrieving parent points along $\pi(\mathbf{x})$ until a source is reached. A "source" here refers to a source of the SPM and is not related to a source edge of a max flow.

Let source points and source segments be defined inside polygonal domain $\mathcal{P}$, which also contains the set of polygonal obstacles $\mathcal{O}$. Our SPM representation encodes lengths and parent points of shortest paths, and can be defined as follows.

Definition 3. (SHORTEST PATH MAP.) *A shortest path map (SPM) in $\mathcal{P}$ is a grid-based representation $s = (s_d, s_p)$, $s_d : G \to \mathbb{R}$, $s_p : G \to \mathbb{R}^2$, where G is a 2D grid covering $\mathcal{P}$, $s_d(\mathbf{x}) =$ length of $\pi(\mathbf{x})$, and $s_p(\mathbf{x}) =$ parent vertex of $\mathbf{x}$ along $\pi(\mathbf{x})$. If $\mathbf{x}$ is a non-reachable point $(s_d(\mathbf{x}), s_p(\mathbf{x})) = (-1, \mathbf{x})$.*

An SPM therefore encodes, for each reachable point in $\mathcal{P} - \mathcal{O}$, 1) its geodesic distance to the closest point in a source, and 2) the next "parent point" to reconstruct the shortest path to the closest point in a source, which is always an obstacle vertex or a point in a source. Fig. 4.3 shows an example SPM computed for a single segment source at the bottom of the environment.

Our max flow maps are obtained by composing SPMs computed for source polygonal lines from $\mathcal{P}$ and $\mathcal{O}$. This process will be described in Section 4.4. However, obtaining a flow map only partially solves the navigation problem of routing multiple agents to traverse $\mathcal{P}$. After a flow map is obtained, we still need to determine where to direct agents to enter P_{src}, a process we address by determining lanes in the map. In addition, we are also interested in minimizing the length of the lanes such that the overall travel time is reduced when agents follow the lanes. Our overall approach is illustrated in Figure 4.4.

While the process illustrated in Figure 4.4 represents a complete methodology for routing agents using max flows, an alternative max flow map construction method is needed in

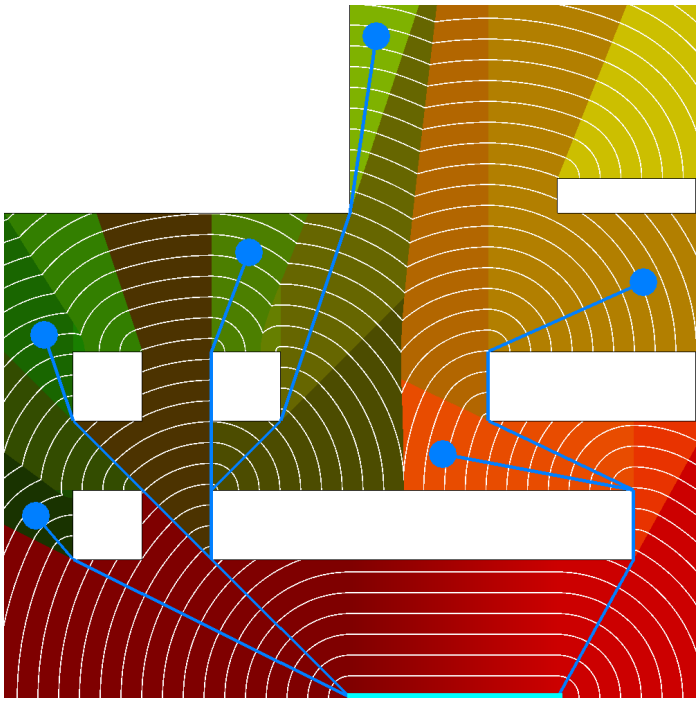
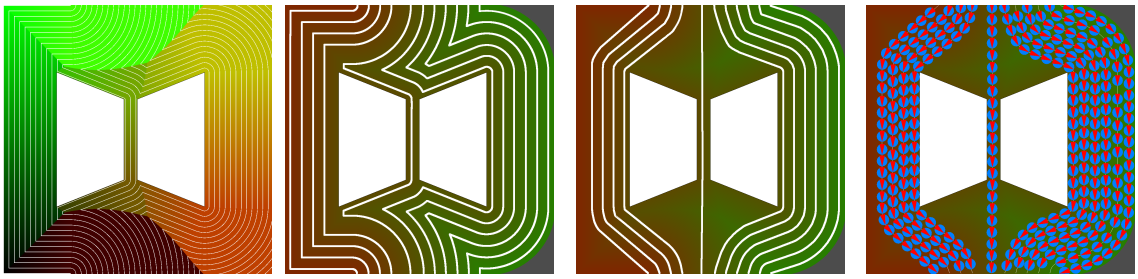


Figure 4.3: Shortest Path Map (SPM) example. Contour lines represent points equidistant to the SPM's source segment (highlighted bottom segment. Discs represent agents whose polygonal shortest paths are also shown. Each region of the SPM, denoted with a same color, shares the same vertex to be taken when reconstructing a shortest path to the source segment.

order to guarantee that the final system of lanes utilizes the optimal maximum capacity of the environment. This alternative method is needed because the max flow map encodes a flow without observing clearance constraints. The capacity of the max flow can therefore be only guaranteed to be optimal for particles of infinitesimal size. If a max flow uses corridors in the environment with less clearance than the clearance required by an agent, that portion of the flow will not be useful for that particular agent.

We therefore introduce in this work (in Section 4.5) an alternative method for generating a max flow map that takes into account the size of agents. The approach is to contract the corridors of the environment such that the *minimum width* of the flow passing by each



(a) (b) (c) (d)

Figure 4.4: Overview of the main steps of our overall approach. (a) Max flow map of the input environment. (b) Lanes extracted from the flow map. (c) Optimized lanes. (d) Using lanes to guide agents from source to sink.

corridor becomes a multiple of the agent size. In this way the generated flow will allow agents to fully utilize the maximum capacity of the environment. We define this clearance-based max flow map below.

Definition 4. (CLEARANCE-BASED MAX FLOW MAP.) *A clearance-based max flow map for paths with clearance c in $\mathcal{P}$ is a max flow map $v_c : G \to \mathbb{R}^2$, where $\forall \mathbf{x} : |\sigma(\mathbf{x})| = 1$, the minimum width of the flow passing by $\mathbf{x}$ is a multiple of $2c$.*

In the above definition the minimum width of the flow passing by $\mathbf{x}$ is restricted to be a multiple of $2c$, such that the generated flow can always be fully utilized by paths of clearance c, and the restriction is limited to the flow lines that go from P_{src} to P_{snk}. If a disc agent has radius r, the path clearance needed is r, and the width of a lane is $2r$. The construction of clearance-based max flow maps is presented in Section 4.5. An improved lane determination method is also presented based on points evenly spaced along segments used to determine the flow width, which are called *gates*.

Overall, our presented methods are able to produce lanes among obstacles that achieve maximum flow, minimize total length, and can ensure a given clearance r.

4.4 Computing Max Flow Maps

We compute max flow maps following the approach described by Mitchell [38] which is based on applying Shortest Path Maps (SPMs) [37] to accumulate distances across the environment. The distance accumulation however requires to compute the SPM for the special case of considering $0/1/\infty$-cost regions. This is a limited case of the general Weighted Region Problem where only three weights exist: 0 (no cost), 1 (cost proportional to distance traveled), and ∞ (impassable region) [17]. Figure 4.5 illustrates the difference between a traditional shortest path considering an obstacle of ∞ cost, and a shortest path considering a 0-cost obstacle.

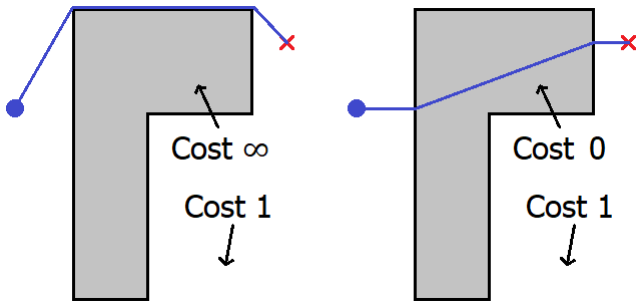


Figure 4.5: Shortest pats for obstacles of infinite (left) and zero (right) cost. When an obstacle has zero cost it means that the portion of the path passing through the obstacle does not add any amount to the total cost of the path.

Given the polygonal domain $\mathcal{P}$, its obstacles $\mathcal{O}$, source edges P_{src} and sink edges P_{snk}, the max flow map from P_{src} to P_{snk} will be obtained by computing the SPM with source as P_{bot} or P_{top}, and considering the obstacles to be 0-cost regions. We will denote the target SPM considering obstacles to have cost 0 as the SPM^0.

SPM^0 will accumulate distances from one boundary of the domain to the other without considering distances across obstacles. The distances that are accumulated will only encode the width of the free corridors in the environment, which will specify how much flow can pass by each corridor. The vector field defining the max flow map will then consist of the

vectors orthogonal to the isolines of the obtained SPM^0.

In order to compute SPM^0 we will apply our SPM method for regular regions multiple times, updating distances at each stage according to information obtained by first building the so-called *critical graph* of the environment.

4.4.1 Critical Graph

The critical graph of a polygonal domain captures key visibility information in the environment [38, 17]. Our critical graph is comprised of the shortest line segments connecting every pair of obstacles, every pair of obstacle and boundary, and P_{bot} and P_{top}, such that each segment does not cross an obstacle. In our representation these segments become the edges of the critical graph, and the obstacles become the nodes. The source and sink edges do not need to be considered for the purpose of computing SPM^0.

Figure 4.6 illustrates all shortest segments that are considered in order to identify the shortest ones composing our critical graph.

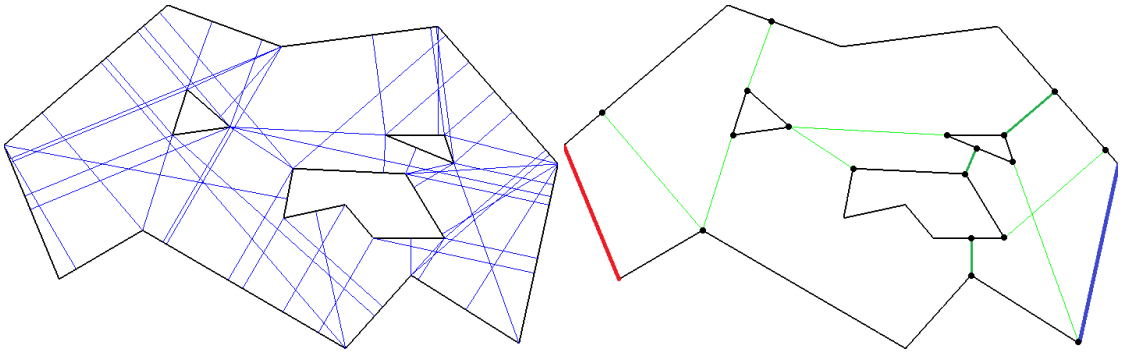


Figure 4.6: Segments considered (left) in order to identify the shortest segments connecting pairs of obstacles and boundaries that compose our critical graph (right).

The critical graph encodes the width of all the corridors in the environment, and as well the pairs of obstacles and boundaries that delimit the narrowest parts of corridors. This information will be used to compute SPM^0.

