UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Doctoral Thesis

A Framework for Navigation of Autonomous Characters in Complex Virtual Environments

Author:
Ramon Oliva

Supervisor:
Dr. Nuria Pelechano

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Research Center for Visualization, Virtual Reality and Graphics interaction (ViRVIG)

Departament de Ciències de la Computació (CS)

September 19, 2016

Dedicated to my family and friends.

Abstract

The simulation of large numbers of autonomous characters in real time is an important field of research in both computer graphics and robotics. The computer graphics industry has many applications that require these characters to navigate complex virtual environments in a believable manner. Such applications include video games and crowd simulation for planning or evacuation purposes. In order to create autonomous characters it is necessary to solve the problem of moving agents between two locations, at both the global and local navigation levels. Global navigation has two main components: the path finding algorithms being used and the space representation needed to abstract away the complexity of the static geometry. Once a path is found in a virtual environment, a local movement algorithm is applied to guide the agent through the free space representation from one way point to the next. Local movement algorithms steer the agent while avoiding collisions with obstacles and other agents. Although both problems have been widely addressed in the literature, there are still many aspects that need to be further improved at both the global and local level. This PhD dissertation therefore explores two areas: (1) new algorithms to automatically compute navigation meshes from complex virtual environments that can improve the global navigation; and (2) new methods to enrich the quality of the local movement given a general navigation mesh.

The main goal of this thesis has been the development of a unified framework for the movement of autonomous characters in complex virtual environments, specifically aimed at those challenging applications that require a real-time response. In order to accomplish this goal, we first focused the research on developing a fully automatic system capable of generating a *navigation mesh* (a special space partition used for navigation) for any 3D scene represented as a polygon soup. Our system, entitled *NEOGEN* (from *NEar-Optimal GEnerator of Navigation meshes*) produces a navigation mesh that satisfies two very important requirements: it provides a near-optimal space subdivision and a tight adjustment to the input geometry. The first requirement is very important in order to minimize the computational cost of path finding, while the second is important for local movement. Realistic local movement requires an accurate representation of the walkable space that allows the algorithm to make a more realistic use of the environment by letting the characters move within the whole navigable space. We call our algorithm *near-optimal* since it almost always yields a solution in the lower fourth of the optimality interval.

As presented in this dissertation, the autonomous navigation mesh generator has been built in several phases: (1) first we present a novel solution to compute a near-optimal partition for a 2D polygon with holes (*NEOGEN-2D*); (2) then we introduce a novel method to flatten the geometry given by a 3D polygon soup to obtain the 2D simple polygon with holes needed for the previous step; (3) next we present a method entitled *NEOGEN-ML* (the ML is for Multi-Layered geometry), to classify any 3D geometry into layers through a GPU

based voxelization process, and then flatten each individual 3D floor. *NEOGEN-ML* provides 2.5D navigation mesh as it can handle multiple layers with a tight adjustment in 2D (X and Z axis) but not in Y; Finally (4), with the knowledge acquired at this stage of the PhD and being aware of the limitations of the voxelization phase, we present a final method, entitled *NEOGEN-3D*, which is a novel algorithm that can handle any 3D geometry and extract a navigation mesh that adjusts to the input geometry in its three dimensions. Taken together, our contributions are a powerful tool to ease the process of creating the navigation meshes needed to move characters in complex virtual environments.

The final contribution is a real-time technique to compute paths with any desired amount of *clearance* that is independent of the underlying navigation mesh being used. Clearance values are used for both global navigation (discarding unreachable nodes due to the character's size) and local movement by strategically steering the characters. Attractor points for characters are set based on clearance, position and trajectory, thus guaranteeing that agents in a crowd will have different attractor points assigned. This reduces considerably the number of collisions between agents or against the static geometry, as well as obtaining a better usage of all the available space for navigation.

Declaration of Authorship

I, Ramon Oliva, declare that this thesis titled, 'A Framework for Navigation of Autonomous Characters in Complex Virtual Environments' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Contents

Chapter 1

Introduction

Autonomous characters are needed in many real time applications to enhance the overall realism of the environment. In order to have those character wandering such scenarios in a believable manner it is necessary to achieve realism in both path planning and natural local motion. Real time path planning for autonomous characters is a central problem in the fields of robotics, video games, and crowd simulation. Such applications require one or many characters to follow visually convincing paths in real time. Characters should move towards their destination along a realistic path, maintaining an appropriate amount of clearance with respect to the obstacles while avoiding collisions with other agents as smoothly as possible.

Figure 1.1: Virtual crowds is of main importance in games such as the *Assassin's Creed* series. Crowded virtual environments highly improve the realism of the scene.

The main contributions of this thesis have been: (1) the development of novel algorithms to create automatically navigation meshes for 2D, 2.5D and 3D geometry, and (2) to introduce novel algorithms to enhance current local movement techniques in order to achieve smoother and more natural looking paths.

Currently the most popular solutions in the literature to have characters wandering virtual environments are based on a combination of global and local movement techniques. The target of global navigation is to provide a representation of the free space of the scene that is usually obtained by constructing a navigation mesh (*NavMesh*). This especial data structure encodes the free space of the scene by splitting it into convex polygons, known as cells. A Cell-and-Portal Graph (CPG) is then obtained where a node represents a cell of the partition and a portal is an edge of the graph that connects two adjacent cells. Given a start and a goal position, paths can be calculated through the classic A* algorithm [26] or any of its variants. Once the path has been determined, and intermediate goals or way points have been assigned, the final part consists of applying a local movement algorithm at every step of the simulation, to guide the agent following the sequence of way points along the path.

Despite *NavMeshes* being widely used in complex applications such as video games and virtual simulations, there are limited applications to automatically generate a *NavMesh* suitable for path planning. In most cases, either the user needs to manually refine semi-automatically generated *NavMeshes*, or create them manually from scratch, which is extremely time consuming and a source of errors. There is therefore a need for automatic methods to generate CPGs from any given 3D environment with minimum user input required.

Previous work ([18, 60, 35, 37, 59, 80, 89]) is either not fully automatic, cannot handle any geometry, and/or provides CPGs with far too many cells or ill-conditioned cells (a cell where vertices are practically collinear which occurs when any of the internal angles is close to 0). The optimal number of cells for a partition should be lower than twice the number of convex nodes [14], therefore we consider a partition to be over-segmented when it cannot guarantee to be within the bounds of optimality. The problems with over-segmented partition are: firstly, the performance of the path finding algorithm directly depends on the dimensions of the generated graph, so the fewer cells we have, the faster this step will be; and secondly, depending on the underlying local movement algorithm being used, an over-segmented partition may end up with characters walking in zig-zags through a long convex space as they are forced to go through unnecessary portals, or in portals so close together that they add too many unnecessary nearby attractors and therefore complexity when trying to achieve natural looking local movement.

The architecture developed for this PhD thesis presents a completely novel approach, as it overcomes all limitations described above and presents an entire pipeline to go automatically from a 3D virtual environment given as a polygon soup, to the final *NavMesh* which adjusts tightly to the original geometry [65].

The target of local movement techniques is to provide a mechanism for the autonomous characters to move from one location to the next one in the computed path in a smooth and natural manner, while avoiding collisions against static and dynamic obstacles. These methods are generally driven by setting intermediate goal points (commonly known as way points) within the portals that work as attractors to steer the agents in the right direction ([67, 73, 74, 79, 92, 93]).

When simulating a variety of characters, it is convenient to be able to calculate the shortest route for the characters based on their size. If we think of applications such as video games, this would allow a skinny character to escape from a large monster by running through a narrow passage. Efficiency is also a key aspect, as many characters may require a path computation in the same frame over a large scenario, so only a small fraction of a second is available. Additionally, the method used to compute the way points is also critical in order to produce visually convincing routes. Most proposed solutions are based on computing a single point over the portal (usually at the center, or at the endpoints of the portal), so all agents share the same way point. This results in agents that tend to line up when approaching the portal from the same side, or form bottlenecks when several agents attempt to cross the portal coming from different directions. These artifacts reduce artificially the flow rates through portals and thus the overall time to reach their destination, and are also perceptually unpleasant. For this thesis, we have developed an algorithm that can run in real time assigning different way points to different characters, mitigating these artifacts.

Previous work ([9, 18, 60, 37, 43]) is either bounded to a specific amount of clearance, does not address correctly the problem of clearance or only work with a specific type of navigation mesh. On the contrary, our current method [66] is able to deal with an arbitrary amount of clearance and can work with any type of *NavMesh* even if cells are not strictly convex.

1.1 Contributions

The goal of this thesis has focused mostly on the development of methods to automatically generate automatic navigation meshes. From the beginning of this work, we had a very clear picture of which should be the main characteristics of such *NavMeshes*: to adjust as tightly as possible to the original geometry, to lower us much as possible the resulting number of cells in order to provide an efficient structure for path finding in real time, to reduce the user input to obtain the final *NavMeshes*, and to provide a data structure that aid in reducing or eliminating typical local movement artifacts.

Therefore in this thesis we propose four methods to obtain the navigation meshes and new algorithms for calculating clearance and enhancing local movement. More specifically the contributions of this thesis are:

- ***NavMesh* Generator for 2D floor maps**: The first contribution of this work presents a fully automatic algorithm to generate a navigation mesh for a given floor map given as a 2D single polygon with holes. The algorithm introduces the concept of convexity relaxation based on local movement capabilities to further reduce the number of cells in the navigation mesh. The results provide near-optimal navigation meshes.

- ***NavMesh* Generator for 2.5D geometry**: The second contribution consists of processing a 2.5D virtual map, in order to produce a 2D polygon with holes that can be used as an input for our first contribution. We define a 2.5D map as a virtual scene represented as a 3D polygon soup where all the obstacles can be safely projected to the 2D plane, therefore ignoring the height component.

- ***NavMesh* Generator for multi-level 3D geometry**: The third contribution manages to process any 3D geometry given as a polygon soup, and through a voxelization process decomposes the scene into layers, that are then projected onto 2D maps to create *NavMeshes* for each level. Then the different *NavMeshes* of each level are automatically linked to obtain the overall *NavMesh*. Each of the *NavMeshes* tightly adjusts to the 2D projection of the geometry.

- **3D Automatic *NavMesh* Generator**: Finally we present a novel algorithm to obtain 3D *NavMeshes* that can adjust perfectly to the three dimensions of the original geometry, and eliminates the need of voxelizing the geometry to process the different layers.

- **Clearance and dynamic way points**: An automatic method to create *NavMeshes* that guarantee clearance for a variety of characters, both at the level of computing paths and at the level of local movement by assigning adequate way points. The dynamic way points algorithm presented further reduces the number of collisions and provides smother paths with a better usage of free space.

1.2 Organization of this document

This thesis is organized as follows. Chapter 2 is dedicated to the state of the art, where we review in depth the most important previous works in the fields of *global movement*, *local movement* and *path finding*. The chapter ends with a section where we discuss the limitations of the previous work and we briefly describe how the work presented in this thesis overcomes the found limitations.

Our contributions to the field of *global movement* based on navigation meshes, are exposed and discussed in Chapters from 3 to 6. More specifically:

Chapter 3 describes *NEOGEN-2D*, our first contribution to automatically generate near-optimal navigation meshes for any given 2D simple polygon with

holes. Also, we introduce a novel concept to further reduce the final number of cells of the partition.

In Chapter 4, we present a GPU based method that takes as an input a 3D scene representing a single floor plan, and automatically computes a 2D polygon with holes abstraction that fits with the input required by *NEOGEN-2D*, previously introduced in Chapter 3. Additionally, a GPU based version of this algorithm is described.

Chapter 5 presents *NEOGEN-ML*, a novel full framework to process any given multi-layered 3D scene (such as a building) and splits it into floor maps for each level that can be then input independently to our 2D algorithm. The individual *NavMesh* created for each level, are joined later on into a single *NavMesh* representing the walkable space of the whole scene.

In Chapter 6 we introduce *NEOGEN-3D*, another approach that is able to compute a navigation mesh for any 3D model representing a scene with an arbitrary topology, thus overcoming the main limitation of *NEOGEN-ML* presented in Chapter 5, which was restricted only to multi-layered environments. Although this chapter is ongoing work, we present initial results and discuss the details of the algorithm and we are preparing a paper for submission.

Our contributions to *local movement* are explained in Chapter 7, where we introduce *ExACT*, a novel algorithm to compute paths with any desired amount of clearance for any type of *NavMesh*. We also propose a dynamic technique for way point assignation that considerably improves the quality of the simulation by reducing the number of collisions against both the static and dynamic geometry. In addition, we propose a novel and straightforward *A** encoding, which solves the problem of having cycles when computing paths with *clearance* on *NavMeshes*.

Finally, in Chapter 8 we present the conclusions of this thesis as well as possible future work that could open new research topics from this work.

1.3 Publications

1.3.1 JCR Journals

1. **R. Oliva**, N. Pelechano. *Clearance for Diversity of Agents' Sizes in Navigation Meshes*. Computers & Graphics. vol. 47. pp. 48-58. April (2015).

2. **R. Oliva**, N. Pelechano. *NEOGEN: Near Optimal Generator of Navigation Meshes for 3D Multi-Layered Environments*. Computers & Graphics. vol. 37. no. 5. pp. 403-412. (2013).

1.3.2 International Conferences

1. W.van Toll, R. Triesscheijn, M. Kallmann, **R. Oliva**, N. Pelechano, J. Pettré, R. Geraerts. *A Comparative Study of Navigation Meshes.* ACM SIGGRAPH conference on Motion in Games (MIG'2016). October 10-12. San Francisco (California). (2016).

1. **R. Oliva**, N. Pelechano. *A Generalized Exact Arbitrary Clearance Technique for Navigation Meshes.* ACM SIGGRAPH conference on Motion in Games (MIG'2013). November 7-9. Dublin (Ireland). Article 81, 8 pages, pp: 103-110. (2013).

2. **R. Oliva**, A. Beacco, N. Pelechano. *Computing Exact Arbitrary Clearance for Navigation Meshes.* ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA'2013). Poster. July 19-21. Anaheim, California. (USA). (2013).

3. **R. Oliva**, N. Pelechano. *Automatic Generation of Suboptimal NavMeshes.* The Fourth International Conference on Motion in Games 2011 (MIG'2011), Edinburgh (UK), November 13-15, 2011. LNCS, Vol 7060, pp: 328-339. (2011).

1.3.3 National Conferences

1. **R. Oliva**, N. Pelechano. *NavMeshes with Clearance for Different Character Sizes.* Congreso Español de Informática Gráfica. CEIG'13. Sección Española de Eurographics (EGse). Madrid (Spain) September 17-20, (2013).

2. **R. Oliva**, N. Pelechano. *A GPU Accelerated Method for the Automatic Generation of Near-Optimal Navigation Meshes.* Congreso Español de Informática Gráfica. CEIG'12. Sección Española de Eurographics (EGse). Jaén (Spain) September 12-14, (2012).

1.3.4 Book Chapters

1. **R. Oliva**, N. Pelechano. *Navigation Meshes.* In Virtual Crowds: Steps Toward Behavioral Realism, pp. 59-74. Morgan & Claypol, (2015).

Chapter 2

State of the art

In this chapter we review the state of the art regarding navigation of autonomous characters in virtual environments. In Section 2.1, we give an overview of the main *global movement* methods proposed. Then, different *local movement* techniques are discussed in Section 2.2. Finally, Section 2.3 presents the current work done in *path finding*. At the end of this chapter we present a discussion section where we summarize the main limitations of the current work on the aforementioned fields and how we tackle these problems in the present thesis.

2.1 Global Movement

The aim of *global navigation* methods is to generate a special data structure that, given a virtual environment and a set of navigational capabilities (such as the dimensions of the virtual agents or the maximum slope they can traverse), it encodes the available free space of the scene. We can classify those methods into 4 main blocks according to the type of data structure they generate: *Regular Grids*, *Roadmaps*, *Corridors* and *Navigational Meshes*. Regardless of the specific *global navigation* method being used, this data structure is subsequently translated into a connectivity graph that can be used for a path finding algorithm (such as the popular $A*$ or any of its variations) in order to be able to find a valid path between the current position of the agent and a goal position in the virtual world.

2.1.1 Regular Grids

In this approach, the navigational graph is constructed by overlapping a 2D grid with the virtual scene. Each cell of the grid is then marked as *accessible* if it is in the free space or *obstacle* otherwise. The *accessible* cells represents the nodes of the graph and an edge is created between each pair of adjacent and *accessible* cells.

In 2001, Tecchia et al. [15] presented a complete framework for crowd rendering and simulation of multi agent systems. The described system, called *Agent Behavior Simulator (ABS)*, is based on a 2D-grid with an associated four-layered structure that determines the local rules applied to each autonomous agent in the simulation. Each layer can be seen as a map aligned with the underlying grid and describes a different aspect of an agent's behavior. Therefore, each cell of the grid corresponds to an entry to each layer. The four different layer behaviors proposed are the following ones:

1. Inter-collision detection layer: This layer gathers the agent-to-agent collision detection. Before moving to a new cell, an agent checks if the target cell is already occupied by another agent or it is free.