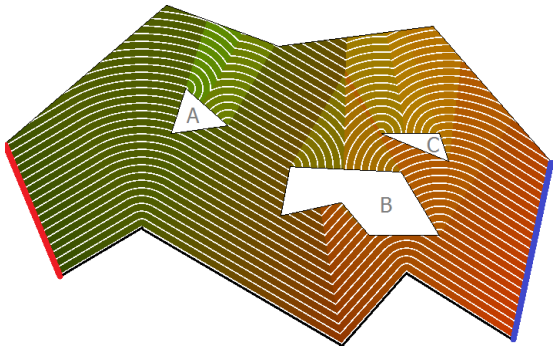
4.4.2 Main Algorithm

Once the critical graph of the environment is available we start by computing the SPM for segment sources which are either P_{bot} or P_{top}. In this section we choose P_{bot} as the starting polygonal line source. The process consists of the steps described below, which use Figure 4.7 as reference:

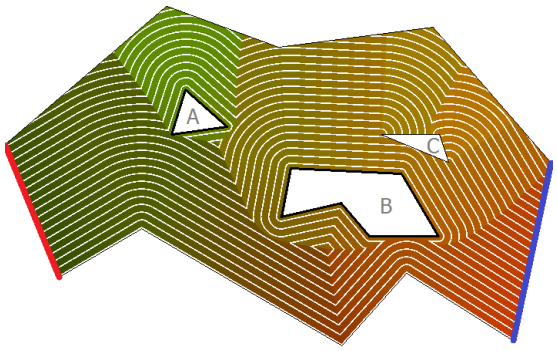
1. First, the segments of P_{bot} are set to be the initial SPM segment sources and the SPM of the scene is computed. The result is shown in Figure 4.7a.

2. For each obstacle O_i in the domain, the set E_i of the edges of the critical graph that connect to O_i is determined. Let $\mathbf{p}_{i_j}$ be the points that the edges in E_i connect to O_i. These are the narrowest corridor points in O_i. At each point $\mathbf{p}_{i_j}$ the accumulated distance $s_d(\mathbf{p}_{i_j})$ from the current SPM generation can be obtained from the SPM buffer. Considering all the edges in E_i, let d_i be the smallest $s_d(\mathbf{p}_{i_j})$. Each value $s_d(\mathbf{p}_{i_j})$ is compared against d_i. If d_i is not found to be smaller than any $s_d(\mathbf{p}_{i_j})$, then no shorter path to O_i was found. If this is true for every O_i, the algorithm stops and SPM^0 has been obtained. Otherwise, proceed to the next step.

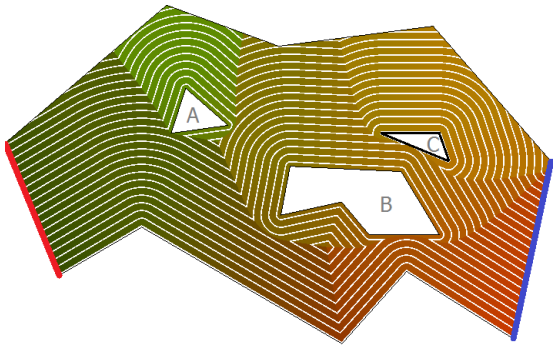
3. Here d_i has a smaller distance than some $s_d(\mathbf{p}_{i_j})$, because the 0-cost of the obstacle has not been considered. The boundary of O_i is included in the list of line segments to be used as segment sources for the SPM construction of the next iteration, with the modification that d_i is used as the initial distance for the segment sources generated from O_i. The same is performed for every other obstacle O_k which is found to have a smaller d_k and is thus contributing to the new set of segment sources. Once all segment sources are identified, a new SPM is generated. However, the new values $s_d(\mathbf{p}_{i_j})$ generated will only go to the current buffer if they represent smaller distances than the values already in the buffer. Figure 4.7b shows the result obtained after the second SPM execution in the illustrated environment.



(a) SPM generated with lines sources taken from P_{bot}. The red line is the source and the blue line is the sink.



(b) Obstacles A and B have shortest distances than the ones accumulated by the previous SPM, and a new iteration was performed with the boundaries of A and B as sources, altering the map.



(c) Obstacle C had a shortest distance, through obstacle B, and generated one additional iteration, finalizing the max flow map.

Figure 4.7: Example steps to generate a max flow map.

4. Go to step 2.

The iterations will stop when no more updates are needed. At this point SPM^0 will be obtained. Figure 4.7c shows the final result for the illustrated environment.

Once the process above is completed each direction of the max flow map is set to be the unit vector orthogonal to the final distance field, i.e., the max flow map vector field will store vectors parallel to the white isolines of the SPM^0 shown in Figure 4.7c.

4.4.3 Lane Extraction from a Max Flow Map

The max flow map as computed in this section can be directly used to route agents to traverse the environment. However, depending where each agent enters a source edge, it may arrive or not at the sink edge, and it is also useful to know how many agents can be deployed at the same time without creating collisions between agents. It is therefore useful to extract lanes from the flow so that agents can quickly select a free lane to use.

Lanes are represented with paths originating at P_{src} and following the flow field until reaching P_{snk}. By following lanes agents will move towards the sink in an orderly fashion without any bottlenecks. Agents always have a certain size, and although the max flow map described in this section does not consider agent size, lanes can still be determined with the needed clearance so that agents following lanes will not collide with obstacles or with other agents in adjacent lanes. We consider that each agent is represented by a circle of radius r and a lane determination process for clearance value r is therefore necessary.

We employ a simple lane determination procedure that is implemented as follows. We take points along P_{src} from an extreme endpoint towards the other extreme endpoint in order to determine candidate starting points for lanes. If the current candidate lane is found to be invalid, due lack of clearance or not reaching P_{snk}, we advance by a small increment Δ along P_{src} and try again. In our scenarios we have set Δ to be the height of a pixel, such

that we consider lanes starting at every center of a cell in our grid representation G of the flow map. If a valid candidate lane is found, we advance by $2r$ because we know adjacent lanes must be spaced by at least $2r$. Because of this, and the fact that lanes run aligned with each other, we do not need to check for collisions with previously-accepted valid lanes.

The lanes that are produced simply follow the flow and they can often display unnecessary turns in the environment. Section 4.6 describes a length optimization procedure that can greatly improve the overall system of lanes that is obtained. Although the lane determination procedure described in this section already incorporates clearance, the underlying flow may pass by corridors that are too narrow for a lane to pass. This may lead to a sub-optimal number of lanes generated. In order to achieve the maximum number of possible lanes with a given clearance, it is possible to generate the flow already taking into account the clearance value. This leads to a clearance-based max flow map, as described in the next section.

4.5 Clearance-Based Max Flow Maps

Given a pair of disjoint obstacles, it is possible to capture the narrowest passage between them with the shortest segment that connects them without crossing any obstacles. We will refer to this segment as the *gate* of the corridor between the two obstacles. Such segments will be directly available from the critical graph of the input environment (Figure 4.6-right).

The length of a gate determines the maximum number of lanes that can pass between those two obstacles. Most of the time a gate will not exactly accommodate a whole number of lanes, but rather will have some "leftover space" when a given number of lanes cross the gate. For a lane candidate to be considered valid, all gates it crosses must have enough clearance for it. However, in some cases the leftover space may be still used by the flow instead of using other corridors that might still have space for lanes to pass. This means that a clearance-insensitive max flow may flow through unpassable narrow corridors instead of

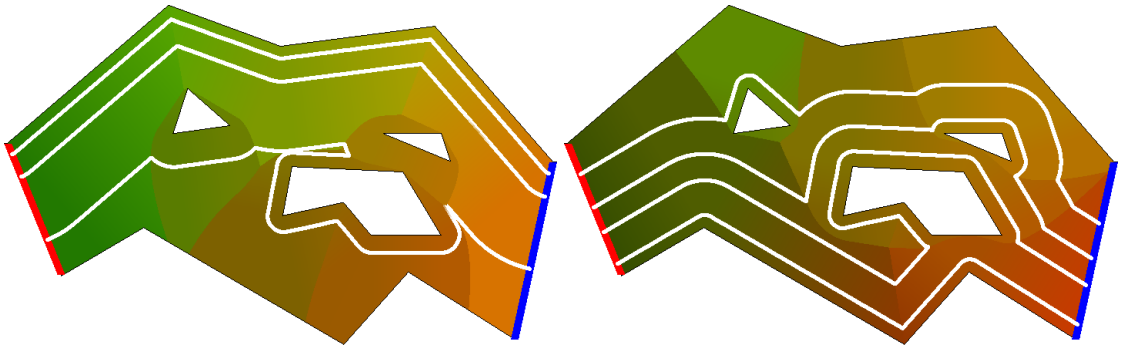


Figure 4.8: Left: because of how the clearance-insensitive flow winds up in this example, a lane with clearance that simply follows it might close off a space in a way that a corridor is blocked and the maximum number of possible lanes in the environment is not extracted. Right: In this environment, when the max flow map is generated from the bottom boundary, the maximum number of lanes is correctly extracted. To ensure that generated flows always generate the maximum number of lanes the desired clearance is taken into account during the flow construction.

larger ones. See Figure 4.8 for an illustration of the problem.

In order to address such situations we need to eliminate the leftover space of all corridors in the environment by forcing all gates to only fit a whole number of lanes. In this way we prevent the creation of flow portions that cannot fit a lane, and obtain the correct maximum capacity of the environment when considering lanes with clearance. The approach of contracting edges of the critical graph has been already suggested for solving *integer* flows, which addresses the equivalent case of routing wires that need to be spaced by 1 unit. The implementation of this approach in our max flow map methodology requires that we alter the boundary of the obstacles in a way equivalent to contracting gates.

First, by using the critical graph we are able to determine all the gates in the environment between obstacles and $\mathcal{P}$. We then contract every gate such that its new length is equal to a whole number of lanes based on the needed agent clearance. To do this we pick one of

the endpoints of the gates, point $\mathbf{p}$, and displaced it along the gate, in the direction of the gate mid-point, by distance $l = l_g \mod 2r$, where l is the leftover space, l_g is the original gate length, and r is the radius of the agent. In doing so we obtain the contracted point $\mathbf{p}_c$. We then add the original point $\mathbf{p}$ as a vertex of the obstacle along its boundary, in case it is not already a vertex, and replace its coordinates with $\mathbf{p}_c$. In this way we locally enlarge the obstacle only enough such that the new gate will fit the same number of lanes as before, but exactly. See Figure 4.9.

The eliminated leftover space will force any remaining flow to be generated in different corridors. An alternative method for gate contraction would be to displace each gate endpoint by $l/2$, leading to a corrected flow with space $l/2$ at both sides of the corridor.

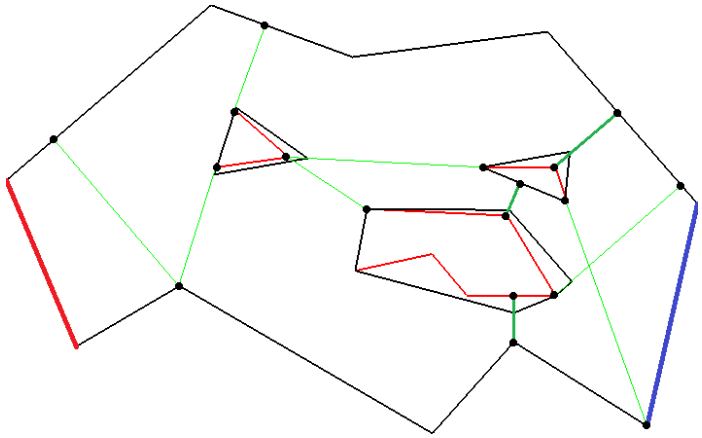


Figure 4.9: The *gates*, or shortest segments, between the boundary and obstacles, and between pairs of obstacles, are contracted by displacing vertices of the obstacles such that corridors will fit an exact number of lanes. Here, as an additional optimization, the convex hull of the obstacles is adjusted. Because gate contraction is only important to distribute flows in corridors, there is no need to contract the gate between P_{bot} and P_{top}.

4.5.1 Improved Lane Extraction using the Min Cut

Given that the gates of the critical graph for a clearance-based max flow map are contracted, an improved lane determination method can be generated based on the min cut of

the environment.

The min-cut of the environment can be found with a Dijkstra search on the critical graph. We consider the critical graph where each obstacle or boundary is a node, and the edges are the gates connecting pairs of nodes. We then consider P_{bot} (or P_{top}) to be the graph's initial node for the Dijkstra search, and P_{top} (or P_{bot}) to be the goal node. The result of the search will be the min cut, which consists of the collection of gates capturing the narrowest length to cross the environment from source to sink. Its length also determines the maximum number of lanes that can travel from source to sink.

Once the min cut of the environment is determined, we simply place lanes in the contracted gates of the min cut, with starting points evenly spaced along the contracted gates, with $2r$ space between them and with r space to obstacles or boundaries. We then follow the flow both ways from each starting point, to P_{src} and P_{snk}, in order to create the initial set of lanes. Since all gates can now fit a whole number of lanes, there is no longer a need to check if a lane has sufficient clearance. Note that because the final number of valid lanes depends also on the length of P_{src} and P_{snk}, lanes still have to be tested to reach them before being accepted, which is a trivial check.

This improved method can also be employed with the clearance-insensitive flow maps, however, the full validity tests will still have to be executed for each candidate lane.

Figure 4.12 shows examples where the maximum number of lanes were achieved by the clearance-based flow maps while some lanes were missed when using clearance-insensitive flow maps.

4.6 Length Optimization of Flow Lanes

Utilizing the max flow maps described in the previous sections and assigning agents to the extracted lanes provides an effective methodology to route agents through an environment.

However, depending on the layout of the scene and the relative sizes of the source and sink, the generated lanes may be inefficient with respect to the path they take through the scene, taking extremely long detours when shorter, more direct paths are available. A post-processing optimization process to reduce this inefficiency can then be applied.

For each lane, we randomly choose a pair of points p_1 and p_2 along its path, and check to see if the segment $\overline{p_1 p_2}$ is a valid "shortcut." This is only true if $\overline{p_1 p_2}$: does not intersect with any other occupied lane or obstacle segment, keeps its minimum distance to all obstacles and $\mathcal{P}$ as at least r, and keeps its distance to all other lanes as at least $2r$.

This optimization can be applied individually to any lane, in any order, and repeated any number of times. In practice, we start the process with the last assigned lane and work our way backwards to the first. This is because if P_{src}'s length is less than the min cut of the environment, the lanes will not be able to use all available space up to the side opposite of the one that initiated the generation of the max flow map, and therefore the last lane tends to have the most open space to be optimized. As shortcuts are accepted and lanes are shortened, they also free up new space for subsequent lanes.

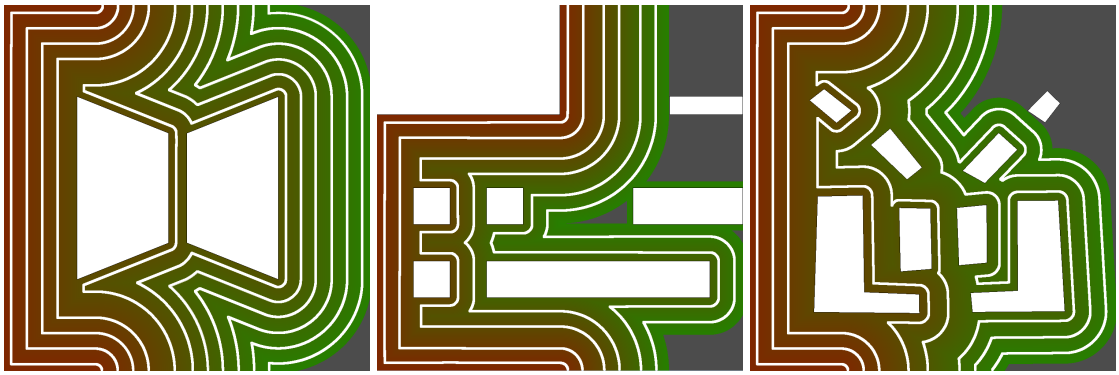
The optimization does not change the original lane assignment, it only shortens the lanes instead of searching new ways through the environment, so this optimization does not guarantee a globally-optimal configuration of lanes length-wise nor does it alter the maximal flow. However, by iterating enough times, it converges to a locally-optimal solution. The effect of this process on the lanes of our test environments is illustrated in Figure 4.10.

4.7 Results and Discussion

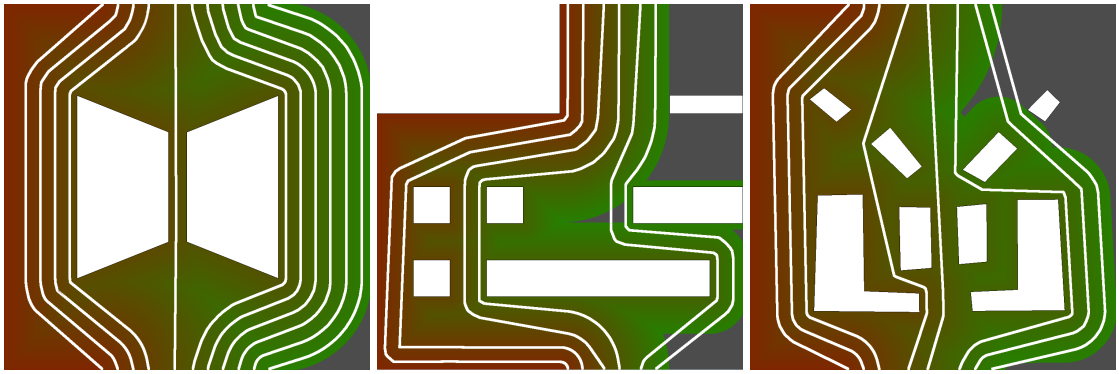
Our methods have been evaluated by producing several max flow maps and lane systems for a variety of environments. We have also produced simulation examples comparing the benefits of using our max flow trajectories versus having agents simply following their

	Scenario 1				**Scenario 2**				**Scenario 3**			
	min	**max**	**avg**	**n**	**min**	**max**	**avg**	**n**	**min**	**max**	**avg**	**n**
S:	1.97	2.02	1.99	453	2.24	2.28	2.26	214	1.97	2.03	2.00	441
L:	2.57	3.92	3.06	1053	3.30	4.39	3.89	553	2.74	4.31	3.20	766
O:	1.97	2.52	2.39	1165	2.65	3.61	3.09	611	1.98	2.89	2.40	843

Table 4.1: The left-most column indicates the used method. S: shortest paths to sink using SPM_{snk}. L: lanes from the max flow map. O: optimized lanes from the max flow map. The simulations had the agents continuously spawn at the source whenever there was space for them, and then the agents moved towards the sink according to the used method. The simulated period was of 60 seconds. Columns *min*, *max*, and *avg* refer to the minimum, maximum, and average path/lane lengths computed for the scene, respectively, and *n* is the total number of agents that were able to reach the sink in the allotted time. The three scenarios are illustrated in Fig. 4.10.



(a) original lanes



(b) optimized lanes

Figure 4.10: Scenarios 1 (left), 2 (middle), and 3 (right).

shortest paths.

4.7.1 Efficiency for Flowing Agents

In order to illustrate the benefits of using our max flow trajectories we have produced multi-agent simulations employing our generated lanes. One simulation is illustrated in Figure 4.11 and the results obtained are summarized in Table 4.1. In each simulation we define P_{src} at the top of the domain boundary and P_{snk} at the bottom, then proceed to construct three types of navigation environments:

1) The first type is based on the SPM computed with P_{snk} as source, such that agents will

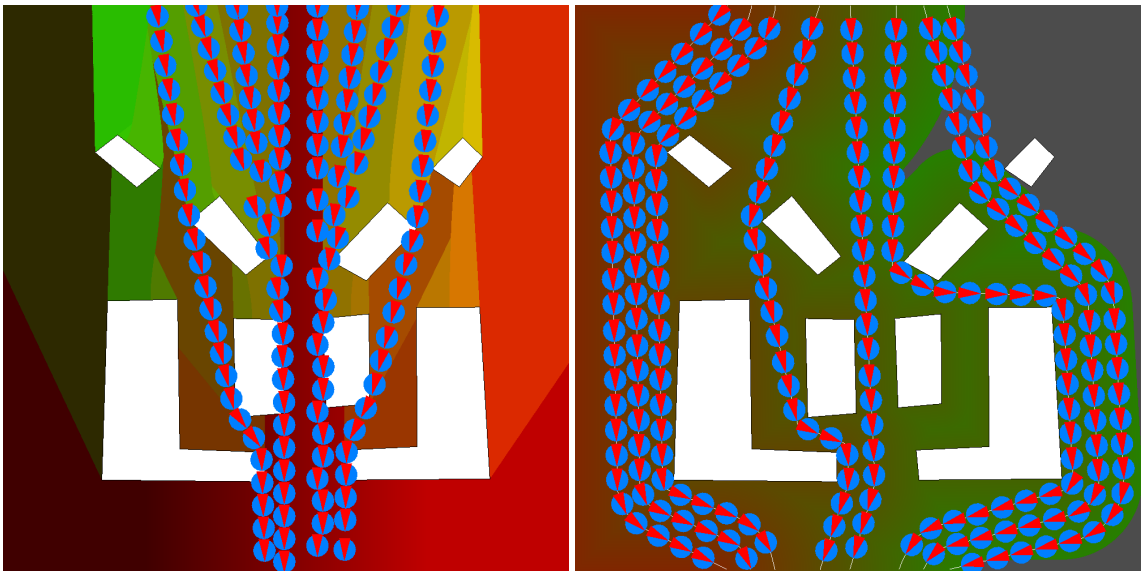


Figure 4.11: Snapshots of simulations on scenario 3. Despite both having the same amount of space and 8 lanes to start with, the SPM$_{snk}$ only permits 4 agents to reach the exit at a time, while the max flow map permits all 8 to do so.

follow their shortest paths to P_{snk}. We call this SPM as SPM$_{snk}$. Paths are the shortest possible but several bottlenecks occur which are handled with simple collision avoidance between the agents. This SPM$_{snk}$ simply encodes shortest paths and does not include any flow information. The first row in Table 4.1 presents the results.

2) The second type uses our max flow map of the environment, built starting the underlying generation from the P_{bot} side. The flow map provides directions to agents placed anywhere in the covered regions of the environment but the retrieved lanes are used to guide the agents. The lanes are optimal with respect to the flow capacity but their lengths can be further optimized. The second row in Table 4.1 presents the results.

3) In the third type the lanes obtained from the max flow map are optimized leading to a system of trajectories with minimized total length while still achieving the max flow of the environment. The results are presented in the third row of Table 4.1.