2. Collision detection layer: It corresponds to collision detection against the static geometry of the environment. An image is used as an input, where white pixels determine accessible cells whilst black pixels determine the location of an obstacle (see figure 2.1(a)).

3. Behavior layer: This third layer encodes more complex behaviors for each local region of the grid. A color map is used as an input file where the user associates a specific color with a specific behavior. An example of behavioral map is shown in figure 2.1(b), where reddish pixels reflects how agents are attracted by some points of interest, such as a bus stop or a shop-window.

4. Callback layer: The last layer allows the user to associate a *callback function* to a cell of the grid in order to simulate more complex behaviors such as pushing a button to call the elevator or climbing aboard a bus on its arrival.

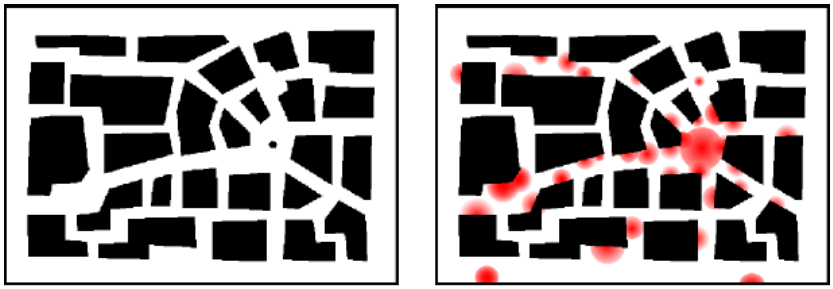

Figure 2.1: (a) An example of a collision map for a flat terrain. Obstacles are encoded in black and accessible areas are encoded in white. (b) depicts an example of attraction map for the behavior layer. (Tecchia et al., 2001 [15])

The combination of the aforementioned layers, determine the final behavior of the autonomous agent. For example, this system could be used to simulate an agent walking along the street whilst avoiding other agents (layer 1) and typical obstacles of an urban environment such as rubbish bins and streetlights (layer 2). When it reaches the cell corresponding to the bus stop, it waits for the bus (layer 3) and finally, when the bus arrives, a *callback* is activated and makes the agent climbing into the bus (layer 4).

In 2004, Chenney [7] introduced a grid-based technique called *flow tiles*, aimed to simulate flows ranging from environmental effects such as the the flow of a river within its banks, to the flow of autonomous characters in urban virtual environments. Each *flow tile* describes a small, stationary region of velocity field and different *flow tiles* can be pieced together in order to construct a larger stationary field. The tiles presented in this work are stationary, meaning that the velocity field does not change over time.

An important limitation of *flow tiles* is that streamlines never cross. This means for example that using this technique, we cannot simulate characters crossing each other at the middle of an intersection. Furthermore, the work does not consider collision detection nor any behavior other than flowing through the velocity fields. Therefore, the situations where *flow tiles* can be used for crowd simulation are very limited.

Loscos et al. [56] proposed another method based on 2D-grids for simulating pedestrians that automatically generates sidewalks and crossing regions for a given urban virtual environment. First, a 2D map of a certain resolution is automatically constructed by rendering the virtual environment from the top

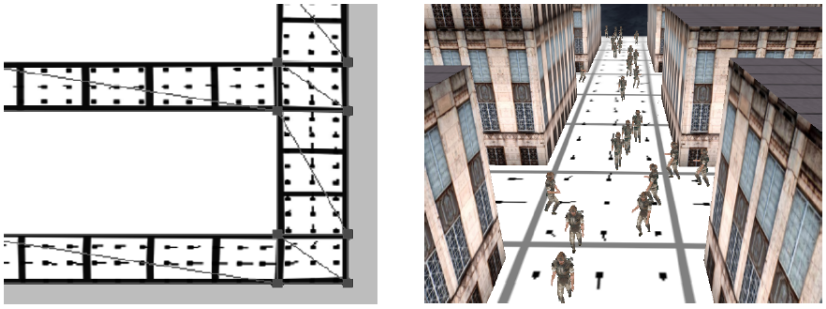

Figure 2.2: The flow tiles mapped to the accessible area of an urban environment (left) and some characters flowing through the streets driven by the velocity fields described by the flow tiles (right). (Chenney, 2004 [7])

using an orthographic camera. The fragments corresponding to ground are colored in white, whilst the fragments corresponding to buildings are colored in black. Then, the sidewalks are constructed on the previous 2D map by enlarging the area covered by the buildings using a convolution filter. Next a graph of goals is constructed by detecting the corners on the 2D map. Finally, for every pair of goals, a crossing area is created if they are on opposite sides of the street. Figure 2.3 shows an example of the application of the proposed method.

Figure 2.3: On the left, the 2D color color map encodes the ground (white), buildings (black), sidewalks (dark gray) and crossings (light gray). On the right, an example of a simulation in a urban virtual environment. (Loscos et al., 2003 [56])

Grid structures can also be extended to handle multi-layered terrains as in [2]. The idea consists of considering only the cells on the upper surface of objects on which humans can walk. Cells are connected based on whether they are considered insurmountable obstacles (human can not climb, cross or pass through), or surmountable obstacles (human can step on, step down, cross or step over).

2.1.2 Roadmaps

The *roadmap* approach captures the connectivity of the free space by using a network of standardized paths (lines, curves). Different approaches can be used to compute a *roadmap*. The visibility graph connects vertices of the environment geometry if the segment joining the two points does not intersect with the geometry of the scene. In [1], a visibility graph representation is proposed for the particular problem of path-planning in a Real Time Computer Strategy Game. Typically on such applications, a virtual battle is modeled by simulating the behaviors of a large number of individual objects that move on a 2D terrain or attack other objects on the user's orders. In addition, the user is only able to see a portion of the scene at a time, so the method proposed simulates in an accurate form only the agents that are visible from the point of view of the user. The behavior of the rest of the agents is determined with an approximated method, but much faster. This allows them to support a large number of agents at interactive rates.

The main issue with the visibility graph representation is the trade-off between paths quality and performance, since the cost of computing the search algorithm is greatly related to the number and position of the vertices of the visibility graph. Hence if we want to obtain smooth paths, we need to sparse a large number of vertices on the geometry scene, but the search algorithm time will increase and it can be problematic especially in applications that require a real-time response. On the contrary, if we want to improve the performance of the graph search algorithm, the number of vertices spread must be low, which may lead to unnatural trajectories for the characters, as if they were walking on rails.

An alternative representation for *roadmaps* is to compute the generalized Voronoi Diagram [62]. An approximation of the generalized Voronoi Diagram can be computed using the graphics hardware [31]. The property of *roadmaps* generated by this way is to maximize the clearance with obstacles. In [87], a system entitled *AERO* is presented that uses a generalized Voronoi Diagram to compute a *roadmap* that defines the free space with respect to static geometry. In addition, the *roadmap* can be updated in real-time in order to avoid collisions with dynamic obstacles, such as other agents. The links between two points of the free space can be deformed in presence of a dynamic obstacle. Those links have a maximum elasticity and are broken (removed) when this value is exceeded, disconnecting both points. Figure 2.4 illustrates this situation. When a link is removed, it is placed in a list and reinserted when the straight line path between the two points is free of obstacles. The most important limitation of the system proposed is that the dynamics formulation to update the links can potentially result in an agent getting stuck in a local minimum of the geometry. In other words, they are not able to provide convergence guarantees on the existence of a collision-free path for each agent in all environments.

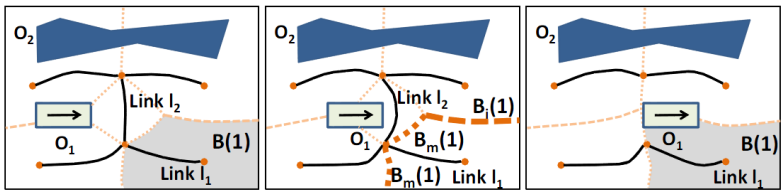

Figure 2.4: The obstacle O1 is moving towards link l2 (left) and it is deformed (center). When the elasticity of the link is exceeded, the link l2 is removed (right). (Sud et al., 2007 [87])

An important limitation of the *roadmap* approach is that it is usually restricted to a single 2D environment. However, virtual applications such as video games may offer real 3D scenarios where the *roadmap* approach is hard to apply. In [75], Rodriguez et al. use a *roadmap* data structure that can work in building-like environments conformed by levels and stairwells connecting adjacent levels. Each level represents a 2D floor and a visibility graph is constructed as usual (sample some positions on the floor, then connect all those samples that do no intersect with the geometry). The individual visibility graphs of each level are connected on the stairwells, conforming this way the *roadmap* of the whole scene.

One limitation of this approach is that it only works in very specific scenarios, basically building like environments. Additionally, there is the lack of an automatic method for connecting the different visibility graphs of each individual level and also as we have seen previously, visibility graphs are not the best way of computing roadmaps.

In [57], the *Probabilistic Roadmap (PRM)* approach is introduced. First, a *roadmap* is constructed by sampling the virtual environment and determining if whether such samples are valid (i.e., they lie in the free space of the scene) or not. Valid samples conforms the nodes of the graph and each node is intended to connect to some neighboring nodes, typically either the *k-nearest* neighbors or all neighbors inside a specific radius. The connection between the node and each of the chosen neighbor is checked for intersections with the obstacles. If the connection is free of collisions, an edge is added to the graph. Samples and connections are added to the graph until the resulting *roadmap* is dense enough. Once the *roadmap* is constructed, a path between two points on the virtual scene is found by adding the start and goal position to the graph and applying some path finding algorithm, such as $A*$.

The *PRM* approach is provably probabilistically complete, meaning that as the number of sampled points increases without bound, the probability of not finding a path if one exists approaches to zero. The rate of convergence completely depends on the properties of the virtual scene. Intuitively, if each sample can 'see' a large fraction of the free space, the algorithm will converge faster than in those virtual scenes with low visibility, containing intricate corridors and obstacles.

Many variants exists of the basic *PRM* technique, which vary in the sampling strategy and/or the connection strategy in order to improve the overall performance. In the work by Geraerts and Overmars [19], several approaches are discussed and compared in a common set of environments.

In [3], Bayazit et al. explore the use of *roadmaps* applied to groups of agents moving in some kind of coordinated form. This type of group behaviors, usually referred as *flocking behaviors* are very common in nature. For example, birds fly in flocks, fish swim in schools, and sheep move as a herd. The addition of a global data structure (a *roadmap* in this specific work) adds crucial information of the virtual environment, resulting in more sophisticated flocking behaviors that would not be possible using only local information.

Agents in the flock are able to share information between them by updating the common *roadmap*. For example, one of the simulated behaviors consist on making some agents in the flock to visit every vertex and edge of the *roadmap*. In this case, the weight of each edge is initialized to one. As members of the flock traverse an edge, its weight is increased, therefore the individual flock members are biased toward relatively unexplored areas of the *roadmap* due to the cost computation on the path finding stage [26]. Although this way of sharing information is fast, each flock behavior may require a different initialization and update of the underlying *roadmap*, so in order to simulate multiple groups exhibiting different behaviors, a copy of the roadmap for each group would be necessary.

Roadmaps also have been used to simulate *shepherding behaviors*. In the work by Lien et al. [51] a group of shepherds can work cooperatively in order to efficiently control the flock. This work assumes that each shepherd is independent of the other shepherds and that there is no communication between them. The tasks of the shepherds is to steer the flock to desired locations or along desired routes. Additionally, shepherds unites separated flock groups.

The *roadmap* of the scene is used in order to define *milestones* (the position towards which the shepherd attempts to steer the flock) and *steering points* (the position towards the shepherd moves himself in order to influence the movement of the flock) for each shepherd. Those concepts are depicted in figure 2.5 (left). The *steering points* are placed in a way such that the shepherds will be arranged in a particular formation with respect to the flock and the desired direction of movement. The position of each shepherd in the formation is decided by using the distance between shepherds and the *steering points*. Figure 2.5 (center, right) shows different two different shepherding formations.

Wein et al. introduce the Visibility-Voronoi Complex [98], which as the name indicates, combines visibility graphs with Voronoi diagrams by evolving from the first one to the second one based on a parameter. The resulting diagram can be used for calculating short and smooth paths with clearance.

The work by Pettré et al. [68] creates a navigation graph that is suitable for multi-layered terrains. It fist computes a roadmap of the static environment

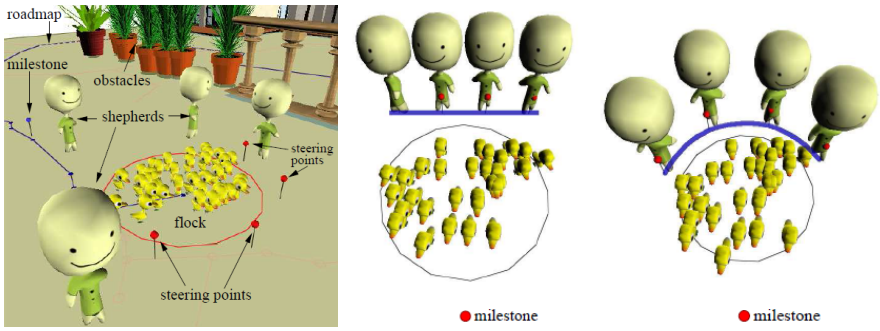

Figure 2.5: The different concepts used on the multiple shepherding behavior (left); two different shepherding formations: straight line formation (center) and arc formation (right). (Lien et al., 2005 [51])

using Voronoi diagrams, and then creates cylinders in a compact manner along the Voronoi diagram 2.6. Each cylinder is a cell of the navigation graph and the intersection between adjacent cylinders represents an edge between their corresponding cells in the graph.

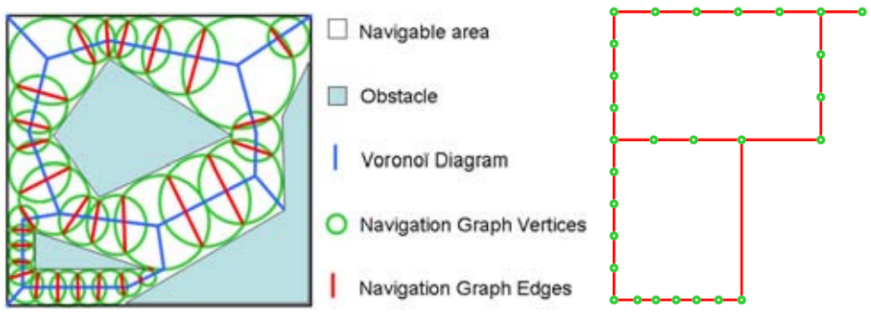

Figure 2.6: Geometric representation of a navigation graph created with cylinders along the Voronoi diagram (Pettré et al., 2005 [69])

In the work by Wardhana et al. [97], they generate navigation graphs by computing relevant way points and then creating transitions between those way points based on whether the character can move between them. One of the main advantages of their method is the ability to handle a variety of movement for the characters, such as walking, jumping or flying. However, they need to create a *roadmap* per each kind of movement.

2.1.3 Corridors

The *corridors* technique can be seen as an extension of the classic *roadmaps*, but the resulting graph also contains some kind of *clearance* information. This is acquired by sampling the edges of the graph and for each sample, a radius value is added which implicitly defines a 2D disk centered on that point. The radius value must guarantee that the disk does not intersect any virtual obstacles, thus an autonomous character can move freely inside the disk without colliding with the static geometry. When querying for a path between two specific positions in the map, the result is a backbone path with clearance information, formerly known as *corridor*.

In [40], *corridors* are used in order to keep coherence for groups of agents. This is particularly useful for example in any *Real Time Strategy (RTS)* game, as all agents integrating the same unit should move as a whole and must exhibit coherence at all time, like passing on the same side of obstacles and waiting for fellow group mates to catch up, maintaining the group together. In this approach, a *Probabilistic Roadmap* is used in order to compute a *corridor* for a specific agent of the unit. The corridor must guarantee that at any point, the sampled clearance is greater or equal than the agent radius. Then, all agents integrating the unit are forced to follow this path. In order to avoid agents lining up at the center of the path, the agents are distributed along the corridor by applying social forces which avoids collisions between agents but keeps the group altogether by limiting the maximum longitudinal and lateral dispersion.

The *Corridor Map Method (CMM)* was firstly introduced by Geraerts and Overmars [20]. This data structure represents the free space of a scene using a graph whose edges correspond to collision-free corridors. The *CMM* is used to extract a *corridor*, which connects the current position of the autonomous character with its desired goal location. Such a *corridor* consists of a backbone path that is used to steer the movement of the characters.

An efficient way of computing the *CMM* for a given virtual scene was proposed in [21]. The *CMM* is constructed by first computing the *Generalized Voronoi Diagram (GVD)* of the scene, which is a decomposition of the free space into Voronoi regions such that all points p in a Voronoi region $R(p)$ are closer to a particular obstacle than to any other obstacle in the environment. The boundaries of the Voronoi defines a graph, where each node is located at a non-convex corner (induced by at least two obstacles) or at a location at which three or more edges of the graph meet. An edge connects two nodes and is created in the boundary between two adjacent Voronoi regions. A GPU based method is used in order to efficiently compute an approximation of the *GVD* [31] (see figure 2.7(a-c)).

Once the *GVD* has been computed, the *Corridor Map* is extracted by sampling the edges of that graph. This step is illustrated in figure 2.7(d). Each sample point also stores the radius of the largest empty disk centered at that point, defining a maximum clearance disk. A sequence of contiguous disks is

referred as a *Corridor* (figure 2.7(e)).

Notice that the *corridors* created this way guarantees a maximum amount of *clearance* with the obstacles. However, it is not easy to compute the shortest minimum-clearance path because this type of *corridor* does not provide a proper description of corridor's boundaries. Another disadvantage of having samples is that the free space is not fully covered. There is a trade-off between space coverage and path computation efficiency.

In order to overcome the previous limitations, the *Explicit Corridor Map (ECM)* was proposed [18], which allows to compute the shortest path with any desired amount of clearance inside a *corridor*. The *ECM* is constructed by assigning to some selected samples of the original *CMM*, their left and right closest points on the obstacles. This produces an explicit description of the corridor's boundaries, as can be seen in figure 2.7(f-h). An *Explicit Corridor* can be shrunk in order to provide a sequence of *way-points* defining the shortest path with clearance [43] (see figure 2.7(i)).

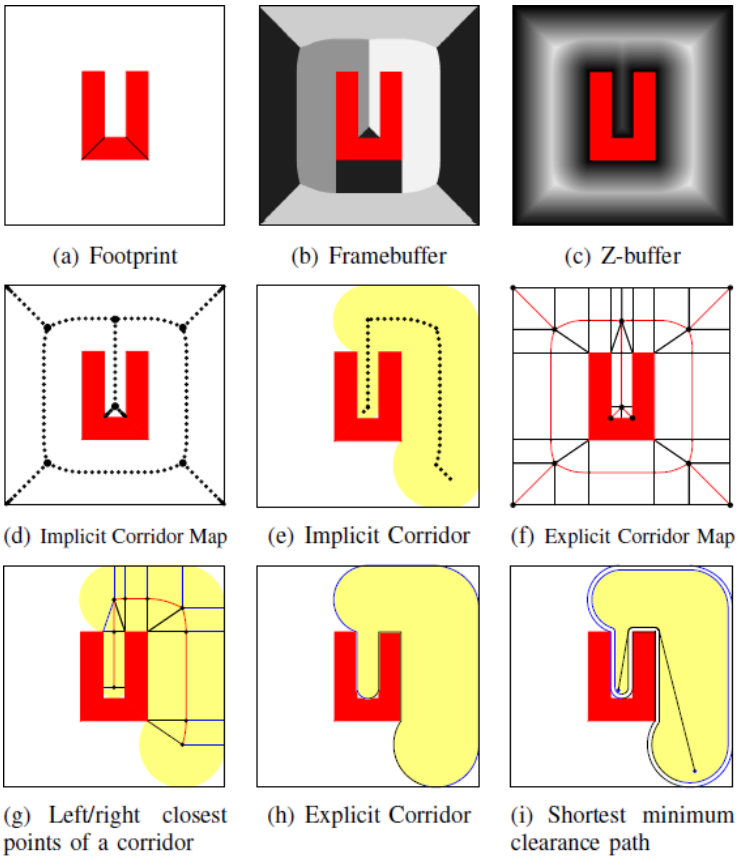

Figure 2.7: Construction of the *Generalized Voronoi Diagram* (a-c) using the GPU; the *Corridor Map Method* (d) and an example of *(Implicit) Corridor* (e); the *Explicit Corridor Map* (f-g) and an example of *Explicit Corridor* (h); the shortest path with a specific amount of clearance in a given (shrunk) *Explicit Corridor*(i). (Geraerts, 2010 [18])

2.1.4 Navigation Mesh

The main limitation of the *roadmap* representation is that it only contains information about which locations of the scene are directly connected, but it does not describes the geometry of the scene nor where the obstacles are, and avoidance of dynamic obstacles is usually a hard task and not always possible, as exposed in [86]. The Cell Decomposition method consists of the partition of the navigable geometry of the scene into convex regions, guaranteeing that a character can move from two points on the same cell following a straight line, without getting stuck in local minima. This particular decomposition is usually known as *navigation mesh (NavMesh)* [80] and a Cell-and-Portal Graph (CPG) can be obtained to compute paths free of obstacles. The collisions against movable obstacles such as other agents, is solved by using a local movement algorithm [74] or by dynamically modifying the *NavMesh*. We can classify the *navigation mesh* methods into methods that partitions the scene using polygons of a fixed number of sides (usually triangles or quads) or methods that produces a partition into cells of an arbitrary number of sides.

2.1.4.1 Fixed cell-shaped NavMeshes

Valve's Game Engine has an automatic NavMesh generator method based on subdividing the virtual map by Axis-Aligned quads of arbitrary size [91]. This method gives satisfactory results mostly for scenes with Axis-Aligned obstacles, such as some indoor scenes. However when there are complex obstacles with random positions and orientations, the partition obtained does not adapt well to the contour of the obstacles, as can be seen in figure 2.8. In addition, the method has important limitations when handling geometry that contains very steep stairs, ramps or hills. In these situations, the resulting *NavMesh* would not cover the entire map, thus requiring that the user manually completes the *NavMesh*.

Triangular meshes are commonly used to represent a *navigation mesh*. In [39, 35], a dynamic Constrained Delaunay Triangulation (CDT) is used to represent the walkable area of a scene. The resulting *NavMesh* adapts perfectly to the contour of the obstacles compared to old grid based methods. In addition, the resulting Cell-and-Portal Graph (CPG) obtained is much smaller, therefore reducing the time to compute the path between two given points in the scene. The method proposed in [39, 35] can be divided in 3 main steps: Given a set of polygonal obstacles, a Constrained Delaunay Triangulation having as constraints the edges of the obstacles is constructed. During run-time obstacles are allowed to be inserted, removed or displaced and the CDT is able to dynamically take into account these changes. Once the CDT is computed, given a starting and a goal point, a graph search is performed over the adjacency graph of the triangulation defining the shortest channel connecting both points. A channel is the sequence of adjacent triangles from the starting point to the goal point. Obtained channels are equivalent to triangulated simple polygons, and thus the last

Figure 2.8: Axis-Aligned quads do not adapt well to the contour of general obstacles. (VALVE, 2005 [91])

step consists in computing the shortest path joining the starting and the goal points inside the channel. For this, the funnel algorithm [6, 28, 48] is applied. The funnel algorithm is able to compute the shortest path inside a triangulated simple polygon in linear time. It is a very popular method in games to calculate the exact character's trajectories. Extensions have been made to include clearance or to compute shortest paths for disks of a specific radius [54, 37, 55]. However shortest paths may not always be the most desirable trajectory, and sometime what we need is to determine the exact subsegment over the portal that the character could walk through without collision, as we will describe in chapter 7.

The CDT provides support for dynamic obstacles, although the performance of the application greatly depends on the complexity of the CDT, as well as on the complexity and number of constraints being moved.

In [10] the CDT technique is compared against grid-based maps of real commercial video games. The results show that the use of a CDT to represent the walkable space dramatically reduces the computation time to find a path between two points, compared to the grid representation of the same map. In [36], more uses of triangular *NavMeshes* are explored, such as the automatic placement of agents in the free space and efficient computation of ray-obstacle queries. In a recent publication [71], a method for computing the CDT using the GPU has been presented. The implementation is done using the CUDA programming model [61] on NVIDIA GPUs and the results show that it runs several times faster than any CPU method.

In [37, 38], a new type of triangulation called Local Clearance Triangulation (LCT) based on a CDT is presented. It allows computing paths free of obstacles with arbitrary clearance. Given a triangle of a channel, it will be traversed by crossing two edges. Let a, b, c be the vertices of this triangle and consider that the free path crosses the triangle by first crossing the edge ab and then the edge bc. In this case, the shared vertex b is called traversal corner. This particular traversal is called τ_{abc}. The traversal sector is defined as the circle sector between edges ab, bc and of radius $min(dist(b,a), dist(b,c))$ and the traversal clearance $cl(a,b,c)$ is the distance from the traversal corner to the closest constrained edge inside the traversal sector. If such constraint does not exist, the traversal clearance is the radius of the traversal sector. Figure 2.9 (left) illustrates this situation.

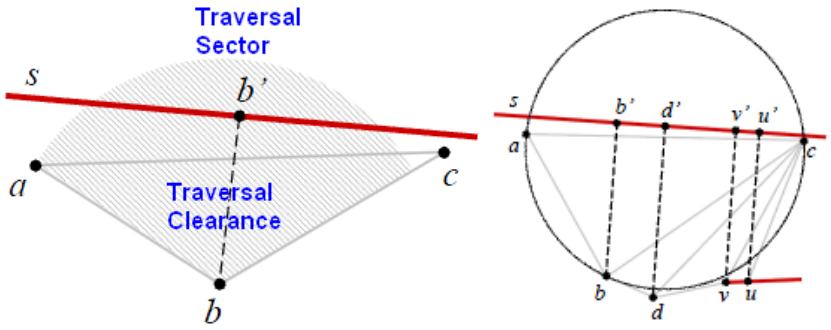

Figure 2.9: The traversal sector and the traversal clearance of a triangle traversal τ_{abc} (left); the vertex v is a disturbance of the traversal τ_{abc} (right). (Kallmann, 2014 [38])

Given the situation depicted on the figure 2.9 (right), a vertex v is a disturbance to traversal τ_{abc} if v can be orthogonally projected on ac, v is not shared by two collinear constraints, $dist(v,s) < cl(a,b,c)$ and $dist(v,s) < dist(v,c)$. Given the definition of disturbance, a traversal τ_{abc} has local clearance if it does not have disturbances. The Local Clearance Triangulation (LCT) is therefore, a CDT with all traversals having local clearance. The local clearance property of the LCT guarantees that simple local clearance test per triangle traversal is enough for determining if a character of a determined bounding radius can traverse a given channel without any intersection with constraints. In the case of the CDT, the local clearance test is not enough to guarantee that the path has enough clearance.

The proposed procedure for achieving a LCT is based on iterative refinements of disturbed traversals. The algorithm starts with the computation of the CDT of the initial set of constraints. If all traversals are free of disturbances, we have obtained the LCT and the process ends. On the other hand, if there are traversals with disturbances, those must be refined. That is, the constraint of the disturbance is refined with one subdivision point in the constraint, as can be seen in figure 2.10. Every time a constraint is refined, it is replaced by two

new sub-segments. After all disturbed traversals have been processed, a new set of constraints and a new triangulation is obtained. However, this triangulation is not guaranteed to be free of disturbances and the process has to be repeated until a triangulation free of disturbances is obtained.

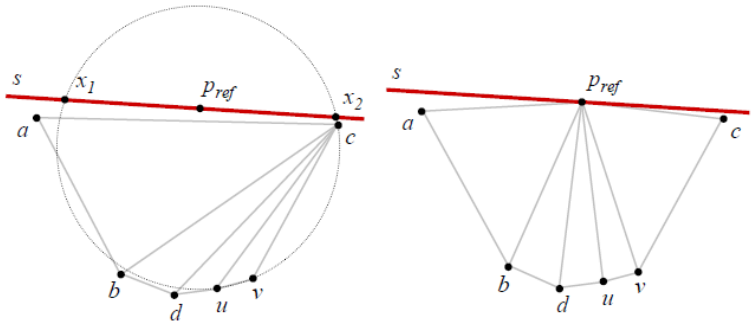

Figure 2.10: The point of refinement Pref is computed as the midpoint of the intersection points with the constrained edge s and the circle(u,v,c) (left); once Pref has been computed, all the vertices are joined to this new point (right). (Kallmann, 2014 [38])

Once the LCT has been computed, a graph search is performed to find the channel joining the starting and the goal point. Triangle traversals are only accepted if the local clearance test is satisfied, guarantying that the resulting channel will have enough clearance. Then, the problem is reduced to find the path inside a channel and an extended version of the funnel algorithm to take into account clearance is presented. The result is a set of straight segments and arcs that defines a path free of obstacles, with enough clearance.

The main limitation of the representation proposed is that the refinement process to obtain the LCT from the initial CDT, introduces new segments into the triangulation and hence, the resulting number of cells is increased with respect to the original CDT, dealing to an over-segmented partition. As proven in the work by Kallmann [37] the iterative process to obtain the LCT converges.

In addition, the support for dynamic obstacles seems not to be possible as described in the dynamic CDT (at least in real-time), because the insertion and movement of constraints may introduce disturbances in triangle traversals that must be refined to obtain the corresponding LCT.

Topoplan [46] is an application that automatically generates a Cell-and-Portal Graph given a virtual environment defined as a mesh of triangles. Firstly, they apply a simplification step consisting in representing the mesh with 3D planar polygons instead of triangles. Those polygons are computed by partitioning the set of mesh triangles into sets of coplanar and connected triangles. Then, an exact 3D prismatic spatial subdivision of the 3D model is computed. The aim of this step is to organize a set of 3D polygons in order to capture ground connectivity and identify floor and ceiling constraints. It represents the envi-

ronment by a set of vertical 3D prisms dividing the 3D model into layers. The workflow of the algorithm is described in figure 2.11. Step (1) presents a simple environment composed of two triangles. The first step consists of projecting the boundaries of each 3D planar polygon on the XZ plane. This produces a set of 2D segments (3) on which a CDT is computed (4). The prismatic subdivision is then obtained by associating to each 2D triangle of the CDT, the set of 3D polygons partially projecting on it. It is computed through ray casting. Once this relation is computed, for each triangle t of the CDT and for each associated polygon p, a 3D cell is computed such as this cell is supported by the plane supporting p and its projection on XZ plane is exactly t (5). Let $prism(t)$ be the list of all 3D cells which are exactly projecting on a triangle t of the CDT. The $prism(t)$ is ordered along the vertical axis in the increasing order of the average vertical coordinates of the vertices of the 3D cells. This spatial subdivision allows them to identify floor and ceiling. Once the prism decomposition has been obtained the navigable zones of the environment are extracted taking into account some humanoid characteristics, such as the maximum traversable slope and the maximum height that a character can overcome with a step. Those zones are then grouped into a set of 2.5D surfaces. A 2.5D surface is defined as the union of interconnected zones that do not overlap. A Constrained Delaunay Triangulation is computed over each of this surface to obtain the final CPG usable for path planning.

The first problem of Topoplan is that it needs as an input a clean mesh, i.e. it does not contain degenerated triangles nor triangle intersections (except obviously, on shared vertices and edges). This is a strong requirement that hardly will be accomplished, because during the modeling process, intersections on the geometry are something common. In addition, the method that they use to sort the set of cells in a prism could deal to a wrong sort on some cases and hence, the identification that they do into floor and cell would not be correct. And finally, the method proposed is very costly, as can be deduced by the description of the method. The results show that it requires more than 15 minutes to compute the CPG for an environment of just 120k triangles.

Recently, Berseth et al. [4] presented ACCLMesh, a triangular *navigation mesh* based on acceleration and curvature between adjacent triangles. Since the resulting mesh is built using the triangles of the input geometry, it avoids mesh intersections and it adjusts well to the geometry. A limitation of curvature based methods, is the inability to handle steps.

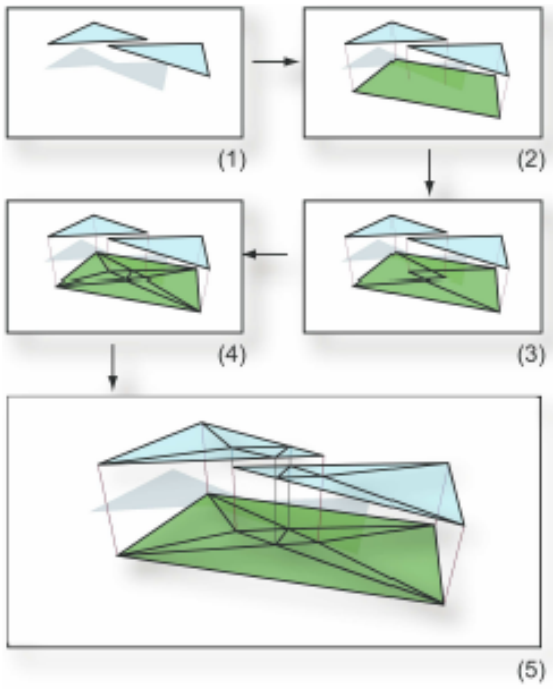

Figure 2.11: Different steps of the subdivision into prisms of the 3D model. (Lamarche, 2009 [46])

2.1.4.2 Arbitrary cell-shaped NavMeshes

An important drawback of techniques based exclusively on polygons of a determined number of sides (typically triangles and quads) is the over-segmentation obtained in the most scenes. The partition obtained is only optimal (or near-optimal) in very specific cases. Also, in the case of *navigation meshes* based on quads, they are not really extensible to general scenes, with obstacles randomly complex. To address these problems, convex-partitioning techniques based on N-gons (polygons of 3 or more sides) have been proposed.