In each environment type we repeatedly spawn agents at the source edge whenever there is space for them, as the agents use either the SPM_{snk}, lanes, or optimized lanes, to navigate towards the sink. Whenever an agent reaches the sink, it is removed from the environment. The example in Figure 4.11 shows a snapshot of the simulation running on scenario 3.

The simulations ran for 60 seconds, measuring the minimum, maximum, and average lengths of the paths computed and the number of agents that were able to reach the sink during that time, as can be seen in Table 4.1. The SPM_{snk} consistently computed the shortest paths in every environment, which is to be expected since it gives the globally shortest path for each point. However, fewer agents were able to reach the sink during the simulation. When too many agents try to follow their shortest paths to the sink, bottlenecks emerge that slow down the majority of their progress.

The environment types relying on the max flow, as expected, despite having longer overall lane lengths, were better for coordinating the movement of agents throughout the environment. No bottlenecks were created, and so the max flow map led to 2 or sometimes close to 3 times as many agents reach their destination. Also, agents using the max flow map lanes did not require collision avoidance behavior. In these examples both the used max flow maps and the respective clearance-based max flow maps would lead to the same number of lanes. While lanes were slightly different, and with slightly different lengths, no significant difference on the reported values would be expected.

Our results clearly show the benefits of computing optimal flow trajectories for deploying large numbers of agents across generic polygonal domains. Because the computed flow is optimal, no better solution can be found in terms of number of agents that can reach the sink polygonal line at the same time without bottlenecks.

4.7.2 Additional Results

Several additional results are presented in Figures 4.12, 4.13, and 4.14.

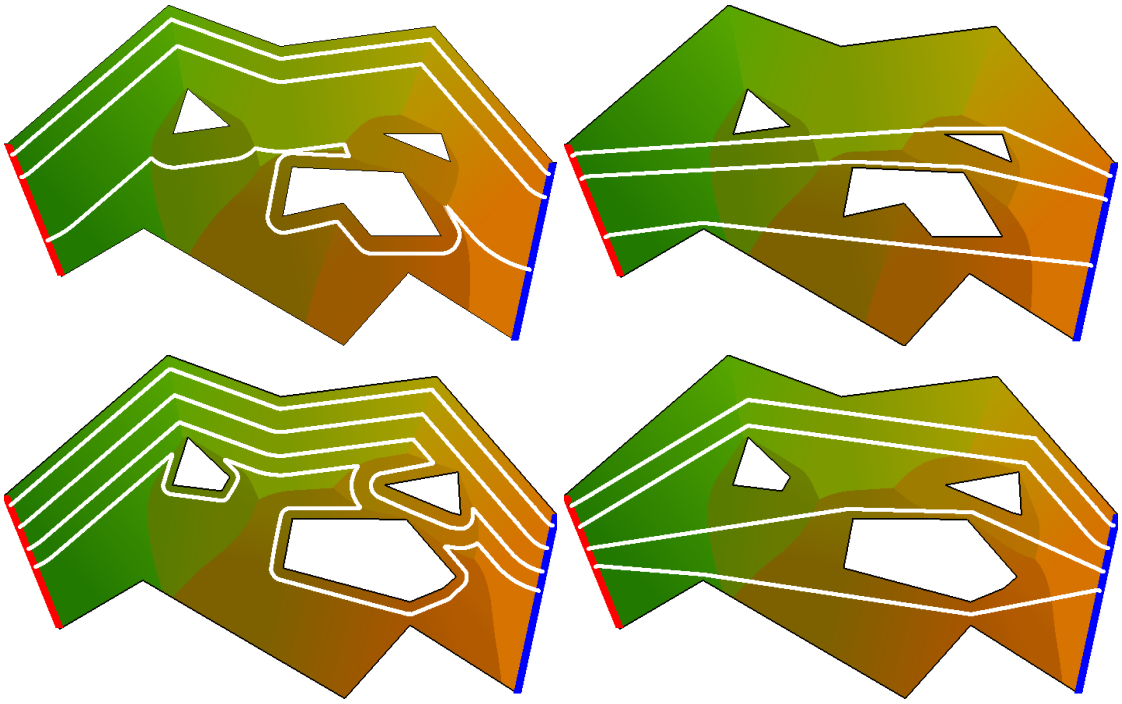


Figure 4.12: The top row shows a clearance-insensitive max flow map missing one lane compared to the clearance-based max flow map in the bottom row. The right column shows the same lanes after a length optimization process.

Figures 4.12 and 4.13 show the benefits of clearance-based flow maps. The top rows of images in Figures 4.12 and 4.13 show clearance-insensitive flow maps which lead to lanes being missed. The maps on the bottom rows are where we apply clearance-based flow maps in order to find the maximum number of lanes.

Figure 4.14 presents the results of our methods on two additional environments with higher number of obstacles. The lanes generated on the left images are unoptimized, whereas the right images shows the same lanes after length optimization.

4.7.3 Limitations and Future Work

Our current lane determination algorithms evaluate lanes by considering positions at the source or min cut edges without considering any additional global information from the

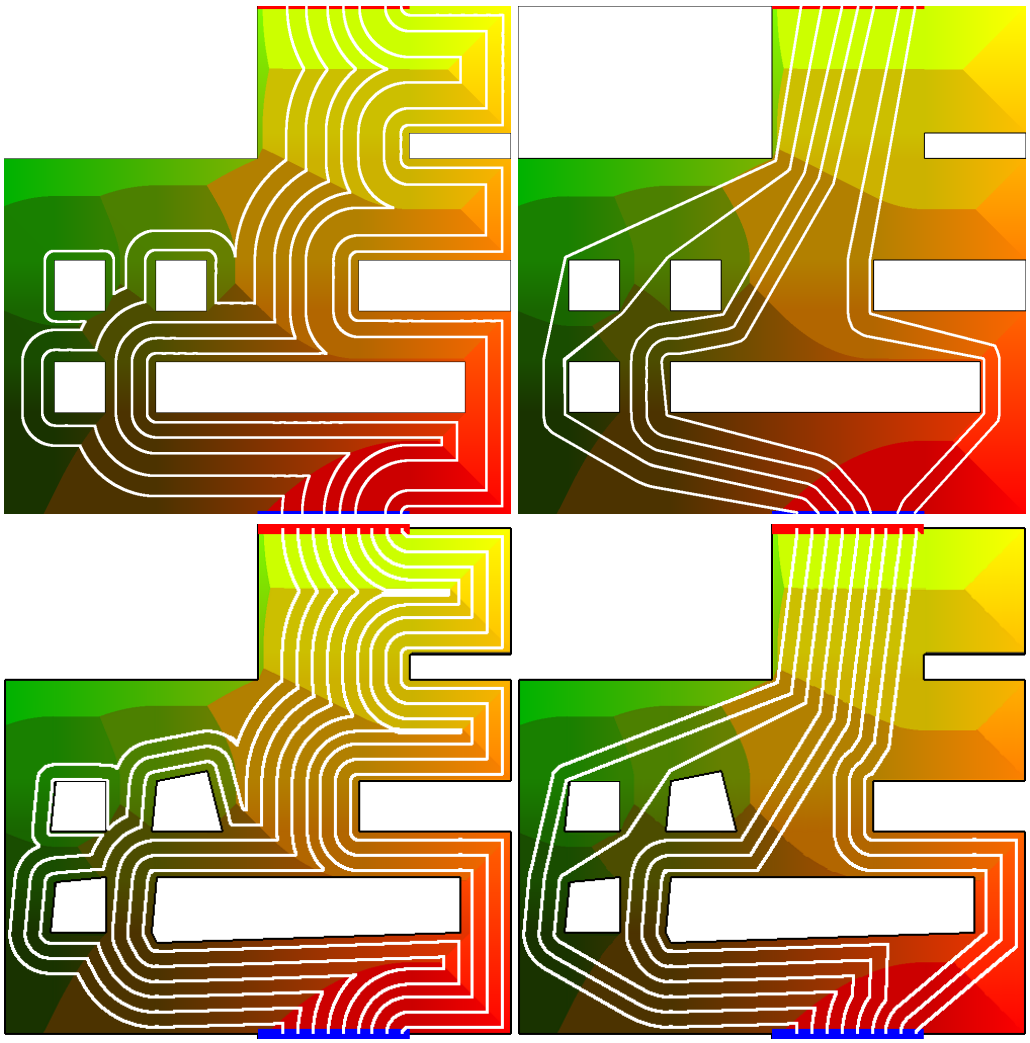


Figure 4.13: In the top row, the clearance-insensitive max flow map misses two lanes compared to the clearance-based max flow map in the bottom row. The right column shows the same lanes after a length optimization process.

environment. This may lead to a poor choice of lanes in some specific situations. For example, the lanes shown in Figure 4.15 achieve the maximum possible flow but miss the shortest path from source to sink. This situation might be addressed by first assigning lanes in the gates containing a shortest path from source to sink, and then proceeding to the remaining gates of the min cut. Another possible improvement is to allow updating the endpoints of each lane to alternative positions, in P_{src} or P_{snk}, in order to further optimize their length. We have also noticed cases where the generated flow wraps around it and comes to intersect again the source segment; however, our lane determination procedures ensure that no lanes are generated in these areas.

One promising direction for future work is to address crossing flows. For instance, groups of agents may be defined as each group having its own goal sink, and one specific flow map for each group can be then computed. Later, it might be enough to deploy agents with the right timing such that they do not collide with each other when following their on flow paths. Reactive behaviors can also be used to avoid collisions at flow crossings. Such possible approaches, among others, would allow the proposed methods to be used for agents with different goal locations.

It is also possible to address max flow maps with multiple disconnected sources and sinks. In this case there are disconnected edges in P_{src} and/or P_{snk} and there is no longer just a pair of segments on the boundary that can be divided into P_{top} and P_{bot}, but rather many segments. By applying every boundary segment that is not part of P_{src} or P_{snk} to be the starting sources of the underlying generation of SPM^0, we can compute a map such as the one illustrated in Fig. 4.16, which produces a flow routing multiple entrances to multiple exits. This approach however has the limitation that any source may be routed to any sink.

4.8 Conclusions

We have introduced in this chapter new techniques to compute max flow maps capturing the maximum flow capacity of given generic polygonal domains. The proposed methods are able to determine bottleneck-free lanes that are able to optimally guide agents to reach a destination exit of the environment. Optimality is addressed with respect to the maximum flow of agents across the environment. The presented simulations demonstrate that our approach can dramatically increase the number of agents that successfully navigate towards the goal exit of the environment in a given time frame.

The proposed approach introduces a new methodology for taking into account continuous flows in polygonal domains, and exposes several promising directions for future work, opening new research avenues in flow-based agent navigation.