Lerner et al. [50] presented a method to automatically generate a Cell-and-Portal Graph that worked both for interior and outdoor scenarios. The goal of their algorithm was to solve visibility problems, so the cells are not guaranteed to be convex. However, this algorithm could be easily adapted to create a *navigation mesh* using a post-processing step to convert the resulting cells into convex, for example, using the Hertel-Mehlhorn method [30] that is used to decompose a simple polygon without holes into convex regions.

In [89], a method to generate a convex partition for a virtual scene is presented. The geometry representing the terrain and the geometry representing the obstacles are treated separately. First of all, the process begins by looking at the raw geometry of the terrain. Typically, this data will be a huge list of triangles. The walkable surface is extracted by iterating over all of the polygons of the terrain and determining which ones has a slope low enough to be traversable by the character. Then, the Hertel-Mehlhorn algorithm [30] is applied to the resulting mesh to remove unessential diagonals, obtaining a partition of the walkable surface into convex N-gons.

Once the *NavMesh* of the terrain has been determined, the objects representing the obstacles are subtracted from the *NavMesh*. That is, for a given obstacle, find the set of cells of the initial *NavMesh* that intersects the obstacle and recursively subdivide them into smaller cells. To subdivide a cell, the center of the polygon is computed and an edge is created that joins the center of the polygon with the midpoint of every edge of the cell. Note that the resulting polygons are always four-sided. For each sub-cell generated from the initial cell, it is not further subdivided (if it is totally outside of the obstacle), it is discarded (if it is entirely inside of the obstacle) or it is subdivided again (if it is partially outside/inside the obstacle) until a maximum number of subdivision steps. Finally, a merging process is applied to eliminate redundant cells as much as possible.

A problem of the method proposed is that it needs to separate the geometry of the terrain from the geometry of the obstacles, instead of simply launching all the geometry of the scene and obtain the resulting *NavMesh*. However, the most important problem is that the subdivision method proposed does not adapt well to the contour of the obstacles, as can be seen in figure 2.12. In addition, many little sub-cells are generated that cannot be easily merged during the merging process, resulting on an over-segmented partition of the scene.

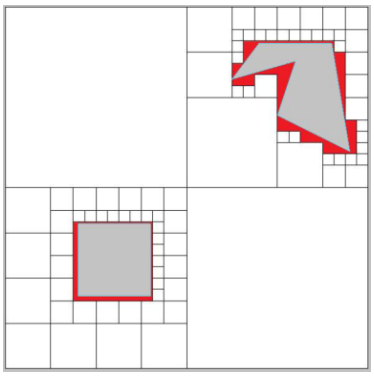

Figure 2.12: The subdivision process applied with two different obstacles. In red it is marked the walkable space that it is discarded due to the subdivision method does not adapts well to the shape of the obstacles. (Tozour, 2002 [89])

In [24], an automatic *NavMesh* generator method is described, that consists in spreading a certain number of unitary quad seeds on the scene. Those quads are expanded as much possible, adjusting to the contour of the obstacles even if they are not Axis-Aligned. Note that during the adjustment process, the cell generated can have more than 4 sides. When the algorithm ends, a merging process is applied to reduce the number of resulting cells. Although the authors say that it can handle complex obstacles, the fact is that the algorithm only gives a reasonably good partition with Axis-Aligned obstacles. In the rest of the cases, the algorithm proposed creates many narrow cells that in most cases cannot be removed during the merging process. In addition, the spreading method of the initial quads remains obscure. They do not give any notion of how many initial seeds have to be spread. Note that the resulting partition is completely dependent on the number of initial quads and its position. Another issue is that there can be intersection of portals which could be problematic when applying a local-movement method, leading to unnatural movement of the characters. The merging process helps to reduce the final number of cells, but the result is far from the optimal subdivision. In addition to these problems, the method only works if every obstacle is convex, so a previous step to decompose the obstacles into convex parts is required. A volumetric version of this algorithm was proposed in [23], but it has the same limitations than the 2D version.

Toll et al. [94] presented an automatic *NavMesh* generator for a multi-layered environment, such as an airport or a multi-story car-park, where the different layers of the scene are connected by elements such as stairs or ramps. Each layer is represented as a set of 2D polygons that lies in the same plane, and the medial axis set for the layer is computed. The connections between layers are used to iteratively merge the different sets of medial axis and create a single data structure. Then, they extend this structure by adding segments with the closest obstacle to create a convex partition of the scene. The main problem is

that a large number of unnecessary cells are created. This could be mitigated in part by reducing the noise on the computed medial axis. An approximation of the medial axis set can be computed using the GPU, as described in [31]. The implementation of this *NavMesh* generation method, restricted to one single layer, can be found in [17]. It requires to manually create a file that describes the contour of the obstacles, so the process is not fully automatic.

Unreal Engine [90] has also its own *NavMesh* generator. Firstly, a high-density grid that covers all the walkable area is automatically generated. Starting by a position placed by a designer, the map is "flood filled". That is, according to some seed size, each segment of the map is examined via raycasts and once verified, added to the grid. Figure 2.13 (left) shows the resulting grid. One disadvantage of this approach is that objects which are slightly out of phase with the seed size being used for exploration can end up being far away from the boundary of the mesh. To alleviate this, when an obstacle is hit the seed size will be subdivided N times to achieve the desired level of accuracy, as illustrated in figure 2.13 (right). Once the high-density grid has been obtained, a process to merge the squares is applied to reduce the number of polygons. Those polygons are then merged into concave slabs separated only by differences in slope and height. Note that this can lead to an over-segmented partition in irregular terrains. Finally, these concave slabs are decomposed into convex shapes. The main problem of the method proposed by Unreal Engine is that it creates many ill-conditioned cells that can introduce artifacts on the movement of the characters. In addition, the partition obtained on irregular terrains is over-segmented.

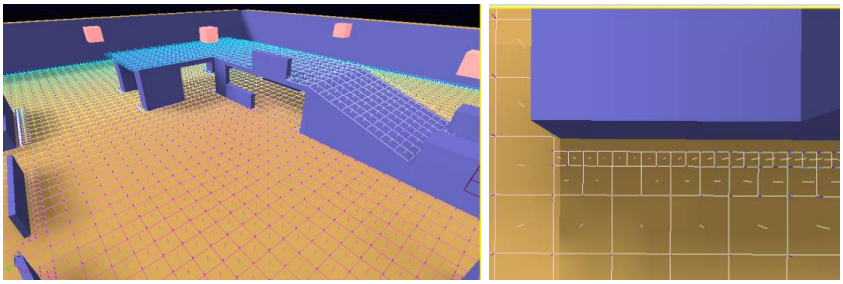

Figure 2.13: The resulting high density grid (Left). The cell size is adapted to fit with the shape of the obstacles (Right). (UDK, 2004 [90])

Recast [59] is an automatic open-source *NavMesh* generator broadly used in popular video games and other complex virtual applications. The method used by Recast is inspired by the work by Haumont et al. [27]. Recast computes a partition of the scene by applying the Watershed Transform (WST) [76] on the Distance Map Field [77] of the scene. Figure 2.14 illustrates this idea.

As an input, Recast takes an arbitrary geometry that is voxelized. This process makes the method robust against degeneracies of the model (such as interpenetrating geometry, cracks or holes) as well as simplifies the furniture

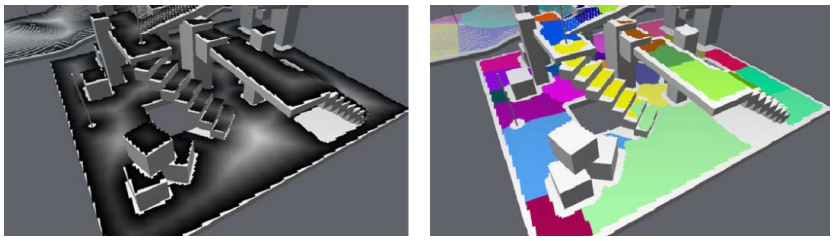

Figure 2.14: The Distance Map Field of a scene (left) and the resulting partition after applying Watershed (right). (Mononen, 2009 [59])

of the scene. The navigable space is built from the voxel model. A voxel is marked as navigable if it passes the following tests: First, the top of the voxel is at least a minimum distance from the bottom of the voxel above it, which means that the agent can stand on the voxel without colliding with an obstruction above. Second, the top of the voxel represents geometry with a slope low enough to be traversable by agents. Once the walkable space has been obtained, its distance map is constructed by estimating of how far each traversable voxel is from its nearest border voxel. A border voxel is a voxel that represents the boundary between the traversable surface and either obstructions (such as walls) or empty space. The distance map describes a topological surface and hence, the Watershed Transform can be applied to obtain a partition of the scene. The cells generated using the WST are not necessarily convex, but it ensures that does not contain holes, so it is easy to convert into convex those regions. What they do is to apply a modified version of the ear-clipping method to triangulate the cells and then, unessential diagonals are removed.

The strong point of Recast is that it can handle any kind of scene. It works on indoor and outdoor scenes and those scenes can contain multiple levels (such as a building). However, the main drawback is the over-segmentation produced, even in very simple scenes, as illustrated on figure 2.15. A better partition could be obtained by applying a post-process to try to merge adjacent cells into biggest cells, while maintaining the convexity condition. Another problem is that the adjustment of the *NavMesh* to the original virtual scene strongly depends on the voxel resolution being used.

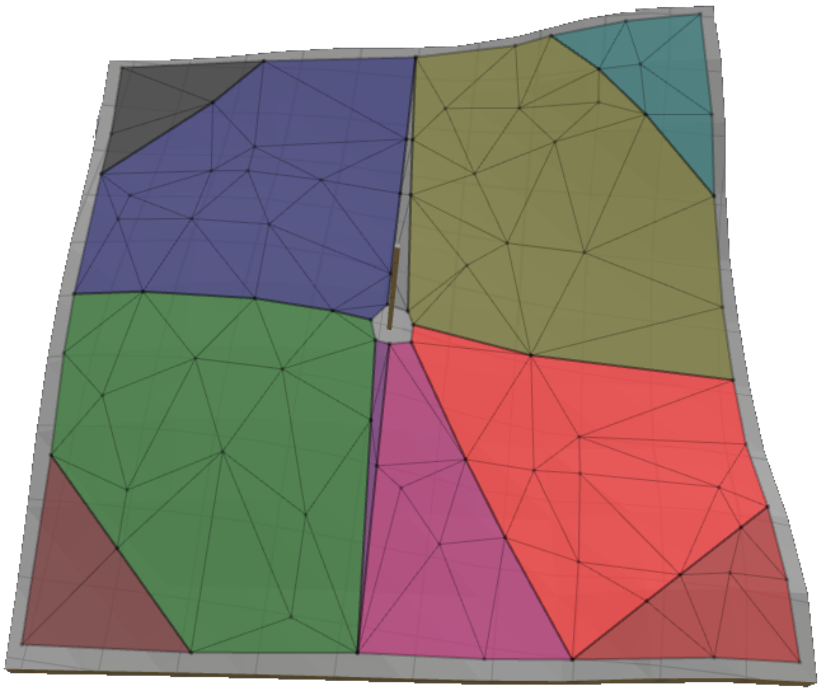

Figure 2.15: A partition (colored cells) obtained using Recast on a very simple scene in which however, suffers from over-segmentation. (Mononen, 2009 [59])

2.2 Local Movement

Local movement techniques aim to provide a mechanism for the autonomous characters to move from one location to the next in a path in a smooth and natural manner, while avoiding collisions with dynamic obstacles. These methods are generally driven by setting way points within the portals of the *NavMesh* that work as attractors to steer the agents in the right direction.

In [73] a dynamic method for simulating flocks is presented, contrary to the traditional technique that consists of scripting the individual behavior of each character conforming the flock. The main target is to simulate the aggregate behavior of groups of agents that can be seen in the nature such as a flock of birds, a herd of land animals or a school of fishes. The simulated flock is represented as a particle system, with the simulated characters being the particles. In this approach, each character is an independent actor that navigates according to its local perception of the dynamic environment, the physics laws present on the virtual world and a programmed set of behaviors that guides the characters through the environment, avoiding collisions static obstacles and the other agents in the flock. The aggregate motion of the simulated flock is the result of the dense interaction of the relatively simple behaviors of the individual simulated characters.

The resulting simulations built from this model seem to correspond to the observer's intuitive notion of "flock-like motion" (see figure 2.16). However, it is difficult to objectively measure how valid these simulations are. The user can tweak the different parameters of the simulation in order to improve and refine the model, as well as to achieve many variations of a flock-like behavior.

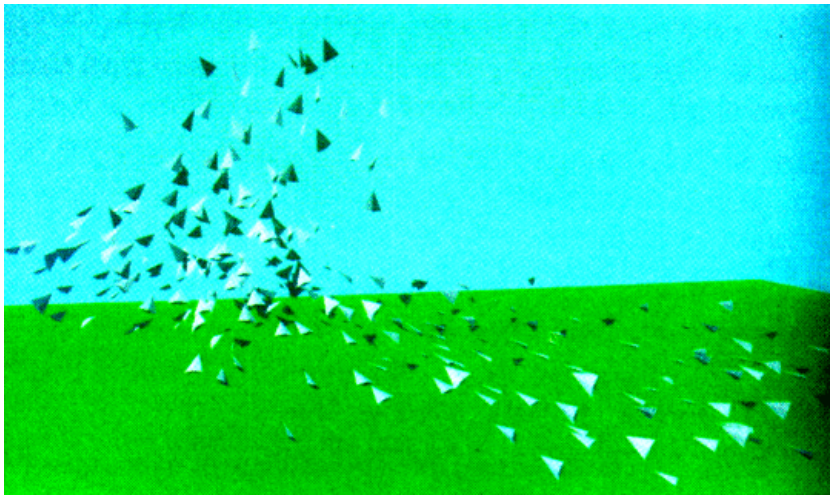

Figure 2.16: A simulated flock of bird-like agents. (Reynolds, 1987 [73])

Additionally, the behaviors presented in this work are simplistic and of low complexity, it basically consists of a set of rules to avoid collisions and to go to a desired point in the virtual world. However, a real character may have more complex motivations. For example, if we think in a real flock of animals, it would be also interesting to simulate more elaborated behaviors such as take into account hunger, finding food, fear of predators, a periodic need to sleep and so on.

The motion behaviors introduced in [73] are further explained, generalized and extended in [74]. These behaviors, formerly named *steering behaviors* are basically a set of simple algorithms for guiding autonomous characters in the virtual world in a life-like manner. More specifically, a *steering behavior* is described in terms of the geometric calculation of a vector representing a desired steering force. This work proposes behaviors such as seek or flee to a point in the scene (that can be static or another moving character), obstacle avoidance (restricted to sphere shaped obstacles), path following, agent grouping and many others (figure 2.17. The *steering behaviors* are independent of the mean the character is using for locomotion, i.e., it can be applied to a character walking on foot or driving a car indistinctly.

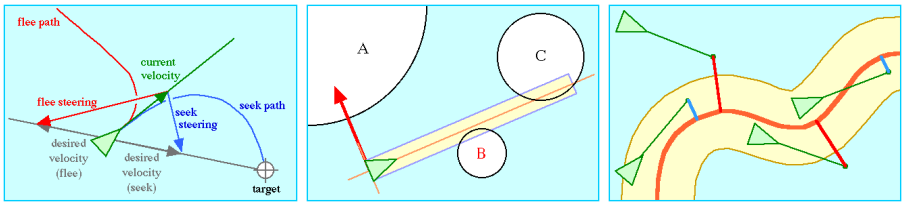

Figure 2.17: Examples of some of the *steering behaviors* proposed. From left to right, *seek and flee*, *collision avoidance* and *path following*. (Reynolds, 1999 [74])

The *steering behaviors* can also be combined or blended together in order to achieve higher level goals. For example, a character can get from A to B while avoiding obstacles, following a specific corridor or joining a group of other characters. The most straightforward way of blending is simply to compute each of the component *steering behaviors* and sum them together, usually with a weight factor for each component. However, the main drawback of this method is that some component behaviors may cancel each other out, as each *steering behavior* is basically a vector. Also, it is hard to find a proper combination for the weights in order to find the desired global behavior. These problems can be mitigated by blending the different component behaviors in smarter way, such as giving a certain priority to each *steering behavior*. For example, first priority is obstacle avoidance, second is seek to the target point, so if the steering force of the obstacle avoidance is non-zero (meaning a potential collision), this is used to steer the character; otherwise, it steers the character using the seek component and so on.

In [67], a system called *HiDAC* (for High-Density Autonomous Crowds) is presented. This software system focuses on the problem of simulating the local motion and global pathfinding behaviors of large and dense crowds moving in a natural manner. The main novelty of this work is that it introduces a set of psychological (e.g., impatience, panic) and physiological (e.g., locomotion, energy level) attributes, so each character may have its own personality, creating a truly heterogeneous crowd. Those psychological and physiological rules are combined with other typical geometrical and physical rules for collision avoidance and basic character steering, providing a high realism for the crowd being simulated (see figure 2.18). *HiDAC* can be tuned to simulate the behavior of different types of crowds, ranging from extreme panic situations (fire evacuation) to high-density crowds under calm conditions (leaving a cinema after a movie). It also can exhibit different behaviors simultaneously, allowing the simulation of heterogeneous crowds.

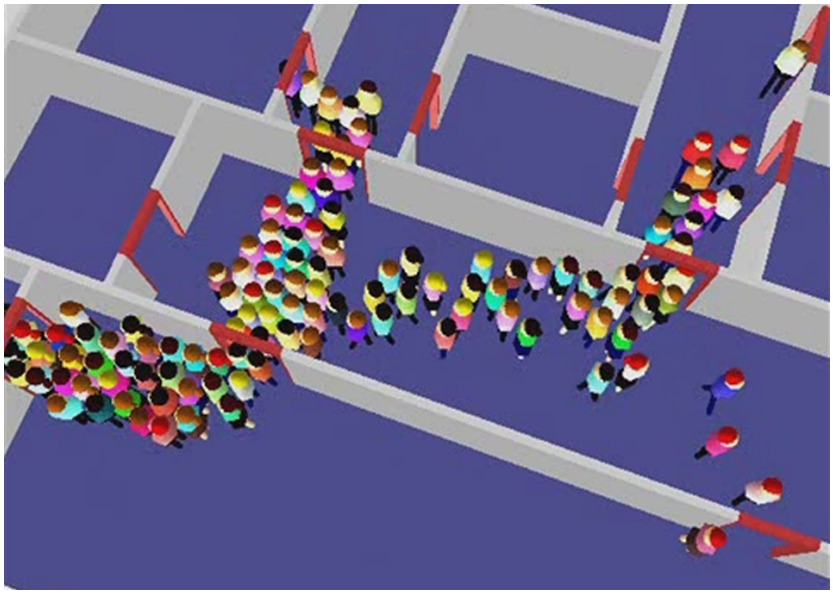

Figure 2.18: By combining psychological and locomotion rules, *HiDAC* improves the realism on the simulated crowds. In this example, red-haired characters are highly impatient, so they are able to look for alternative paths and avoid bottlenecks (pay especial attention to the characters at the bottom right corner). (Pelechano et al., 2007 [67])

However, the user have to understand some of the lower level methodologies of *HiDAC* and manually tweak the different parameters in order to obtain the desired crowd behavior. Additionally, agents should have a wider variety of actions other than locomotion, in order to provide an even more realistic crowd simulation.

One frequent problem when simulating crowded multi-agent systems is that agents tend to present unrealistic oscillating trajectories. Imagine the following situation. Two agents A and B are moving in a velocity (movement direction and speed) v_A and v_B respectively, such that the engine detects that they are going to collide in the immediate future. As a result, agent A alters its velocity to $v_{A'}$, so the new velocity avoids the collision with B. At the same time, agent B also alters its velocity to $v_{B'}$ in order to avoid the collision with A. The problem is that in the next frame, as agent A is moving with velocity $v_{A'}$ and agent B is moving with velocity $v_{B'}$ that guarantee collision avoidance, both agents could select again the previous velocities v_A and v_B, because for example these velocities leaded them directly to their respective goals. In the next cycle, these velocities will result in a collision, so they change their velocity again, and so on. Thus the agents oscillate between these two velocities, even if the agents initially choose the same side to pass each other.

The *Reciprocal Velocity Obstacle (RVO)* approach [92], [93] addresses this problem by taking into account the relative behavior of the other agents, assuming that the other agents will make a similar collision avoidance reasoning. Under this assumption, *RVO* guarantees oscillation free motions. There is no explicit communication with other agents, so each agent navigates in an independent form. Therefore, the problem can be reduced to navigating a single agent to its goal location while avoiding any obstacle and the other agents in the environment. The only information each agent is required to have about the other agents is their current position and velocity (movement direction and speed) and their shape, usually represented as discs in the 2D case or cylinders in the 3D case.

Recalling the previous example with agents A and B trying to avoid an imminent collision, the *RVO* of agent B to agent A *RVO(B, A)* is computed. The result is a cone-shaped obstacle that describes all the potential velocities $v_{A'}$ that would result in a collision with B. In order to avoid collision with agent B, agent A should choose a new velocity $v_{A'}$ that is outside *RVO(B, A)*, but assigning this velocity directly to the agent would result in the previous oscillation problem described before. So the average of the current velocity v_A (that already lies in *RVO(B, A)*) and $v_{A'}$ (a safe velocity for collision avoidance) is finally computed as the new velocity for agent A. Figure 2.19 shows an example of applying this approach with several characters.

The original description of *Reciprocal Velocity Obstacles* has some limitations, particularly that in dense crowds it frequently causes agents to enter in a *reciprocal dance*, an artifact that is produced when two agents cannot reach an agreement on which side to pass each other and they end up choosing a new velocity that does not resolves the collision. An extension called *Hybrid Reciprocal Velocity Obstacles* is presented in [79], aimed to solve this specific problem by enlarging the *RVO* on the side that the agent should not pass. Consequently, if agent A attempts to pass agent B on the wrong side, then agent A has to give full priority to agent B.

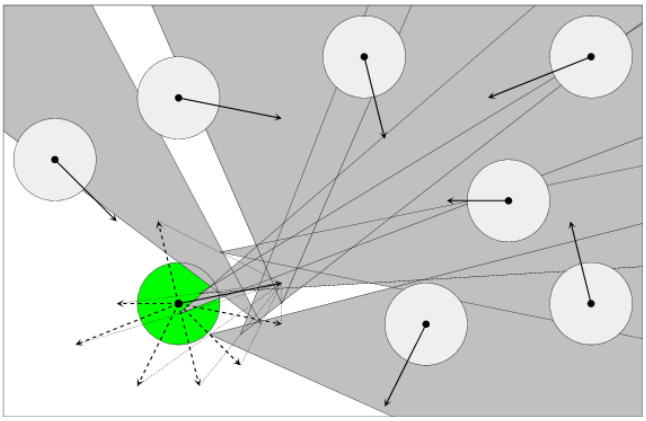

Figure 2.19: In dark-gray, the different *RVO* produced by the light-gray agents surrounding the green one. In this situation, the green agent selects the new velocity that is closest to its preferred velocity and does not intersects any of the *RVO*. (van den Berg et al., 2008 [93])

Another common problem to some local movement algorithms is that characters tend to line up as they share the same intermediate attractor point to steer the agents in the crowd. Some methods for achieving variety in characters' routes have been proposed. For example Pettré et al. [70] presented a solution for *roadmaps* based on having a denser sampling of nodes, which allows for a better use of the free space at the expense of longer computational time.

The obtained paths enable individual behaviorial diversity, while ensuring the achievement that the character reaches its goal. The solution proceeds in two stages. First a dense *roadmap* is built from the 3D definition of the environment and agents' bounding box. Then, given a specific start and goal position, a set of feasible paths is extracted from the *roadmap* using a customized iterative Diskstra's algorithm implementation that returns a set of paths ranging from the shortest to less optimal ones fulfilling some user criterion (number of paths found or relative length from the optimal path overpassed).

The main drawback of this approach is precisely that in order to produce a wide variety of different solution paths, the built *roadmap* needs to be dense and contain redundant connections between nodes, ideally covering the whole free space, which has a direct impact in the time performance when computing the path solution and makes this solution hard suitable for crowd simulation. Figure 2.20 shows the density of the *roadmap* generated for a simple scene.

Other approaches using skeletons [22] allow for larger or smaller distances to the skeleton depending on crowd density. The problem with this later approach is that characters are spread as the density increases, but when densities are low they all tend to follow the same trajectories.

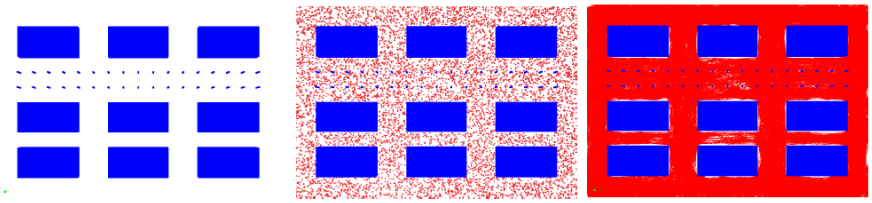

Figure 2.20: *Roadmap* construction stage. From left to right, the initial scene, the sample points in the free that conforms the nodes of the *roadmap*, and finally the edges of the *roadmap* covering practically the whole free space. (Pettré et al., 2005 [70])

Another usual way of constructing *roadmaps* is by computing the *straight skeleton* of the free space of the scene. In [22] a method for distributing pedestrians in a urban environment is described.

In [9] an improvement to traditional *way points* is introduced by extending the definition of an intermediate attractor point to a line segment called *way portal*, thus avoiding the problem of having agents lining up and allowing a better use of the available free space, as the whole portal is able to be used for steering agents. The *way portal* concept is general and it is independent of the specific local navigation algorithm being used, as well as it can be applied to any *Navigation Mesh* data structure, where the *way portals* are directly mapped to the portals of the *NavMesh*.

Given an start and goal position, the *way portal path* is defined as the set of portals that must be crossed from the start to the goal positions. Notice that this defines a space of paths, contrary to a common *way point path* that defines a single path. When an agent traverses the environment, one single path must be selected from the space. The strategy to select the point over the next portal p_i is as follows: The equation of the line between the agent's current position and the center of portal p_{i+k} is computed for a user defined value $k >= 0$. If the line intersects portal p_i, the steering point for the agent is set to the intersection point. Otherwise, it is set to the endpoint of the portal p_i that is closest to the previously defined line. Each time the agent crosses a portal, the same computation is performed again in order to compute the steering point over the next portal and so on, until the agent reaches the target position.

Notice that as the computation of the steering point takes into account the current position of the autonomous agent, the resulting paths are better distributed along all the free space in comparison to traditional *way points*, as can be seen in figure 2.21. However, the parameter k is completely dependent on the scene complexity and the underlying *NavMesh* quality, so this value must be tweaked in order to obtain a proper agent behavior. An automatic strategy to select k is proposed, which consists of selecting p_{i+k} as the next non-visible portal from the current agent's position, but it is more computationally expensive than having a predefined value. Additionally, for the last portal in

the path the steering point of all the characters will be the center point of the portal, so all the agents will tend to converge towards the center of the portal.

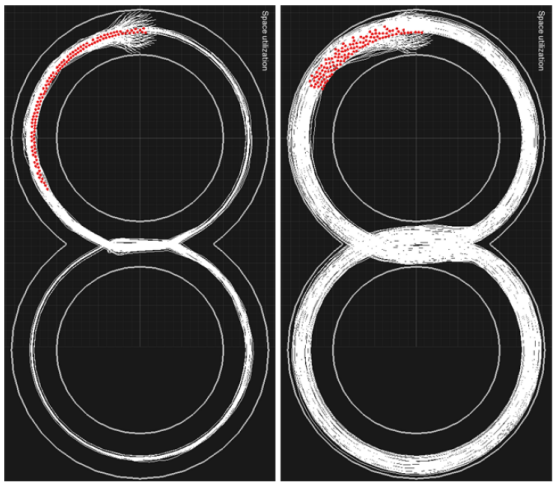

Figure 2.21: A comparison between the space used by traditional *way points* (left) and *way portals* (right). The space utilization in the case of *way portals* is improved as agents make a better use of the available free space. (Curtis et al., 2012 [9])

Finally, agents in a crowd are usually not point agents but they have a bounding radius and therefore, they should keep a minimal safe distance with respect to the static geometry to avoid having characters sliding along the walls of the virtual world. This distance is formerly known as *clearance*, and it is equal to the bounding radius of an agent. A common strategy to address this problem consists of enlarging the set of obstacles by a specific amount of *clearance*, known as the *Minkowski sum*. An example of an application using this method is Recast [59] (figure 2.22).

The main advantage of this approach is that characters will move respecting the desired amount of *clearance* calculated offline. Therefore this method does not have an impact on the performance of the algorithm being used. However, its major drawback is that it is bounded to a specific value of *clearance* that is defined by the size of the biggest agent in the simulation. Notice that characters with a smaller bounding radius won't be able to use the whole available free space, thus producing unnecessary bottlenecks and collisions in highly crowded environments.

Some *NavMeshes* data structures are created with the goal of solving the clearance problem, as the *Local Clearance Triangulation (LCT)* [37] described in Section 2.1.4. Although the *LCT* allows computing free paths with any desired value of *clearance*, one drawback is that this technique is not general, it only works with this specific data structure. Furthermore, the initial *CDT*

of the scene will be in the general case, a non-optimal convex partition, as any *NavMesh* based on a specific cell shape. Therefore, by introducing new portals in the *CDT*, the result is an over-segmented convex partition that will contain many unnecessary cells and this will have an impact to the global performance of the pathfinding algorithm.

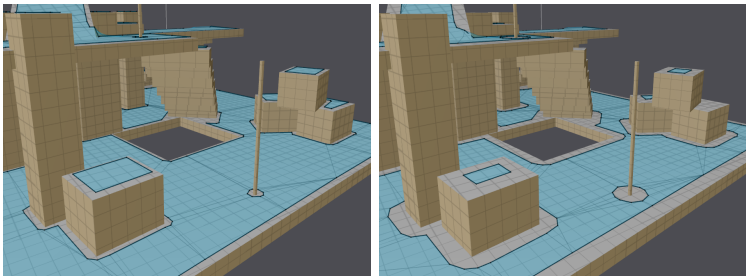

Figure 2.22: *Recast* is able to generate *NavMeshes* respecting a defined value of clearance by enlarging the obstacles using the *Minkowski sum*. This figure shows the same portion of the generated *NavMesh* for a *clearance* value of 0.3 (left) and 1.0 (right). (Mononen, 2009 [59])

2.3 Path Finding

As we have described in Section 2.1, the ultimate goal of any *global movement* algorithm is to end up with an abstraction of the virtual scene, which can usually be treated as a connected graph. Therefore, when a character needs to find a path from its current position to a goal position in the scene, this problem is reduced to search a valid path from the node containing the current position of the character and the node containing its desired goal position. In this section we review the most important *path finding* algorithms on graphs.

The $A*$ algorithm [26] uses a cost function to restrict the number of states that must be evaluated before finding the true optimal path between the given initial and goal nodes. As $A*$ traverses the graph, it builds up a tree of partial paths. The leaves of this tree are stored in a priority queue that orders the leaf nodes by a cost function, which combines a heuristic estimate of the cost to reach the goal and the cost from the initial node to the current one. Given a node n, the cost function is defined as:

$$f(n) = g(n) + h(n) \tag{2.1}$$

Where $g(n)$ is the known cost of the optimal path to go from the initial node to the current node n. This value is tracked by the algorithm; $h(n)$ is an heuristic that estimates the cost to go from the current node n to the goal node. The quality of the heuristic greatly influences the efficiency of the search. In the specific case of path finding in a CPG, a typical heuristic consists of the euclidean distance between the current node n and the goal node. Notice that in order to find the optimal path, $h(n)$ must be *admissible*. That is, $h(n)$ is necessarily less or equal than the real cost from the node n to the goal node.

The $A*$ algorithm guarantees that the path found (if it exists) is the minimal possible one between the initial and the goal nodes and due to its simplicity on implementation and performance, $A*$ is widely used in the field of path finding in real-time applications, and it is the base of a large amount of subsequent work. However, $A*$ algorithm can be very time consuming for large scenarios, with high memory requirements.

In the most challenging real-time applications, the time to obtain a solution is critical and sometimes there is no need (nor time) to find the optimal solution, but a suboptimal one is enough. In that context, *Anytime planers* are well suited for these problems. On this general approach, a suboptimal solution is quickly given and then it is continually improved until a given limit time runs out. *Anytime Repairing A* (ARA*)* [53] is one of the most popular anytime heuristic search algorithm.

Contrary to the common $A*$ implementation, the $ARA*$ algorithm uses an *inflated* heuristic. That is, using $\epsilon * h(s)$ for $\epsilon > 1$. Inflating the heuristic often results in much fewer state expansions and consequently, the search is faster.