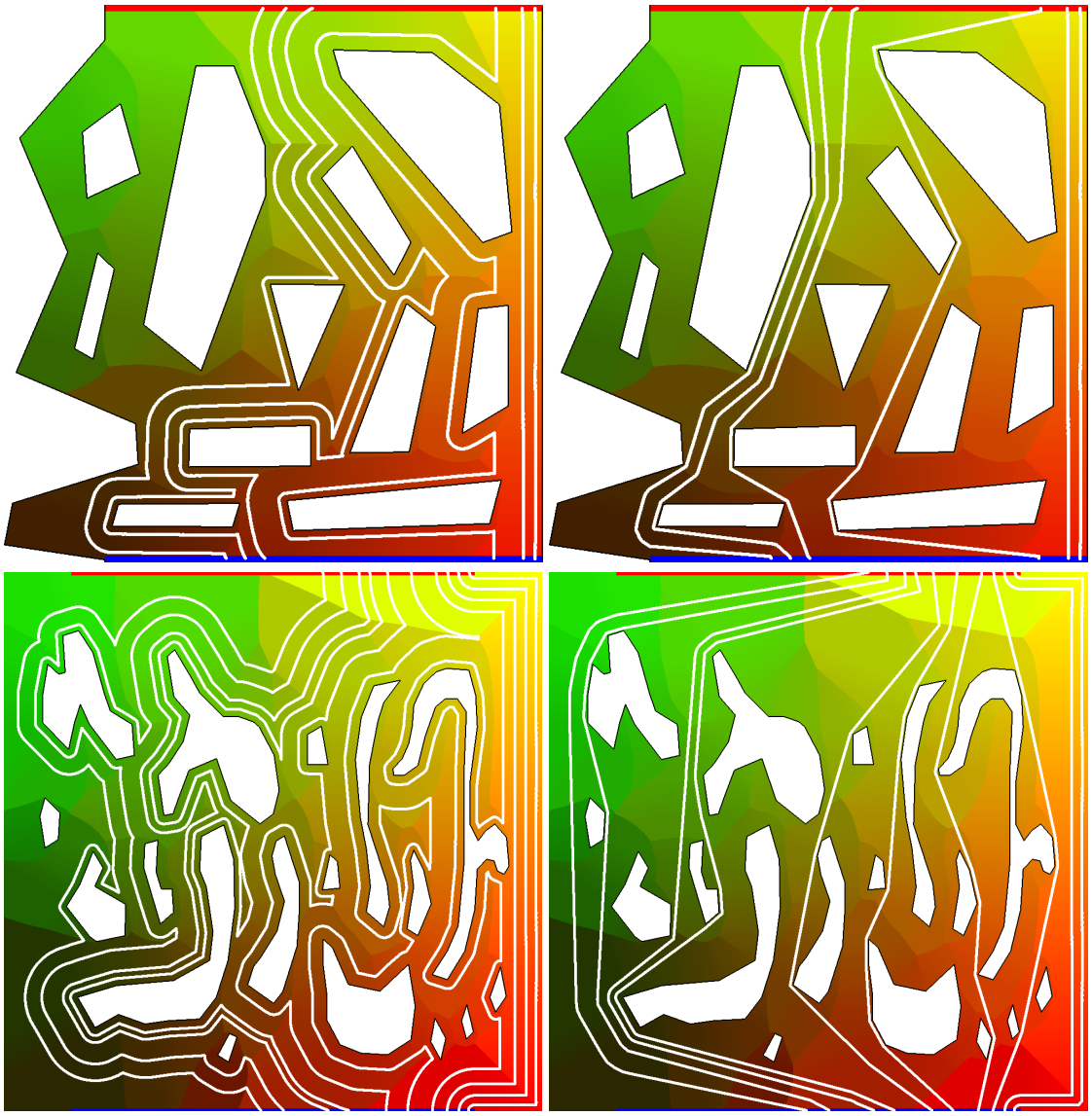


Figure 4.14: Examples generating the maximum number of lanes before (left) and after (right) length optimization.

Figure 4.15: In this environment lanes are inefficient in terms of length whether the map is generated from both P_{bot} or P_{top}.

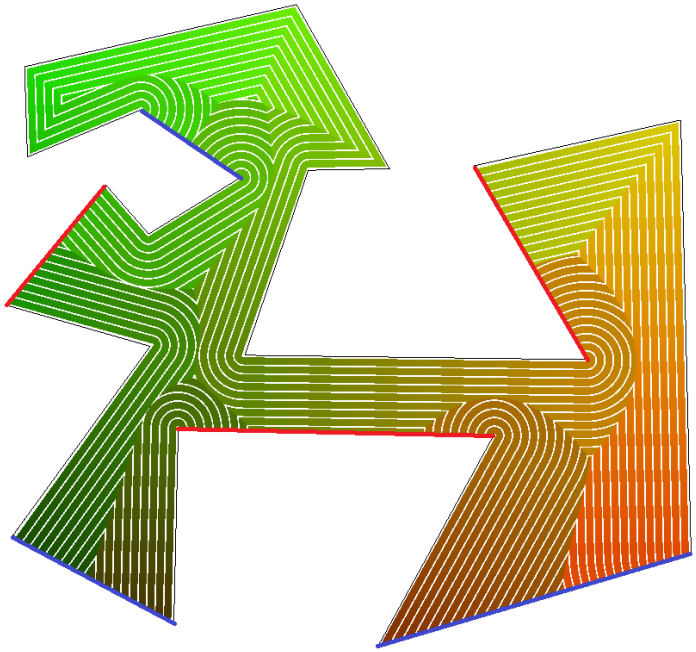


Figure 4.16: A flow map with multiple sources and sinks. The red lines are source segments belonging to P_{src} and the blue lines are sink segments belonging to P_{snk}. Every segment on the boundary inbetween them is used in the SPM generation process and thus the map is created such that agents from any source may travel to any sink, preventing crossing lanes.

CHAPTER 5

Multi-Agent Navigation with Shortest Path Maps and Adaptive Weighted Barriers

5.1 Introduction

Shortest Path Maps can efficiently provide the globally optimal shortest path for every agent traversing a polygonal environment to a given goal. However, as discussed in Chapter 4, if a large group of agents simply follows these optimal paths, they may create bottlenecks which slow the overall progress of the group. The continuous maximum flows method detailed in that chapter deals with this problem by organizing agents into lanes, guaranteeing maximum throughput but locking agents into predetermined paths.

We therefore introduce a method for leveraging both the optimal paths provided by the SPM and knowledge of local bottleneck conditions that can arise during a multi-agent navigation. The method dynamically detects regions where there is a bottleneck of agents, and places a weighted barrier in them to alter the Shortest Path Map in order to encourage some agents to choose alternative routes. This will be detailed in later sections.

The advantage of the method is that it is able to reduce the effects of bottlenecks by making some agents take alternative paths while still leveraging the information provided by the SPM to choose the best possible alternative path given the chosen parameters and the conditions of the simulation. We show that this is more efficient than merely using the SPM in

conjunction with local collision avoidance.

5.2 Related Work

There has been previous research into combining global path planning with local behavior.

Sharma et al. [50] explored the advantage of integrating Shortest Path Maps with local collision avoidance. Their work demonstrated that even using an effective collision avoidance technique, agents could easily become trapped in "pockets" of the environment. This did not happen when they also utilized the SPM's global planning.

The work of van Toll and Pettré [63] also explores the combination of global and local navigation. They combine path planning with path following and collision avoidance to define navigation strategies, which are a set of decisions for agents to pass obstacles and agents on certain sides, or even to explicitly send agents in certain directions.

Van Toll et al. [59] incorporate crowd density information into global path planning. They generate a navigation mesh with density values based on how many characters are inside each region, and then perform an A* search on this graph. The density information encourages characters to use a variety of routes, which is a goal similar to our own method. Other works have also added this extra layer of local density information to global planning [19, 46].

5.3 Problem Definition

A Shortest Path Map (SPM) is a structure, constructed with respect to a "source," which partitions a polygonal environment into regions where each region shares the same parent point on the shortest path back to the source. Once constructed, an SPM can efficiently and optimally answer path queries for any point in the environment. The goal of our method is to utilize the SPM with modifications to detect and alleviate the effects of bottlenecks

so that a large amount of agents can traverse a polygonal environment in a more efficient manner than just using the SPM with local collision avoidance.

The set of polygonal obstacles $\mathcal{O}$ of the environment, with a total of n vertices, is defined in $\mathcal{P}$, where $\mathcal{P} \subset \mathbb{R}^2$ defines the polygonal domain containing the line segment sources and obstacles. All shortest paths generated are collision free with respect to $\mathcal{O}$.

Let n_l be line segment sources $\{\mathbf{l}_1, \mathbf{l}_2, ..., \mathbf{l}_{n_l}\}$ defined in the plane, such that $\mathbf{l}_i$, $i \in \{1, 2, ..., n_l\}$, consisting of two endpoints $\in \mathcal{P}$. These line segment sources represent the goals the agents will attempt to reach, and will be used to construct the SPM of the environment that will be used for path queries.

Given this, an SPM will be constructed following our method [13] discussed in Chapter 3 which will represent all globally-optimal collision-free paths from any point $\mathbf{p} \in \mathcal{P} - \mathcal{O}$ to the closest reachable point on its closest segment source $\mathbf{l}_i$. See Figure 5.1.

A "corridor" is loosely defined here as the space between a pair of obstacles in the environment. This is where bottlenecks tend to occur and is where the weighted barriers will be placed. Both the bottleneck detection process and how barriers modify the SPM are discussed in the following sections.

5.4 Bottleneck Detection

In our multi-agent simulation, agents emerge from segment(s) in the environment and navigate towards a common goal. Each agent queries the SPM of the environment to get its shortest path to the goal and follows it. Naturally, bottlenecks will occur when the influx of agents into a corridor is greater than the outflow on the other side. The first thing our method must do is detect that a bottleneck is occurring.

Let Δt be the timestep of the multi-agent simulation, and Δd_{max} be the maximum dis-

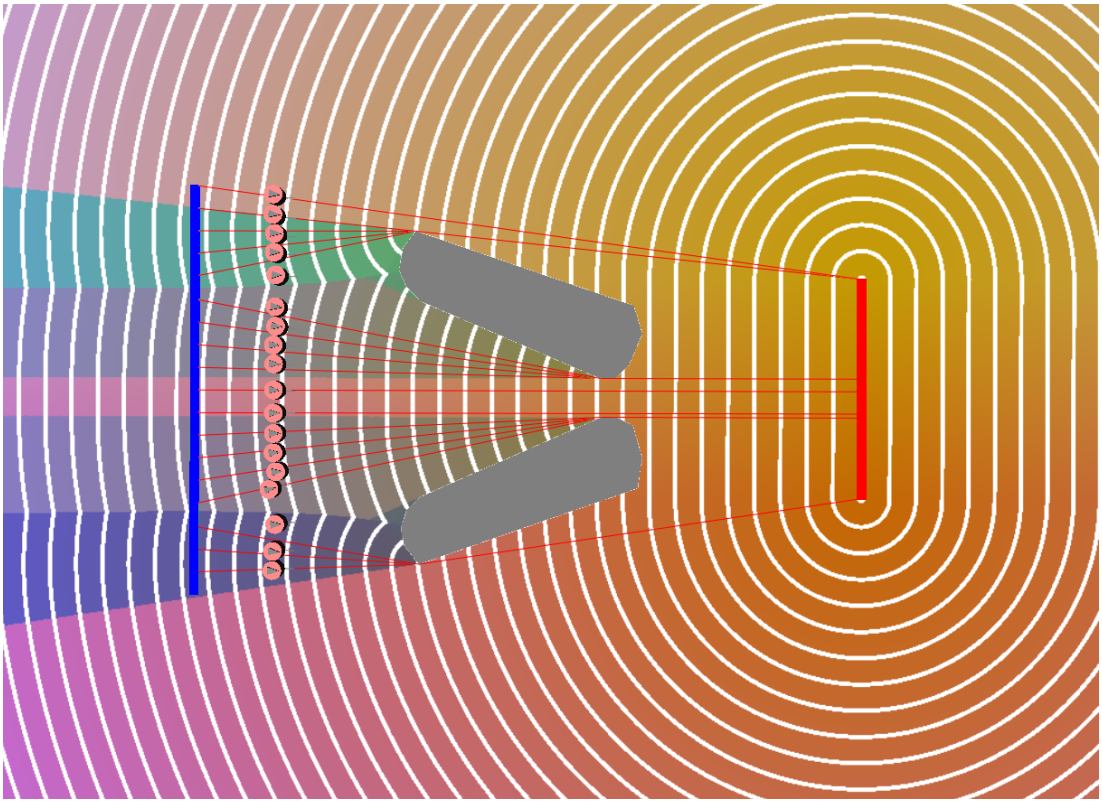


Figure 5.1: SPM computed from our simulation environment. The red segment on the right is the segment source. Agents emerge from the blue segment on the left and follow their shortest paths to the source.

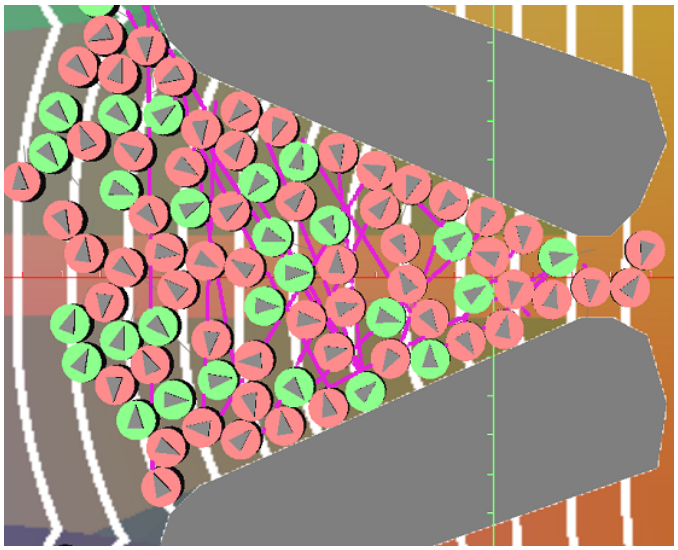
tance that an agent can cross in Δt, when it is completely unobstructed and moving at max speed. Let $\Delta d_{current}$ be the distance from the agent's current location to the goal, and Δd_{prev} the distance from its location in the previous iteration to the goal, both following the shortest path provided by the SPM. Thus, the difference between the two is given by $\Delta d = \Delta d_{current} - \Delta d_{prev}$.

If during an iteration $\Delta d < \Delta d_{max}$, we say that the agent was "slowed" and was not able to make sufficient progress to the goal. This is usually the case when collision avoidance forces an agent to locally modify its trajectory. We keep a counter that is incremented every iteration an agent is "slowed," and decremented otherwise. If the counter exceeds a threshold *thres* that agent is considered to be in a "bottlenecked" state.

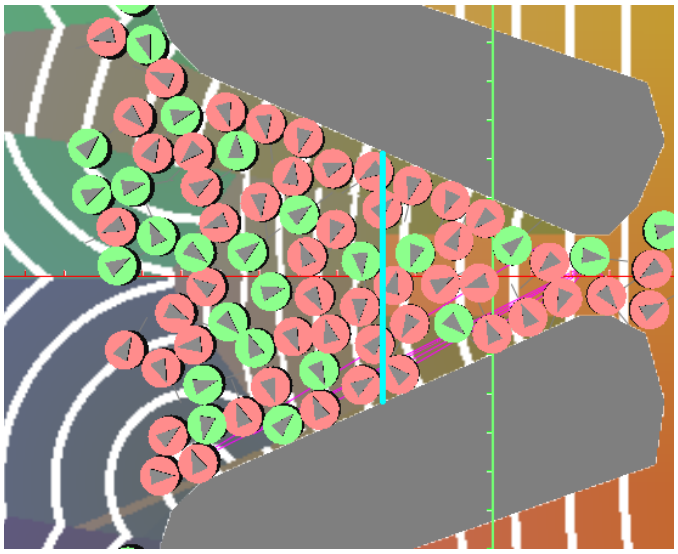
We consider that a bottleneck is happening when we are able to find a continuous chain of neighboring bottlenecked agents from one side of the corridor to the other. We start a search from every bottlenecked agent that is in contact with one of the pair of obstacles that forms the corridor, and examine its neighbors. Bottlenecked neighbors have their own neighbors examined recursively. If eventually we reach a bottlenecked agent that is in contact with the other side of the corridor, then we have detected a bottleneck.

When a bottleneck is detected, a segment is created from the first bottlenecked agent of the chain to the last, across the corridor. This is called a "candidate segment", and it is possible that many of these segments are created. Every *interval* seconds (the barrier update interval), we: 1) take the bounding box of the set of candidate segments in the corridor and place a weighted barrier down the middle of it connecting both obstacles, and 2) remove all of the candidate segments, starting the bottleneck detection search over again. See Figure 5.2.

The weighted barrier alters the SPM in a manner explained in the next section.



(a)



(b)

Figure 5.2: (a) Between update intervals, candidate segments are created when chains of bottlenecked agents going from one side of the corridor to the other are identified. The candidate segments are rendered beneath the agents as magenta lines. (b) Out of the set of candidate lanes, a weighted barrier (cyan) is created and placed down the middle, modifying the SPM. Previous candidate segments are removed.

5.5 Weighted Barrier

As described in section 3.3, the normal computation of an SPM considers that the distance between two points in the environment is simply the Euclidean distance between them. Section 3.5 introduced an extension where this could be modified at the vertices of obstacles with a weight that would act as a multiplier to the Euclidean distance. Weighted barriers change the SPM in a similar way, however instead of acting at vertices, weighted barriers are line segments connecting a pair of obstacles that apply a weight change on one side of the barrier.

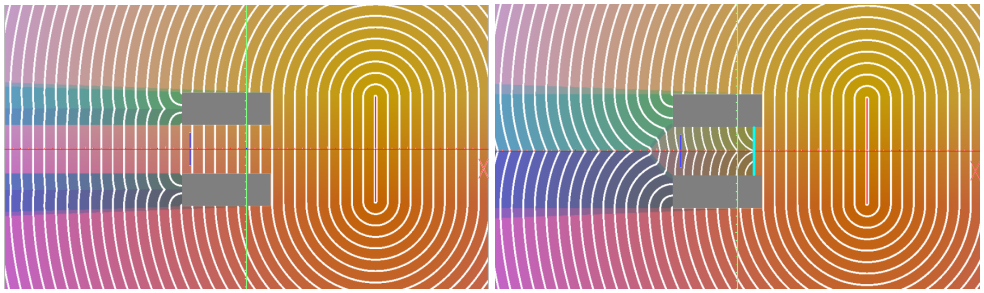
Given points a and b, let $dist(a,b)$ be the Euclidean distance between them. Absent weighted barriers, the SPM will simply use $dist(a,b)$ for its distance computations. Now consider that there is a weighted barrier inbetween a and b, with weight w and intersecting $\overline{ab}$ at point c. The distance computation becomes a weighted sum of the two parts, such that $dist(a,b) = w \times dist(a,c) + dist(c,b)$. This represents that the region beyond the barrier is costlier to move through, as there is a bottleneck there.

As seen in Figure 5.3, the result is that the SPM will adjust such that other regions will become more prevalent over the costlier region.

5.6 Results

In order to evaluate the effectiveness of the method, we executed a multi-agent simulation in our example environment where 180 agents emerge from an entrance on the left side and navigate to the goal on the right side using the paths given to them by the SPM. When an agent reaches the goal, it is removed. The parameters used were: barrier weight $w = 1.5$, bottleneck threshold $thres = 30$, and barrier update interval $interval = 10$. The simulation ran until all agents had successfully reached the goal.

The two scenarios of the simulation are illustrated in Figure 5.4. In the first scenario, the



(a) Normal (b) Weighted

(c) Normal (d) Weighted

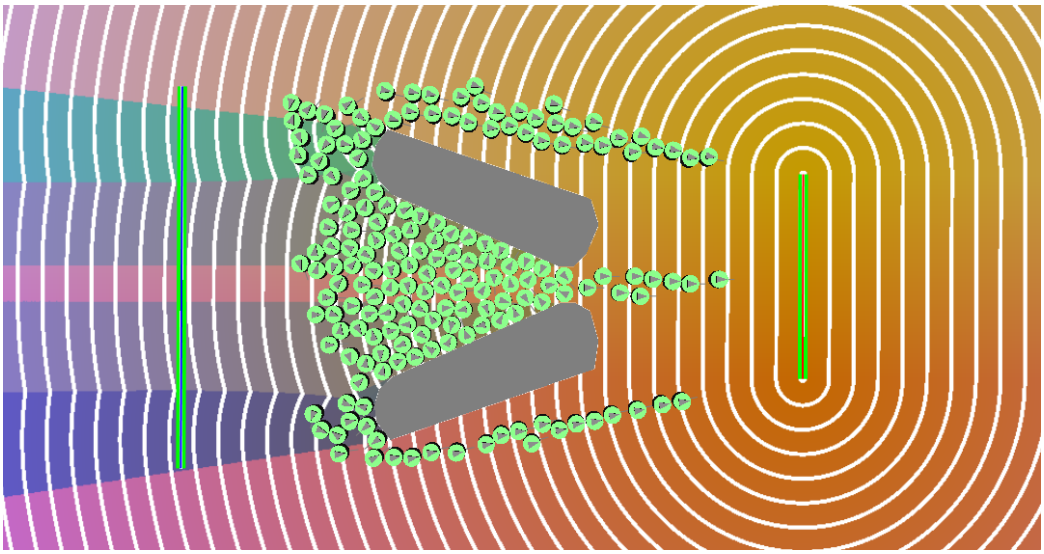
(e) Normal (f) Weighted

Figure 5.3: Examples of how the SPM changes when weighted barriers are added. SPM source segments are in red and weighted barriers in cyan.

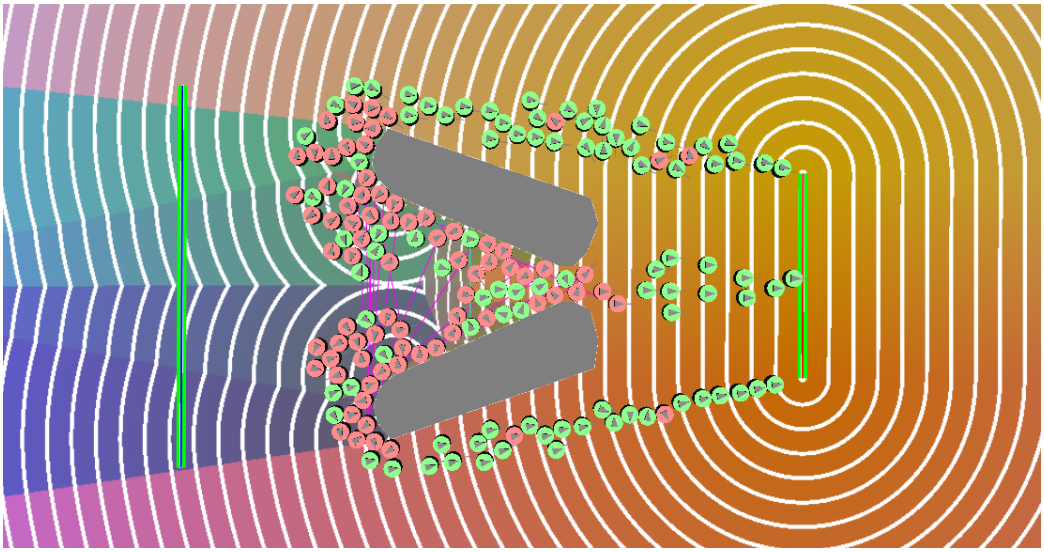
simulation ran with just the SPM and local collision avoidance, but no weighted barriers. Predictably, a large bottleneck formed in the narrow corridor in the middle, which slowed down the group. In the end it took 135.96 seconds for all of the agents to reach the goal. In the second scenario, we applied bottleneck detection and weighted barriers, which made some of the agents that would have waited in the bottleneck to prefer to go around the sides. In this case it took the agents only 105.44 seconds to reach the goal, a reduction of almost 23%.