However, this may violate the *admissibility* property of the heuristic, so the result is no longer guaranteed to be optimal. *ARA** starts with a given ϵ and it iteratively improves the solution by reducing ϵ and reusing previous search efforts to accelerate subsequent searches. The solution is refined until the given limit time is reached or when ϵ is equal to 1, resulting to the standard *A** and therefore, retrieving the optimal solution. Although *ARA** does not guarantee optimality, the sub-optimal solution is bounded by the factor ϵ, so the found solution is no longer than ϵ times the length of the optimal solution.

In some applications like planetary exploration or military reconnaissance, the virtual environment is not known beforehand but it is constructed whilst is being discovered. In those situations, an autonomous robot equipped with a number of sensors (figure 2.23) has the mission of finding the path between two points in the space without any previous knowledge of the environment, nor where the obstacles are. The *Dynamic A* (D*)* [81] tackles this specific problem. The autonomous robot sweeps the terrain for obstacles, records its progress through the environment and builds a map of sensed areas containing the location of the obstacles. With each addition to the map, the *D** algorithm is able to optimally replan the global path and recommend steering commands to reach the goal.

Figure 2.23: The *Navigational Laboratory II (NAVLAB II)* was the robot used for testing the capabilities of the proposed algorithm. It successfully found a path between the initial position and a goal position set in an unknown area of 500 x 500 meters. (Stentz et al., 1995 [81])

The *D** finds the optimal path between the current position of the robot and the goal using the partial information of the environment that the robot has acquired so far. Then, the robot moves along the path until either it reaches the goal or the sensors detect an obstacle, updating the map accordingly and

replanning a new optimal path from the robot's current position to the goal. The path replanning is done efficiently, allowing the algorithm to run in real-time. The *D* Lite* [45] approach is a revision of the original *D**, offering a new implementation at least as efficient as *D** but substantially shorter.

The *Anytime Dynamic A* (AD*)* [52] is an anytime replaning algorithm that combines previous algorithms described before. It efficiently generates solutions for complex and dynamic environments. *ARA** is used in order to find a suboptimal solution in the specified limit time. When changes in the environment are detected, *D** is applied in order to repair previous solution incrementally, without the need of recomputing the solution from the scratch. Therefore, *AD** is highly suitable for those applications with limited or not knowledge at all about the environment, where the computation time is more important than finding the optimal path and a suboptimal path is enough.

Most of the work on path finding focuses on planning the path for individual agents through the virtual environment. However, sometimes agents join together to form groups and from the point of view of the path planner algorithm, the whole group should be treated as a single agent. Huang et al. [88] addresses the problem of group path planning while maintaining group coherence and persistence as much as possible. The group of agents is modeled as a deformable and splittable shape. The *coherence* property of a group of agents tries to minimize the dispersion and this is achieved by introducing a deformation penalty to the cost computation. When the deformation penalty reaches a threshold value, the group may split in two or more subgroups that are merged together again later on. The *persistence* property is modeled by introducing split and merge actions of the group, and penalizing the split action in the cost computation. Therefore, the computed paths to given goals tries to minimize the three-tuple cost vector of distance, deformation and splitting.

Although paths can be computed efficiently in reasonably complex environments, the introduced possibility of splitting groups can considerably increase the computational cost of the path finding algorithm as the search space increases because we need to compute a valid path for each subgroup generated. This can be mitigated by increasing the deformation threshold to split a group, so limiting in some way the maximum number of possible splits in order to meet performance constraints.

Hierarchical graph representations [78], have been also introduced in order to improve the efficiency of the desired path finding algorithm by representing the search space as a hierarchy. In particular, it is possible to create a simpler search graph by grouping a number of nodes of the original search graph into a single node in an abstract search graph. Abstract edges are then added based on the edges that exist in the original search graph. This process can be done recursively, giving as a result a hierarchy of abstractions. An approximate solution can be found by a search in the abstract graph, which can then be refined to a solution in the original search space. This way, time requirements are better affordable even in really complex domains. Notice that a *hierarchical graph*

is an abstract representation that can be applied regardless the path planning algorithm being used. For example, Holte et al. applied the *hierarchical graph* representation to the popular *A** algorithm [33].

In [82], Sturtevant describes the path finding strategy used in the *Dragon Age Origins* video game. The algorithm is restricted to grid-based maps and it abstracts the grid into sectors and regions. A sector is a 16 x 16 grid of cells. A region is a set of cells in a sector that are mutually reachable without leaving the sector. A sector may have multiple regions, and a region is always in only one sector. Figure 2.24 exemplifies the subdivision of a grid-based map into 4 sectors which results in a graph with 9 regions and 10 edges.

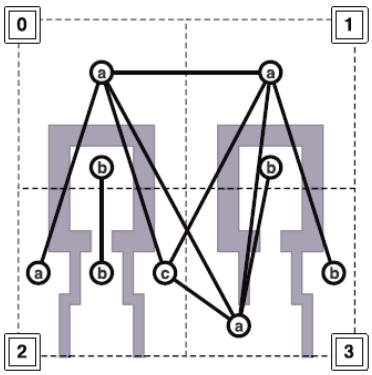

Figure 2.24: Abstract graph created with a 16 x 16 overlay on a grid-based map. Gray cells represent obstacles. (Sturtevant, 2008 [82])

The previous described approach is further improved in the work presented by Sturtevant and Geisberger [84] in order to handle a number of situations. The main problem they faced with the first approach is that on the largest maps, path finding requests could run out of time when planning in the abstractions, while on smaller maps such as indoor virtual scenarios, the abstraction was not adequately representing the underlying terrain. In order to solve this problem, they added an additional layer of abstraction, built on top of the previous described abstraction. Figure 2.25 illustrates this step. Sector 0 in this figure is the same as 2.24 but is now contained into a single high-level sector. This extra abstraction layer was built using a sector size twice the size of the original sector abstraction. Depending on the size of the original map, the final game used sectors of size 8 x 8 to 16 x 16 for the first abstraction layer and 16 x 16 to 32 x 32 sector size for the additional abstraction layer.

The *DBA** approach [49] is another grid-based path finding algorithm that uses a database of pre-computed paths to reduce the time to solve search problems. *DBA** starts by performing an offline pre-computation before online path finding. The offline stage abstracts the grid into sectors of size n x n and regions just as previously described in [82]. The algorithm proceeds by constructing a database of optimal paths between adjacent regions using *A**. Each path found

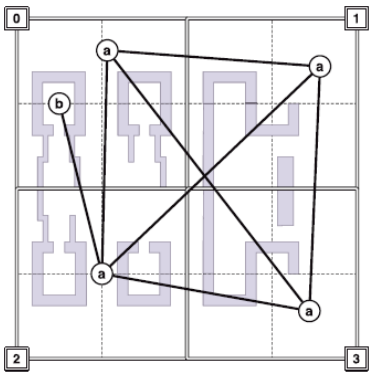

Figure 2.25: Additional level of abstraction. (Sturtevant et al., 2010 [84])

is stored in a compressed format as proposed in [47].

In the work of Sturtevant and Jansen [85], several abstraction methodologies for graphs are evaluated in a testbed formed by different 2D grid-based maps. The *clique abstraction (CA)* was introduced by Sturtevant and Buro [83] and it basically consists of grouping connected nodes into cliques at each level of abstraction. The maximum clique size allowed is 4-connected nodes. Figure 2.26 illustrates the whole process of *cliques abstraction* over a simple graph. The left figure (a) shows the initial graph. The dotted lines indicate two cliques sample. The middle figure (b) is one possible result the previous graph can be abstracted. The same abstraction mechanism can be applied more than once in (b) to obtain the graph in (c).

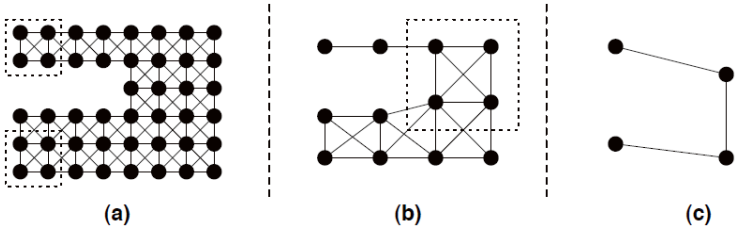

Figure 2.26: The clique abstraction process. (Sturtevant et al., 2010 [85])

The *sector abstraction (SA)* technique was suggested by Botea et al. [5] and it is limited to grid-based maps. This abstraction is parametrized by a fixed sector size k. At the first level of abstraction, sectors of size $k \times k$ are overlaid onto the grid-based map. At the i^{th} level of abstraction, sectors of size $k^i \times k^i$ are used. Figure 2.27 exemplifies this abstraction process with a sector size of 2. In graphs depicted at (a) and (b), the dotted lines determine two of the sectors (4x4) used for constructing the final graph depicted at (c). As only nodes which form a connected component within a single sector can be abstracted together,

the top left sector is split into two separated nodes when abstracted. This results in one extra node in the most abstract graph on the right, (c). Note that if we apply an extra level of abstraction, the entire graph would be abstracted into a single node, because all the nodes in graph (c) are connected within an 8x8 sector.

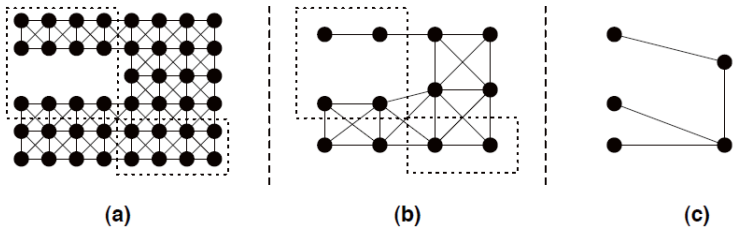

Figure 2.27: The sector abstraction process. (Sturtevant et al., 2010 [85])

The *radius abstraction (RA)*, originally named *STAR abstraction* was introduced by Holte et al. [32]. This abstractions mechanism works by first selecting a reference node. All the neighboring nodes to this node within a fixed radius r are abstracted into the same abstract node. Figure 2.28 illustrates this abstraction process for a radius $r = 1$. The reference nodes are marked in gray. In (a), all the immediate neighbors $(r = 1)$ connected to the selected nodes are abstracted into the same abstract node, conforming the graph depicted in (b). This process is repeated again, giving as a result the graph in (c). An additional abstraction step would completely abstract the graph into a single abstract node. Notice that the result completely depends on the radius parameter as well as the methodology used to select the reference nodes.

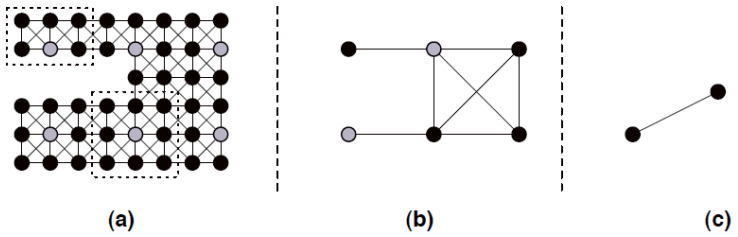

Figure 2.28: The radius abstraction process with $r = 1$. (Sturtevant et al., 2010 [85])

The *line abstraction (LA)* technique consists of finding sequences of nodes length k, and abstracts them together. The strategy used to select the sequences of nodes as well as the length of the sequences, determines the final result. In the example depicted in figure 2.29, the strategy used consists of first abstracting horizontally and then vertically for a sequence of nodes length $k = 2$. So taking as an input the graph represented in (a) each node is attempted to be abstracted with its neighbor to the right, resulting in the graph on (b). Next, each node

in (b) is attempted to be abstracted with its neighbor below, resulting in the graph on (c).

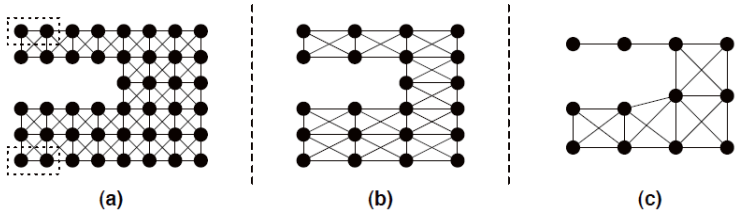

Figure 2.29: The line abstraction process with $k = 2$. First abstract horizontally, then vertically. (Sturtevant et al., 2010 [85])

In [16] a GPU-based technique is presented for multi-agent path planning in extremely large, complex and dynamic environments. The method consists of an adaptive subdivision of the environment with efficient indexing, update and neighbor finding operations on the GPU. The proposed method works as follows. On the CPU, the virtual scene is subdivided using a hierarchical quad tree representation, which is ported onto the GPU to compute an initial plan. Changes on the virtual environment are monitored on the CPU which triggers local repairs in the quad tree. The GPU representation of the quad tree is updated in order to reflect those changes, and the plan is efficiently repaired by updating only the costs which have been invalidated as a result of the change. Figure 2.30 illustrates this approach on a simple map.

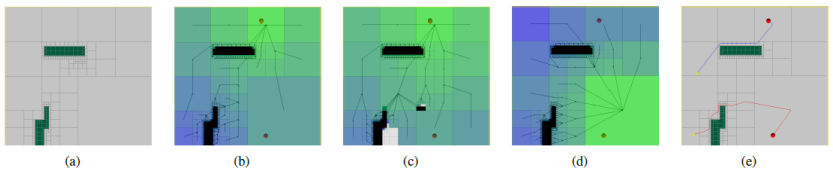

Figure 2.30: Method overview. (a) shows the initial subdivision of the environment into quads. (b) shows the plan computed for the top goal (red circle) for any quad. The colors represents each quad's g-value of the cost equation relative to a scale where green = 0, blue = max g-value and white = not computed. In (c), an obstacle is introduced with respect to the the previous image and the corresponding quads are locally repaired. In (d) the plan for the bottom goal is depicted. Finally, (e) shows the computed plan for all the autonomous agents. (García et al., 2014 [16])

The proposed adaptive representation of the environment reduces the memory requirements by an order of magnitude compared to other GPU based approaches, which enables path planning environments larger than $2048m^2$ for hundreds of autonomous agents with different goal targets. Additionally, the computational speed is up to 1000X faster compared to previous work.

Path finding techniques usually assume that all agents have the same navigational capabilities, which means that if a region of the terrain is not traversable by one agent, it is not traversable by anyone. Another common assumption consists of considering that all agents have the same size. Such assumptions limit the applicability of these techniques to homogeneous autonomous agents in homogeneous environments. The aforementioned problems are addressed in the work of Harabor and Botea [25] by introducing a new clearance-based hierarchical planner for grid-based maps, named *Annotated Hierarchical A* (AHA*)*. In this approach, a clearance value is associated to each cell of the graph for each traversal capability. In order to better illustrate this, let us consider the simple example depicted in figure 2.31 featuring two terrain types: Ground (white tiles) and Trees (grey tiles). Hard obstacles are colored in black. In this example, the set of capabilities C required to traverse the map is defined as $C = \{\{Ground\}, \{Trees\}, \{Ground \vee Trees\}\}$.

Figure 2.31 (a) to (d) illustrates the iterative process to compute the clearance value for a specific traversable tile. The clearance value is initialized to 1 and subsequent iterations (figures 2.31 (b) and (c)) uniformly extend the square and increment the clearance value until the square contains an obstacle (2.31 (d)) or it extends beyond the map boundary. Figures 2.31 (e) to (g) shows the clearance values associated to $\{Ground\}$, $\{Trees\}$, and $\{Ground \vee Trees\}$ traversal capabilities respectively. Additionally to the clearance annotation, a cluster-based hierarchical abstraction is used in order to build a highly compact representation of the original virtual environment.

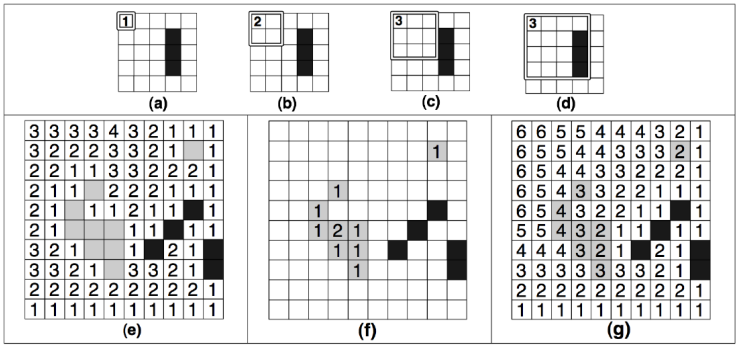

Figure 2.31: (a)-(d) Computing clearance. (e)-(g) Clearance values for different capabilities. (Harabor et al., 2008 [25])

Jorgensen presented an automatic structuring method based on a hierarchy that separated buildings into floors linked by stairs and represents floors as rooms linked by doorsteps [34]. This method has a strict hierarchy and does not scale to large outdoors environments such as the ones often presented in video games. Zlatanova [99] presented a framework of space subdivision exclusively for indoor navigation, by identifying rooms and corridors and including semantical information.