5.7 Conclusions

We have presented here a method to combine the global optimality of Shortest Path Maps with local knowledge of bottlenecks that may arise during a multi-agent simulation. Our method is able to alleviate the effects of bottlenecks by encouraging agents to take alternative paths. Because these paths are provided by a modified SPM, each agent will choose the best possible alternative path given the conditions of the simulation and the parameters chosen.



(a) Only SPM and local collision avoidance



(b) With bottleneck detection and weighted barriers

Figure 5.4: A snapshot of the simulation running in two scenarios: a) using only the SPM and local collision avoidance, agents form a large bottleneck in the middle, b) applying the bottleneck detection process and placing weighted barriers, more agents prefer taking the side routes, reducing the severity of the bottleneck.

CHAPTER 6

Conclusions

In this dissertation I have presented new GPU-based approaches for the problems of path planning and multi-agent navigation. The focus of my work has been on methods utilizing GPU rasterization techniques to construct structures which allow us to address these problems in novel ways.

6.1 Discussion

Global navigation has been extensively explored in the past as it is a crucial element of many applications in a wide variety of fields. However, computing optimal paths efficiently is not a trivial task, and thus the focus of previous works applied to real application has remained on efficiently computing collision-free paths without optimality guarantees. My initial work on SPMs in Chapter 3 introduced a novel alternative to achieve both efficiency while still maintaining optimality. The shader-based approach greatly simplified the implementation of the SPM, and allowed us to develop multiple extensions such as vertex weights. This method also served as the foundation for the other works presented here.

Our work on continuous maximum flows in Chapter 4 computed max flow maps that are able to capture the maximum flow capacity of a polygonal domain. Optimality with respect to the maximum flow of agents through the environment is maintained, as the extracted lanes are guaranteed to be bottleneck-free, not even requiring any sort of local col-

lision avoidance. While previous works have been limited to discrete flow definitions, our method is a first method suitable for implementation for computing maximum flow maps in continuous domains, relying on the insight of applying GPU rasterization techniques.

Finally, our initial work in Chapter 5 on applying a bottleneck-detection process followed by modifying the SPM with weighted barriers shows promising results. While bottleneck-free navigation is not guaranteed as with max flows, this method also does not depend on fixed pre-computed lanes, meaning it can easily handle agents entering the environment at any point, and also generates paths which are efficient in terms of length.

6.2 Future Work

CPU Computation of Max Flows The medial axis provides much of the same information as the critical graph in our max flow method, namely the capacities of the corridors of the environment. In our method, the computed max flow map must be transferred from the GPU to the CPU for the computation of lanes, and this transfer incurs a cost. It would be interesting to explore an alternative approach that uses the medial axis as a starting point for a CPU-only solution, to see how it compares in terms of efficiency.

Disconnected Sources and Sinks Extending our max flows method to be able to handle multiple disconnected sources and sinks is an interesting direction. Figure 6.1 shows an example of such an environment, where the flow from any source can go to any sink. Our current max flow generation method appears to already be sufficient for generating the map here, although generating lanes with efficient lengths will be even trickier than before, because each lane has multiple possible goals to consider.

Crossing Flows A more difficult scenario is one where the flow from each source must reach one specific sink. In this case the flows from different sources may cross, leading to a much more difficult problem to solve globally. Optimality will likely not be possible and

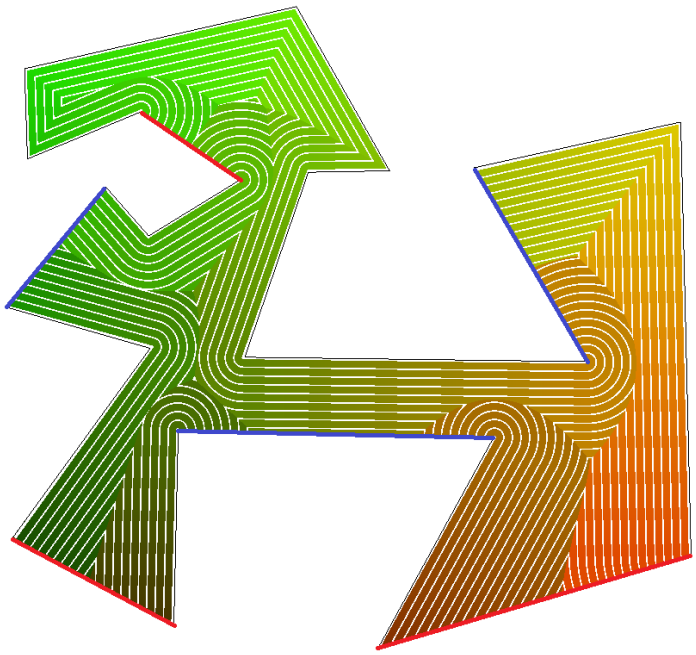


Figure 6.1: Multiple disconnected flow sources and sinks.

we will require some synchronization between lanes, such as incorporating timesteps, to avoid collisions while also remaining bottleneck-free.

Adaptable Weighted Barriers Ideally, the weight w assigned to a weighted barrier should be precisely chosen to divert just as many agents as necessary to eliminate its bottleneck and prevent it from reoccurring. This should lead to better results overall, but would require a more computationally intensive approach to exactly determine the weights needed. Additionally, the weight and position of a barrier could be adjusted up or down dynamically, if its current effect on its bottleneck could be measured.

Bibliography

[1] Timo Aila and Samuli Laine. “Alias-free Shadow Maps”. In: *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*. EGSR ’04. Norrköping, Sweden, 2004, pp. 161–166. ISBN: 3-905673-12-6. DOI: 10.2312/EGWR/EGSR04/161-166.

[2] Adam Barnett, Hubert P. H. Shum, and Taku Komura. “Coordinated Crowd Simulation With Topological Scene Analysis”. In: *Computer Graphics Forum* 35.6 (Sept. 2016), pp. 120–132.

[3] Kristof Berczi and Yusuke Kobayashi. “The Directed Disjoint Shortest Paths Problem”. In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 13:1–13:13. ISBN: 978-3-95977-049-1.

[4] W. Bilodeau and M. Songy. *Real time shadows*. Los Angeles, California, and Surrey, England, 1999.

[5] Carlo Camporesi and Marcelo Kallmann. “Computing Shortest Path Maps with GPU Shaders”. In: *Proceedings of the Seventh International Conference on Motion in Games*. MIG ’14. Playa Vista, California: ACM, 2014, pp. 97–102. ISBN: 978-1-4503-2623-0. DOI: 10.1145/2668064.2668092. URL: http://doi.acm.org/10.1145/2668064.2668092.

[6] John Carmack. *Z-fail shadow volumes*. Internet Forum. 2000.

[7] Jindong Chen and Yijie Han. “Shortest Paths on a Polyhedron”. In: *Proceedings of the Sixth Annual Symposium on Computational Geometry*. SCG ’90. Berkley, California, USA: ACM, 1990, pp. 360–369. ISBN: 0-89791-362-0. DOI: 10.1145/98524.98601. URL: http://doi.acm.org/10.1145/98524.98601.

[8] Mark De Berg et al. *Computational Geometry: Algorithms and Application*. Springer, 2008.

[9] Daniel Delling et al. *PHAST: Hardware-Accelerated Shortest Path Trees*. Tech. rep. 2010.

[10] Andrew Dobson et al. "Scalable asymptotically-optimal multi-robot motion planning". In: *2017 International Symposium on Multi-Robot and Multi-Agent Systems (MRS), Los Angeles, CA, USA, December 4-5, 2017*. 2017, pp. 120–127.

[11] Elmar Eisemann et al. "Casting Shadows in Real Time". In: *ACM SIGGRAPH ASIA '09 Courses*. Yokohama, Japan: ACM, 2009.

[12] C. Everitt and M. J. Kilgard. *Practical and robust stenciled shadow volumes for hardware-accelerated rendering*. Published online at http://developer.nvidia.com. 2002.

[13] Renato Farias and Marcelo Kallmann. "Optimal Path Maps on the GPU". In: *IEEE Transactions on Visualization and Computer Graphics* (2019).

[14] Henry Fuchs et al. "Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-planes". In: *Proceedings of SIGGRAPH '85*. New York, NY, USA: ACM, 1985, pp. 111–120. ISBN: 0-89791-166-0. DOI: 10.1145/325334.325205.

[15] Francisco M. Garcia, Mubbasir Kapadia, and Norman I. Badler. "GPU-based Dynamic Search on Adaptive Resolution Grids". In: *IEEE International Conference on Robotics and Automation*. 2014, pp. 1631–1638. DOI: 10.1109/ICRA.2014.6907070.

[16] Roland Geraerts. "Planning Short Paths with Clearance using Explicit Corridors". In: *IEEE International Conference on Robotics and Automation*. 2010, pp. 1997–2004.

[17] L. Gewali et al. “Path Planning in $0/1/\infty$ Weighted Regions with Applications”. In: *Proceedings of the Fourth Annual Symposium on Computational Geometry*. SCG ’88. Urbana-Champaign, Illinois, USA: ACM, 1988, pp. 266–278.

[18] Stephen J. Guy et al. “ClearPath: Highly Parallel Collision Avoidance for Multi-Agent Simulation”. In: *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 2009, pp. 177–187.

[19] Mario Höcker et al. “Graph-based Approaches for Simulating Pedestrian Dynamics in Building Models”. In: Sept. 2010, pp. 389–394. ISBN: 978-0-415-60507-6. DOI: 10.1201/b10527-65.

[20] T. Heidmann. *Real shadows, real time*. Iris Universe 18. Silicon Graphics, Inc, 1991, pp. 28–31.

[21] Mark Henderson et al. “High-Dimensional Planning on the GPU”. In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 2515–2522. DOI: 10.1109/ROBOT.2010.5509470.

[22] John Hershberger and Subhash Suri. “An Optimal Algorithm for Euclidean Shortest Paths in the Plane”. In: *SIAM Journal on Computing* 28 (1999), pp. 2215–2256.

[23] Kenneth E. Hoff III et al. “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware”. In: *Proceedings of SIGGRAPH ’99*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311567. URL: http://dx.doi.org/10.1145/311535.311567.

[24] Gregory S. Johnson et al. “The Irregular Z-buffer: Hardware Acceleration for Irregular Data Structures”. In: *ACM Trans. Graph.* 24.4 (Oct. 2005), pp. 1462–1482. ISSN: 0730-0301. DOI: 10.1145/1095878.1095889.

[25] Marcelo Kallmann. “Dynamic and Robust Local Clearance Triangulations”. In: *ACM Transactions on Graphics* 33.4 (2014).

[26] Mubbasir Kapadia et al. “Dynamic Search on the GPU”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 3332–3337.

[27] Ioannis Karamouzas, Roland Geraerts, and A. Frank van der Stappen. “Space-time Group Motion Planning”. In: *Workshop on the Algorithmic Foundations of Robotics*. June 2012.