There are other approaches that focus on allowing agents to be more environment aware [42]. In this work, planning is based on an *Anytime Dynamic A**, and it is carried out satisfying multiple special constraints imposed on the path, such as: Stay behind a building, walk along walls or avoid the line of sight of other agents. In [41] a multi-domain anytime dynamic planning framework is presented which can efficiently work across multiple domains, by using plans in one domain to accelerate and focus searches in more complex domains. It explores different domain relationships including the use of way points and tunnels. The different domains use only two representations in terms of spacial subdivision, a 2D grid, and a triangular mesh.

Hierarchical representations have been used to calculate agents moving between two points at different levels of complexity [95], from finding a route to animating the 3D characters.

2.4 Discussion

As mentioned in the previous chapter, our main objective is to solve the problem of autonomous character navigation in complex virtual environments, which is a central problem in the fields of robotics, video games and crowd simulation. The limitations of the previous exposed methods that address this problem, have pushed us to develop a new approach that introduces improvements in both the *global movement* and the *local movement*.

Regarding to regular *grid* based partitions, an important drawback is that autonomous agents can only move to an adjacent free cell. This checkerboard approach offers realistic results only on lower density crowds, but looks unrealistic when trying to simulate high-density crowds. Additionally, the resulting connectivity graph used for path finding will be inevitably oversized and the adjustment to the contour of the obstacles greatly depends on the grid resolution, making it hard to fit general environments containing obstacles with an arbitrary shape.

The main limitation of the *roadmap* technique is that it only contains information about which points of the space are directly connected, but does not provide a proper description of the scene nor where the obstacles are. Therefore, calculating collision against dynamic obstacles is usually a hard task, not guaranteeing that it will always be possible to avoid collisions, and often characters get stuck stuck in local minima of the geometry.

In the case of *corridors*, the main limitation is that it is hard (and usually not possible at all) to cover the whole free space of a scene by using a finite number of 2D discs. Additionally, although this representation can give satisfactory results in 2D environments, there is no clear way of how to apply this approach on real 3D scenes. If we think in a multi-layered environment such as a building, problems may arise at the regions where two different layers meet. Moreover, a 2D disc cannot correctly represent the topology of the terrain, so it only works in perfectly flat floors.

The cell decomposition technique (i.e. *navigation mesh*) fits better with our requirements as it provides a more accurate description of the free space of a virtual scene, but the methods studied suffers mainly from over-segmentation, ill-conditioned cells or poor adjustments to the contour of the obstacles missing some portions of the walkable space.

As for the *local movement* there are mainly two problems that need to be addressed. On one hand, some current work sets a fixed steering point over the portal shared by all the autonomous agents. This increases the collisions against other agents, creates bottlenecks and agents tend to line up towards the steering point, so the available free space is poorly used. On the other hand, the *clearance* problem is usually limited to a single *clearance* value or bounded to a specific *NavMesh* data type.

Finally, when a *NavMesh* is used in order to create the connectivity graph for the *path finding* stage, the graph generated could contain cycles i.e., a character needs to pass more than once through the same cell, but this case is not supported by the *path finding* algorithms. Although some special *NavMeshes* such as the *Local Clearance Triangulation* [38] already guarantee that the generated graph does not contain cycles by construction, little research has been done on this specific problem because many of the *path finding* algorithms studied are restricted to grid-based maps, where the problem of having cycles does not appear. Therefore, there is a need for a general solution to handle this specific problem correctly.

In this thesis, we will present a novel and complete framework for autonomous characters that solves some of the limitations of the current state of the art. Firstly, we will introduce *NEOGEN*, our automatic *global movement* technique based on *navigation meshes* that produces a near-optimal convex partition of the scene, eliminates almost all the ill-conditioned cells and perfectly fits with the original geometry. Secondly, we will introduce *ExACT*, our novel *local movement* technique based on rules that allows computing paths free of obstacles with any desired amount of clearance for any *NavMesh* data structure, and dynamically computes the steering point of the agent based on its position with respect to the portal, so all agents have a different steering point, considerably improving the free space usage and reducing collisions against both the static and the dynamic geometry. Finally, we propose a new encode of the connectivity graph in order to avoid the problem of having cycles when the underlying data structure used is a *navigation mesh*. Our approach is general and can be applied to any type of *NavMesh*.

Chapter 3

Computing Navmeshes for 2D simple polygons

In this chapter we present our main contributions to the generation of *NavMeshes* in 2D environments. First, we introduce a novel automatic method for computing *NavMeshes* for a given environment represented as a 2D simple polygon (i.e., no self intersections) that can contain holes. The area delimited by the outer polygon can be seen as the floor plan of the scene, whilst the holes represent the static obstacles. Our algorithm, entitled *NEOGEN-2D*, takes this input and splits it into a near-optimal subdivision of convex polygons that are highly suited to path finding by avoiding the presence of both degenerated polygons and almost all ill-conditioned polygons. The cells generated are not restricted to a particular shape but they can contain an arbitrary number of sides. We refer to our subdivision as being near-optimal, because we can guarantee a number of partitions within the bounds of optimality. Moreover as we will proof in the results section, our methods leans towards the first half of the optimality bound.

Our second contribution is the introduction of the *convexity relaxation* concept, that exploits the fact that, depending on the characteristics of the underlying *local movement* algorithm being used to steer the movement of the autonomous characters through the environment, we can allow cells with certain small concavities. This allows us to further reduce the number of cells, which has a direct impact in the performance of the *path finding* stage.

To summarize, the overall goals of our navigation mesh generator are:

1. To achieve as few cells as possible
2. To achieve portals as short as possible (since it introduces less inaccuracies when setting attractors to drive the natural movement of the agents).
3. To avoid cells with interior angles close to zero, since it complicates the

local movements and leads to agents being physically in more than two cells simultaneously.

3.1 Overview

The problem of subdividing a polygon into convex regions has many application areas such as computational geometry, robotics and graphics. An optimal solution to this problem can be found in polynomial time for the case of simple polygons without holes [44] and code is available in the CGAL library [29]. However when the initial polygon contains holes, finding the optimal decomposition into convex polygons allowing Steiner points becomes NP-hard [44].

There are two possibilities when subdividing a polygon into convex cells. The first one consists of subdividing by adding diagonals, which are edges between pairs of vertices in the original geometry. The second one consists of using segments which are edges between a vertex of the geometry and a new point that is created on the boundary of the original geometry (also known in the literature as boundary Steiner points). The algorithm presented in this chapter carries out a partition based on segments, and thus we are not limited by the position of the vertices in the original geometry. Our motivation to follow such approach is that the resulting portals lead to better way-points to drive steering algorithms (short portals and minimum ill-conditioned cells, i.e. cells with interior angles close to zero)

As *NavMeshes* are usually constructed in a pre-processing stage, we are not concerned about the time complexity of our algorithm, and given that it only deals with static geometry, no further changes need to be made at run time. Dynamic obstacles and other agents are avoided through local movement techniques based on Reynolds' steering behaviors [74]. However, some game companies prefer to do these computations online while a game is loaded or when a level needs to be dynamically updated. Therefore although not necessary, it may be desirable to keep computational times as low as possible.

Once the subdivision is created, we automatically generate the Cell-and-Portal Graph (CPG) representing the environment, where cells are the convex polygons resulting from the subdivision, and portals are the segments created to subdivide the original polygon into convex cells.

We finally present an example of a multiplayer game where path finding is carried out through $A*$ over the generated *NavMesh* and movement within cells and dynamic obstacle avoidance are performed through steering behaviors. The physics library *Bullet* [8] has been integrated for several purposes including: speed up of the local movement simulation, guarantee non overlapping between agents, and keeping track of agents' within each cell to quickly update their mental maps in cases where agents are accidentally pushed through portals. Section 3.5 shows results of *NEOGEN-2D* as well as multi-agent navigation in

a game application.

3.2 Algorithm Description

The approach followed by our algorithm consists of subdividing the input polygon by first detecting which are the notches (concave vertices, i.e. interior angle larger than π) that appear in the polygon and then splitting them by creating portals so that for each original notch in the geometry, we will split it into two new angles that are both convex (i.e. interior angle smaller than π). In this way, we guarantee that if all the notches in the original polygon are split into convex angles we will obtain a partition consisting only of convex cells.

To guarantee that only one portal will be necessary to split a notch into two convex vertices, we introduce the definition of area of interest:

Definition 3.2.1. The area of interest, I_i of a notch v_i is given by two edges of the geometry, $e_{i-1,i}$ and $e_{i,i+1}$ as the resulting interior area of prolonging $e_{i-1,i}$ and $e_{i,i+1}$ as we indicate in figure 3.1 (left), where $e_{i-1,i}$ is the edge that joins v_{i-1} with v_i.

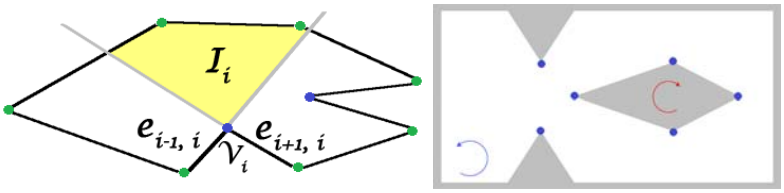

Figure 3.1: On the left, we show the area of interest, I_i of a notch v_i. Green vertices are convex, and blue vertices are notches that need to be split (left). On the right, we show a simple example of an input given to the algorithm, with the order of the vertices implying polygon (*in white*) or holes (*in grey*).

The main advantage of creating a portal within this area of interest I_i is that it guarantees that only one portal is needed to break the notch into two convex vertices.

Given a notch with interior angle $2\pi - \beta > \pi$, we need to create a portal that will split the notch into two new vertices with interior angles α and γ. The portal needs to be created within the area of interest so that it satisfy that $\alpha \in [\pi - \beta, \pi]$.

Proof. Let first proof that if we create the portal outside that area, we will end up with still one notch. If $\alpha > \pi$, by definition of a notch, the vertex with

internal angle being α would be a notch. Let evaluate the case where $\alpha < \pi - \beta$. Since $2\pi = \beta + \alpha + \gamma$ then $2\pi - \beta - \gamma = \alpha$ which turns into $2\pi - \beta - \gamma < \pi - \beta$ and so $\pi < \gamma$ which means that the vertex with internal angle γ would be a notch. □

Proof. Now we will proof that any portal created within the range $\alpha \in [\pi - \beta, \pi]$. guarantees that both α and γ will be convex. Similarly as before, if $\alpha < \pi$ by definition the vertex with interior angle α will be convex and since α is the upper bound of the range, then every value within the range will give a convex node. So to guarantee that we only need one portal we need to show that if $\alpha > \pi - \beta$ then the vertex with interior angel γ will also be convex. As in the previous proof we know that $2\pi = \beta + \alpha\gamma$, so if $\alpha > \pi - \beta$ then we have $2\pi - \beta - \gamma = \alpha$ which turns into $2\pi - \beta - \gamma > \pi - \beta$ and so finally $\pi > \gamma$. Therefore this range of values guarantee that both α and γ will be smaller than π □

The floor plan of the virtual environment where we want our characters to navigate is given as a simple polygon 2D, and the vertices are given in counter-clockwise order. Any obstacle within the virtual environment will be given as a polygon with its vertices in clockwise order. Obstacles can be seen as holes in the main polygon that represents the entire map (figure 3.1, right).

The input geometry consists of a polygon P enclosing other polygons H_1, ..., H_m, where all holes are simple empty polygons. Let δP be the boundary of the polygon P, and δH_i the boundary of the hole δH_i. We assume that the following conditions apply:

$$\begin{array}{ll} 1) & \delta P \bigcap \delta H_i = \emptyset, \quad \forall i = 1, ..., h \\ 2) & H_i \bigcap H_j = \emptyset, \quad \forall i \neq j \end{array}$$

The first step of the algorithm consists of determining which vertices are notches. This step is performed through an orientation test based on calculating the signed area of the triangle defined by three consecutive vertices, v_i, v_{i+1}, v_{i+2}:

$$A\left(v_i, v_{i+1}, v_{i+2}\right) = \frac{1}{2} \begin{vmatrix} \overline{v_i v_{i+1}}_{,x} & \overline{v_{i+1} v_{i+2}}_{,x} \\ \overline{v_i v_{i+1}}_{,y} & \overline{v_{i+1} v_{i+2}}_{,y} \end{vmatrix}$$

If the area $A(v_i, v_{i+1}, v_{i+2})$ is positive, it means that vertex v_{i+2} is on the left hand side of edge $e_{i,i+1}$ given by the previous vertices v_i and v_{i+1}. If it is negative, it means that v_{i+2} is on the right hand side of edge $e_{i,i+1}$. So for the main polygon which is given in counter-clockwise order, if the area is negative it means that v_{i+1} is a notch and thus needs to be split, whereas for the holes, given in clockwise order, we will also find a notch when the area is negative. We will introduce all notches of the geometry in a vertex list V to be treated

in order. This step has cost $O(n)$ where n is the total number of vertices of the geometry.

3.2.1 Creating portals

For each notch v_i in V, the algorithm looks for the closest element in the geometry that falls within its area of interest I_i to create a portal with it. This has cost $O(n \cdot r)$, where n=*number of vertices*, and r=*number of notches*. The closest element to a notch can be another vertex, an edge of the original geometry or a previously created portal. Depending on the element being selected, we classify three types of portals: vertex-vertex, vertex-edge, vertex-portal. Each of these cases needs to be treated differently. In any case, a portal is represented as two oriented edges with the same endpoints but in the opposite order one from the other. Algorithm 1 describes the way we create the portals, according to the closest element to the notch.

Algorithm 1 Portal creation algorithm

```
1: procedure PORTALCREATION
2:     for all v_i ∈ V do
3:         c ← computeClosestElementInAOI(v_i)
4:         if c is vertex then
5:             createPortalVertexVertex(v_i, c)
6:         else if c is edge then
7:             createPortalVertexEdge(v_i, c)
8:         else
9:             createPortalVertexPortal(v_i, c)
```

3.2.1.1 Case 1: Vertex-Vertex portals

When the closest element to v_i is another vertex v_j of the geometry, the algorithm simply needs to create a portal p_i between v_i and v_j. As can be seen in figure 3.2, the portal created guarantees that v_i gets split into two convex regions, and thus no further processing of v_i is necessary to subdivide the original polygon into convex cells. If the other vertex v_j was also contained in V (which means that it is also a notch), then the algorithm also checks whether by creating portal p_i, v_j gets split into two convex angles. This will happen exclusively when v_i falls within I_j as we can see in the example shown in figure 3.2.

3.2.1.2 Case 2: Vertex-Edge portals

When the closest element to v_i is an edge $e_{j,j+1}$ of the geometry, the algorithm needs to create a portal p_i between v_i and a point q in the segment $e_{j,j+1}$. Since

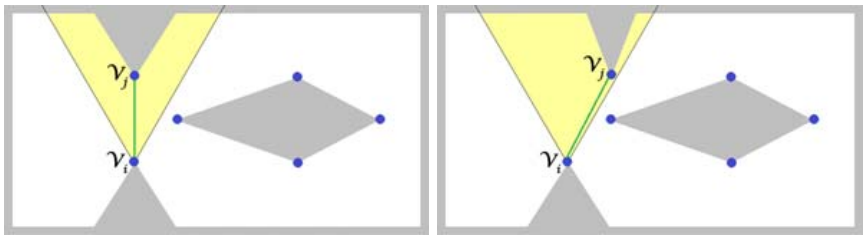

Figure 3.2: Vertex-Vertex portal creation. On left, v_i also falls within I_j, so it can be removed from V, on the right, v_i does not fall within I_j and since it is a notch it still needs to be split.

we want portals to be as short as possible, we first consider the closest point within the segment, which is calculated as the projection of v_i over $e_{j,j+1}$, so in this case $q = (proj_e v_i)$.

If q falls within I_i then a new portal is created and the algorithm proceeds with the next notch in V (see figure 3.3, left). But it could be possible that even though the edge $e_{j,j+1}$ is the closest element to v_i, we could have its projection falling outside $e_{j,j+1}$ or outside the interest area, I_i, and thus the portal between those two points would not be enough to split v_i in two convex angles (see figure 3.3 center and right).

Therefore if the projection is not a good candidate to create a unique portal, the algorithm considers four new candidates:

1. the two end vertices of $e_{j,j+1}$, v_j and v_{j+1} (see figure 3.3, center).
2. the two intersection points q_l and q_r (if they exist) where q_l is the intersection between the closest edge $e_{j,j+1}$, and the result of extending the segment $e_{i-1,i}$ (segment on the left of v_i), and q_r is the intersection between $e_{j,j+1}$, and the result of extending the segment $e_{i,i+1}$ (segment on the right of v_i) (see figure 3.3, right). There is a chance that depending on the orientation of each segment, none of those intersections exist, and therefore only the ends of segment $e_{j,j+1}$ are considered.