[28] Samuli Laine. “Split-plane Shadow Volumes”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS ’05. Los Angeles, California: ACM, 2005, pp. 23–32. ISBN: 1-59593-086-8. DOI: 10.1145/1071866.1071870.

[29] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. “Sample Distribution Shadow Maps”. In: *Symposium on Interactive 3D Graphics and Games*. I3D ’11. San Francisco, California: ACM, 2011, pp. 97–102. ISBN: 978-1-4503-0565-5.

[30] D. T. Lee and F. P. Preparata. “Euclidean Shortest Paths in the Presence of Rectilinear Barriers”. In: *Networks* 14.3 (1984), pp. 393–410. ISSN: 1097-0037. DOI: 10.1002/net.3230140304. URL: http://dx.doi.org/10.1002/net.3230140304.

[31] Maxim Likhachev et al. “Anytime Dynamic A*: An Anytime, Replanning Algorithm”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. 2005.

[32] Yaron Lipman, Raif M. Rustamov, and Thomas A. Funkhouser. *Biharmonic Distance*. 2010.

[33] D. Brandon Lloyd et al. “Logarithmic Perspective Shadow Maps”. In: *ACM Trans. Graph.* 27.4 (Nov. 2008), 106:1–106:32. ISSN: 0730-0301.

[34] Lin Lu, Bruno Lévy, and Wenping Wang. “Centroidal Voronoi Tessellation of Line Segments and Graphs”. In: *Computer Graphics Forum* 31.2pt4 (2012), pp. 775–784.

[35] Hang Ma and Sven Koenig. “AI buzzwords explained: multi-agent path finding (MAPF)”. In: *AI Matters* 3.3 (2017), pp. 15–19.

[36] Hang Ma and Sven Koenig. “Optimal Target Assignment and Path Finding for Teams of Agents”. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. AAMAS ’16. Singapore, Singapore: International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 1144–1152.

[37] Joseph S. B. Mitchell. “A New Algorithm for Shortest Paths Among Obstacles in the Plane”. In: *Annals of Mathematics and Artificial Intelligence* 3.1 (1991), pp. 83–105. ISSN: 1573-7470. DOI: 10.1007/BF01530888. URL: http://dx.doi.org/10.1007/BF01530888.

[38] Joseph S. B. Mitchell. “On Maximum Flows in Polyhedral Domains”. In: *Proceedings of the Fourth Annual Symposium on Computational Geometry*. SCG ’88. Urbana-Champaign, Illinois, USA: ACM, 1988, pp. 341–351.

[39] Joseph S. B. Mitchell. “On Maximum Flows in Polyhedral Domains”. In: *Journal of Computer and System Sciences* 40 (1990), pp. 88–123.

[40] Joseph S. B. Mitchell. “Shortest Paths Among Obstacles in the Plane”. In: *Proceedings of the Ninth Annual Symposium on Computational Geometry*. SCG ’93. San Diego, California, USA: ACM, 1993, pp. 308–317. ISBN: 0-89791-582-8.

[41] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. “The Discrete Geodesic Problem”. In: *SIAM Journal on Computing* 16.4 (Aug. 1987), pp. 647–668. ISSN: 0097-5397. DOI: 10.1137/0216045. URL: http://dx.doi.org/10.1137/0216045.

[42] Alex Nash et al. “Theta*: Any-angle Path Planning on Grids”. In: *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*. AAAI’07. Vancouver, British Columbia, Canada: AAAI Press, 2007, pp. 1177–1183. ISBN: 978-1-

57735-323-2. URL: http://dl.acm.org/citation.cfm?id=1619797.1619835.

[43] Nils J. Nilsson. “A Mobius Automation: An Application of Artificial Intelligence Techniques”. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI ’69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 509–520.

[44] Ramon Oliva and Nuria Pelechano. “NEOGEN: Near Optimal Generator of Navigation Meshes for 3D Multi-Layered Environments”. In: *Computers & Graphics* 37.5 (Aug. 2013), pp. 403–412. ISSN: 0097-8493. DOI: 10.1016/j.cag.2013.03.004. URL: http://dx.doi.org/10.1016/j.cag.2013.03.004.

[45] Mark H. Overmars and Emo Welzl. “New Methods for Computing Visibility Graphs”. In: *Proceedings of the Fourth Annual Symposium on Computational Geometry*. SCG ’88. Urbana-Champaign, Illinois, USA: ACM, 1988, pp. 164–171. ISBN: 0-89791-270-5.

[46] Julien Pettre, Helena Grillon, and Daniel Thalmann. “Crowds of moving objects: Navigation planning and simulation”. In: *In Robotics and Automation, 2007 IEEE International Conference on*. 2007, pp. 3062–3067.

[47] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. “Computing 2D Constrained Delaunay Triangulation Using the GPU”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’12. Costa Mesa, California: ACM, 2012, pp. 39–46. ISBN: 978-1-4503-1194-6. DOI: 10.1145/2159616.2159623. URL: http://doi.acm.org/10.1145/2159616.2159623.

[48] Yipeng Qin, Hongchuan Yu, and Jianjun Zhang. “Fast and Memory-Efficient Voronoi Diagram Construction on Triangle Meshes”. In: *Computer Graphics Forum* 36 (Aug. 2017), pp. 93–104.

[49] Yipeng Qin et al. “Fast and Exact Discrete Geodesic Computation Based on Triangle-oriented Wavefront Propagation”. In: *ACM Transactions on Graphics* 35.4 (July 2016), 125:1–125:13. ISSN: 0730-0301. DOI: 10.1145/2897824.2925930. URL: http://doi.acm.org/10.1145/2897824.2925930.

[50] Ritesh Sharma, Renato Farias, and Marcelo Kallmann. “Integrating Local Collision Avoidance with Shortest Path Maps”. In: *Eurographics 2020 - Posters*. Ed. by Tobias Ritschel and Gabriel Eilertsen. The Eurographics Association, 2020. ISBN: 978-3-03868-104-5. DOI: 10.2312/egp.20201037.

[51] Guni Sharon et al. “Conflict-based Search for Optimal Multi-agent Pathfinding”. In: *Artif. Intell.* 219.C (Feb. 2015), pp. 40–66. ISSN: 0004-3702.

[52] Erik Sintorn, Ola Olsson, and Ulf Assarsson. “An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects Using Per-triangle Shadow Volumes”. In: *ACM Trans. Graph.* 30.6 (Dec. 2011), 153:1–153:10. ISSN: 0730-0301.

[53] Kiril Solovey et al. “Motion Planning for Unlabeled Discs with Optimality Guarantees”. In: *Robotics: Science and Systems*. Rome, Italy, 2015.

[54] James A. Storer and John H. Reif. “Shortest Paths in the Plane with Polygonal Obstacles”. In: *Journal of the ACM* 41.5 (Sept. 1994), pp. 982–1012. ISSN: 0004-5411.

[55] Gilbert Strang. “Maximal Flow Through a Domain”. In: *Mathematical Programming* 26.2 (1983), pp. 123–143.

[56] Vitaly Surazhsky et al. “Fast Exact and Approximate Geodesics on Meshes”. In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH ’05. Los Angeles, California: ACM, 2005, pp. 553–560. DOI: 10.1145/1186822.1073228. URL: http://doi.acm.org/10.1145/1186822.1073228.

[57] Pavel Surynek. “An Optimization Variant of Multi-Robot Path Planning Is Intractable”. In: *AAAI*. 2010.

[58] Jiri Svancara and Pavel Surynek. “New Flow-based Heuristic for Search Algorithms Solving Multi-agent Path Finding”. In: *ICAART (2)*. SciTePress, 2017, pp. 451–458.

[59] Wouter G. van Toll, Atlas F. Cook IV, and Roland Geraerts. “Real-time Density-based Crowd Simulation”. In: *Computer Animation and Virtual Worlds* 23.1 (2012), pp. 59–69. DOI: 10.1002/cav.1424. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cav.1424. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.1424.

[60] Rafael P. Torchelsen et al. “Approximate on-Surface Distance Computation using Quasi-Developable Charts”. In: *Computer Graphics Forum* 28.7 (2009), pp. 1781–1789. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01555.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2009.01555.x.

[61] J. van den Berg, Ming Lin, and D. Manocha. “Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation”. In: *2008 IEEE International Conference on Robotics and Automation*. 2008, pp. 1928–1935.

[62] Jur Van Den Berg et al. “Reciprocal n-body Collision Avoidance”. English (US). In: *Robotics Research - The 14th International Symposium ISRR*. Springer Tracts in Advanced Robotics STAR. June 2011, pp. 3–19. ISBN: 9783642194566.

[63] Wouter Van Toll and Julien Pettré. “Connecting Global and Local Agent Navigation via Topology”. In: *MIG 2019 - ACM SIGGRAPH Conference Motion Interaction and Games*. Newcastle upon Tyne, United Kingdom: ACM, Oct. 2019, pp. 1–10. DOI: 10.1145/3359566.3360084. URL: https://hal.inria.fr/hal-02308297.

[64] Ofir Weber et al. “Parallel Algorithms for Approximation of Distance Maps on Parametric Surfaces”. In: *ACM Transactions on Graphics* 27.4 (Nov. 2008), 104:1–104:16. ISSN: 0730-0301. DOI: 10.1145/1409625.1409626. URL: http://doi.acm.org/10.1145/1409625.1409626.

[65] Emo Welzl. “Constructing the Visibility Graph for n-line Segments in $O(n^2)$ Time”. In: *Information Processing Letters* 20.4 (1985). ISSN: 0020-0190.

[66] Lance Williams. “Casting Curved Shadows on Curved Surfaces”. In: *Proceedings of SIGGRAPH'78* 12.3 (1978), pp. 270–274. ISSN: 0097-8930.

[67] Erik Wynters. “Constructing Shortest Path Maps in Parallel on GPUs”. In: *Proceedings of 28th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators* (2013).

[68] Shi-Qing Xin and Guo-Jin Wang. “Improving Chen and Han's Algorithm on the Discrete Geodesic Problem”. In: *ACM Transactions on Graphics* 28.4 (Sept. 2009), 104:1–104:8. ISSN: 0730-0301. DOI: 10.1145/1559755.1559761. URL: http://doi.acm.org/10.1145/1559755.1559761.

[69] Chunxu Xu et al. “Polyline-sourced Geodesic Voronoi Diagrams on Triangle Meshes”. In: *Computer Graphics Forum* 33.7 (Oct. 2014), pp. 161–170. ISSN: 0167-7055. DOI: 10.1111/cgf.12484. URL: http://dx.doi.org/10.1111/cgf.12484.

[70] Xiang Ying, Shi-Qing Xin, and Ying He. “Parallel Chen-Han (PCH) Algorithm for Discrete Geodesics”. In: *ACM Transactions on Graphics* 33.1 (Feb. 2014), 9:1–9:11. ISSN: 0730-0301. DOI: 10.1145/2534161. URL: http://doi.acm.org/10.1145/2534161.

[71] Jingjin Yu and Steven M. LaValle. “Multi-agent Path Planning and Network Flow”. In: *Algorithmic Foundations of Robotics X*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 157–173.

[72] Jingjin Yu and Steven M. LaValle. “Planning Optimal Paths for Multiple Robots on Graphs”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2013, pp. 3612–3617.

[73] Jingjin Yu and Steven M. LaValle. “Structure and Intractability of Optimal Multi-robot Path Planning on Graphs”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI’13. Bellevue, Washington: AAAI Press, 2013, pp. 1443–1449.