Among the four possible vertices mentioned above, the algorithm selects the closest one that falls within I_i and a new portal is created between v_i and the selected vertex. Algorithm 2 contains the pseudocode of this step (with AOI meaning area of interest).

In figure 3.3 we can see the three different types of portals created in the category of vertex-edge portal. As we can see the cases on the left (projection) and the right (intersection points) are the only cases where new vertices are added into the geometry. By construction, these new vertices can never be notches because they are the result of splitting an edge in two, therefore the algorithm does not need to do any further processing with them.

Algorithm 2 Portal Vertex-Edge algorithm

```
1:  procedure createPortalVertexEdge(Notch v_i, Edge e_{j,j+1})
2:      q ← proj_e v_i
3:      if isInAOI(v_i, q) then
4:          createPortalVertexVertex(v_i, q)
5:      else
6:          Let L be a new Dynamic List.
7:          L.insert(e_j)
8:          L.insert(e_{j+1})
9:          q_l ← lineSegmentIntersection(e_{i-1,i}, e_{j,j+1})
10:         if pointInSegment(q_l, e_{j,j+1}) then
11:             L.insert(q_l)
12:         q_r ← lineSegmentIntersection(e_{i,i+1}, e_{j,j+1})
13:         if pointInSegment(q_r, e_{j,j+1}) then
14:             L.insert(q_r)
15:         q ← closestPointInAOI(v_i, L)
16:         createPortalVertexVertex(v_i, q)
```

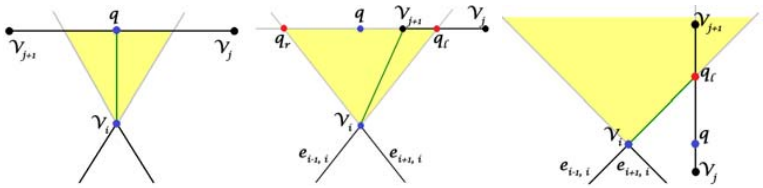

Figure 3.3: Vertex-Edge portal. Candidate point q being the projection (*left*), candidates being the end points of segment v_{j+1} *(center)*, and candidate points being the intersections *(right)*.

Note how this is the only step of the algorithm that may introduced new vertices in the geometry, known as boundary Steiner points.

3.2.1.3 Case 3: Vertex-Portal portals

In the case where the closest element to v_i in the geometry is a previously created portal, the treatment when creating portals differs from the vertex-edge portal since we do not want to have intersecting portals (or *T-shapes*), which would be the case for calculating a projection or intersection over an existing portal.

Therefore we assume that when the closest element is a portal p_k, we will need to create a new portal p_i with either end vertex of portal p_k. The algorithm selects the closest vertex that falls within I_i (figure 3.4, left and center). But since only vertices that fall within I_i can guarantee that v_i will get split into two convex areas, if none of the end vertices of p_k satisfy that requirement (figure 3.4, right), then the algorithm needs to create two portals instead of one. The new portals will be p_i which joins v_i with the left end of p_k and p_{i+1} which joins v_i with the right end of p_k. Notice that given the type of geometry we are dealing with, the interior angle between p_i and p_{i+1} will always be smaller than π, and therefore we guarantee that when adding these two portals, v_i will get split in three convex regions. In order to avoid intersection problems when the endpoints of p_k are not actually visible from the notch, we check for intersections between the segment formed by the notch and the endpoint of the portal and the rest of edges in the scene. If there are no intersections, then the portal can be created; otherwise, the portal is created with the best candidate of the closest intersected element.

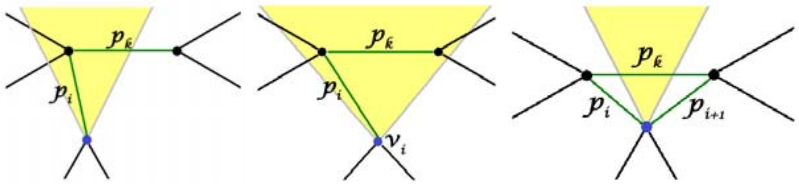

Figure 3.4: Vertex-Portal portal. Only one end of p_k falls within I_i *(left)*, both ends of p_k fall within I_i *(center)*, and none of the ends of p_k fall within I_i *(right)*.

Notice that when a vertex-portal portal p_i is created between a vertex v_j and a previously created portal p_k, we will have at least one vertex v_i, where both portals meet, since portals always meet at their ends which are located over existing vertices. In order to determine whether we could merge the two cells divided by portal p_k, the algorithm checks whether p_k is still a necessary portal, since it is possible that by adding p_i to vertex v_j, this vertex already gets split into two convex regions, and thus there is no need to have two portals

splitting one vertex.

To be able to remove portal p_k it is necessary to check whether both the left and right vertices of the portal need p_k not to be a notch. This step is performed by calculating the interior angle between the two neighboring segments of portal p_k at each end vertex (which can be edges of the geometry or other portals) and testing for convexity. If they both pass the convexity test, then we can remove p_k, and thus merge two convex cells into one larger convex cell (see figure 3.5). The pseudocode of the whole step is described at Algorithm 3.

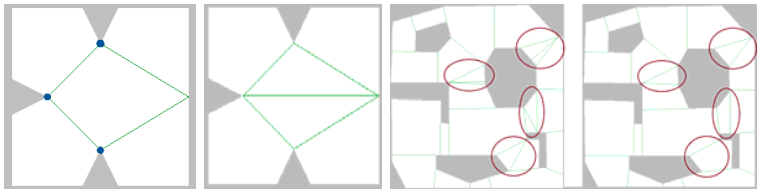

Figure 3.5: Removal of previously created portal, p_k when creating a new set of portals p_i and p_{i+1} (*left*), or several cases of just one new portal being created (*right*).

Algorithm 3 Portal Vertex-Portal algorithm

```
1:  procedure createPortalVertexPortal(Notch v_i, Portal p_k)
2:      if isInAOI(v_i, p_k[0]) & isInAOI(v_i, p_k[1]) then
3:          q ← closestPoint(v_i, p_k[0], p_k[1])
4:          createPortalVertexVertex(v_i, q)
5:      else if isInAOI(v_i, p_k[0]) then
6:          createPortalVertexVertex(v_i, p_k[0])
7:      else if isInAOI(v_i, p_k[1]) then
8:          createPortalVertexVertex(v_i, p_k[1])
9:      else
10:         createPortalVertexVertex(v_i, p_k[0])
11:         createPortalVertexVertex(v_i, p_k[1])
12:     if isNonEssentialPortal(p_k) then
13:         removePortal(p_k)
```

Therefore, the vertex-portal case is the only case of portal creation that may require up to two new portals instead of just one per notch, but in most cases when creating these portals we will be able to remove the original portal p_k, and thus we are on average creating one portal per notch.

3.2.2 Cell and Portal Graph construction

At this point we have split all the notches into convex regions, formerly named cells. All that remains is to identify each of this cells and compute its connectiv-

ity to create the *Cell and Portal graph (CPG)* that will be used later on by the pathfinding algorithm in order to find a valid path between the current position of the agent and its goal position.

The *Edge* data structure stores a pointer to the next edge that is properly updated when creating the portals. So to identify the cells, we just have to iterate over all the previously created portals and keep following the next pointer until we reach the portal edge we started with. By construction, the resulting polygon will be a convex one, i.e., a cell.

Finally, the cells connectivity (edges of the *CPG*) is computed by determining the neighbors of each cell. Two cells are neighbors if they share the same portal, so in one of the cells we are traversing the edge in a specific order, whilst in the other cell we are traversing the same cell in the opposite order.

3.3 Convexity Relaxation

The purpose of *navigation meshes* is to have a decomposition of space into walkable cells with portals joining those cells. Path planning algorithms are used to find the path between two cells of the *navigation mesh*, and a local movement algorithm usually deals with the character's displacement within a cell. Local movement algorithms, use different techniques to avoid obstacles that can also be used against small concavities in walls. Therefore, we can further reduce the number of cells in the final *navigation mesh* if we take this into consideration and relax the notion of convexity.

A vertex is defined to be convex if its internal angle is smaller or equal than π, otherwise it is a notch. This is the mathematical definition of convexity, but in some applications such as the creation of *navigation meshes*, it may not be necessary to consider such a strict definition. As described previously in this thesis, the purpose of *navigation meshes* is to have a decomposition of space into walkable cells with portals joining those cells. *Path planning* algorithms are used to find the path between two cells of the navigation mesh, and a *local movement* algorithm usually deals with the autonomous character's displacement within a cell. *Local movement* algorithms, use different collision avoidance behaviors to avoid obstacles that we can exploit also to overcome small concavities in walls. Therefore, depending on the *local movement* algorithm being implemented, we can relax the definition of convexity by allowing a certain convexity threshold α. Relaxing the definition of convexity results in a smaller number of portals since more cells can be merged together into α*–convex* cells. Note that the concept of convexity relaxation can also have applications in other areas, for example recently it has been used to accelerate ray tracing computation for rendering purposes [58].

Our first approach of exploiting this concept consisted on slightly increasing the internal angle of the *Area of Interest* of a notch, thus increasing the number

of candidates to create a portal. The target here is to increase the probability of finding another notch v_j in the *Area of Interest* of a given notch v_i and so a single portal is sufficient in order to break both notches if v_j is inside the *α–convex Area of Interest* of v_i, and v_i is inside the *α–convex Area of Interest* of v_j. Therefore, the following definitions are provided:

Definition 3.3.1. A vertex v_i is said to have *α–convexity* if its internal angle is smaller than $\pi + \alpha$. Relaxing convexity affects not only the classification of vertices into notches, but also the definition of the *Area of Interest* of a notch.

Definition 3.3.2. An area of interest I_i is said to have *α–convexity* if its internal angle is $\beta_i + \alpha$, where β_i is the original internal angle of I_i before applying convexity relaxation.

In order to avoid obtaining degenerated or non-simple polygons, it is necessary to refine the definition of the threshold per vertex, so that we ensure that the best candidate for any given vertex v_i, will never be laying over the same edge as v_i, or causing an intersection with the boundary of the original polygon. This is achieved by limiting $\beta + \alpha$ to always be smaller than π. And this leads us to the next definition:

Definition 3.3.3. A simple polygon with holes P is said to have been split into *α–convex* cells, when all its vertices have at most *α–convexity*.

The effect of increasing the internal angle of the I_i with the threshold α, implies a larger area to look for candidates, which not only reduces the total number of cells, but also implies a reduction in the number of ill-conditioned polygons (see figure 3.6).

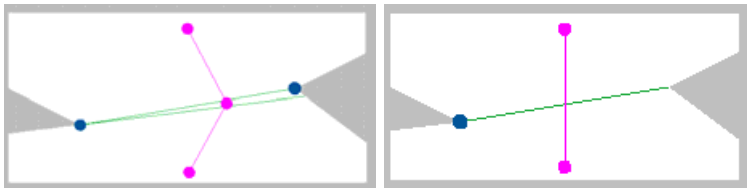

Figure 3.6: On the left a cell created with strict convexity, on the right the same scenario but applying the concept of convexity relaxation.

However, when running the first tests, we realized that this solution does not always give good results since it does not guarantee to reduce the number of created portals for all scenarios. If we observe the previously mentioned figure 3.6, there is no guarantees that the chosen value of convexity relaxation will be enough to break both notches using a single portal. Moreover, increasing the angle of the *Area of Interest* may lead to the creation of more ill-conditioned cells, which is precisely one of the of issues we want to avoid.

Therefore, we maintain the key idea of the *convexity relaxation* concept described before, but we further refine our approach in order to overcome those

limitations. Our final approach has some similarities with the Ramer-Douglas-Peucker algorithm [11], which is commonly used to reduce the number of points in a curve that is approximated by a series of points. Our method focuses exclusively on notches in the floor plan, since our goal is to determine which notches can be ignored when creating portals. So the algorithm is run for every sequence of notches found between two convex vertices (by sequence we mean 1 or more). Given the input threshold τ, the algorithm recursively finds the notches that need to be kept in order to create portals. So at each step it finds the center notch of the sequence and calculates the distance between the center notch and the line segment joining the first and last vertex of the sequence. If the distance is bigger than τ then that notch needs to be kept and the algorithm calls itself recursively for the sequence of notches before and after the center notch. The recursivity stops when the distance is smaller than τ, and all notches not marked as kept are ignored. Note that we are not eliminating these notches, we are simply not creating portals to split these notches into convex vertices. The following definitions are provided:

Definition 3.3.4. A vertex v_i is said to have α*–convexity* if its internal angle is smaller than $\pi + \alpha$. This affects only the classification of vertices into notches, the *Area of Interest* of a notch is not affected.

Definition 3.3.5. A sequence of notches N is said to have τ*–convexity* if all notch v_i in N is at a distance less or equal than τ with respect to the segment formed by the initial and final endpoint of such a sequence.

Definition 3.3.6. A simple polygon with holes P is said to have been split into $\alpha + \tau$*–convex* cells, when all its vertices are α*–convex* and all the sequence of notches are τ*–convex*.

The user is able to manually tune the α and τ thresholds in order to find the best compromise between number of cells generated and crowd behavior. Figure 3.7 shows the benefits of applying the *convexity relaxation* method over a relative complex scenario. This solution allows us to keep a detailed geometry for local movement purposes, yet reduces the final number of cells to speed up path planning.

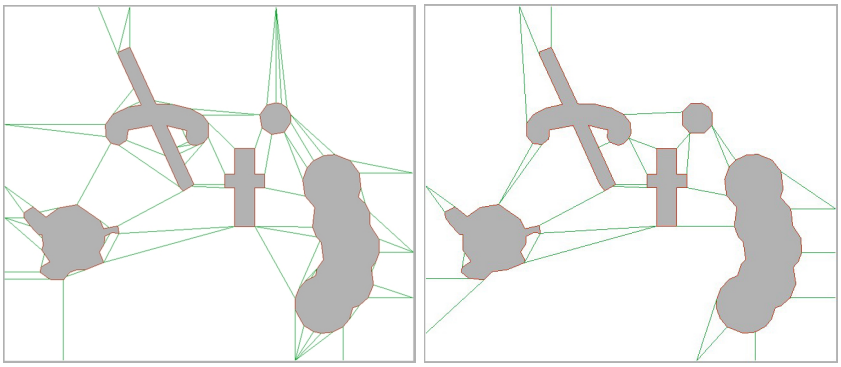

Figure 3.7: The resulting CPG with $\tau = 0$ (left) contains 69 cells. This is the case where we consider only strict convexity. In the same example, the CPG produces only 29 cells with $\tau = 0.5$.

3.4 Discussion on the Navigation Mesh Created.

The optimal number of cells when decomposing a polygon without holes by diagonals is known to be within the following bounds [13]:

$$\left\lceil \frac{r}{2} \right\rceil + 1 \leq OPT \leq 2r + 1$$

where OPT is the optimal number of convex sub-polygons into which a polygon P may be partitioned, and r is the number of notches.

Proof. Our algorithm has the same bounds as the subdivision based on diagonals. Just like in the case of diagonals we know that at most two portals are needed per notch which gives the upper bound in the number of cells being $2r + 1$ (corresponding to case 3 of our algorithm *vertex-portal portals*, and the lower bound of $\left\lceil \frac{r}{2} \right\rceil + 1$ corresponds to case 1 of our algorithm *vertex-vertex portals*).

The case of decomposing a polygon with holes has similar bounds, but including the number of holes in the equation. As proven by Fernandez et. al.[14], we can consider that each hole can be joined to the boundary of the polygon with one portal, so this portal does not create a cell. Therefore we need to subtract the number of holes to both limits of the optimal number of cells.

The optimal number of cells, OPT, for a convex decomposition of a polygon with holes is within the bounds:

$$\left\lceil \frac{r}{2} \right\rceil + 1 - h \leq OPT \leq 2r + 1 - h$$

Where r is the number of notches, and h the number of holes (which represent obstacles in our case). The lower bound corresponds to the ideal case where all portals created join two notches, and the higher bound corresponds to the case where a portal needs to be created for each notch to turn it into a convex vertex.

Our *NavMesh* generator decomposes the polygon with holes by creating either one portal per notch within the Area of Interest, or else two portals per notch at most when necessary (typically the case of the closest element being another portal). Therefore the proof given by Fernandez et al. [14] for the case of diagonals, applies also to our solution. Moreover, since our algorithm favors the creation of portals within the area of interest, this guarantees that in most cases only one portal will be needed per notch. As we will see in the results section the number of cells created by our algorithm tends to be below $r - h$. We will show this in the results section through a testbed of scenarios that have been evaluated. Note how the lower bound corresponds to situations where it is possible to create all portals between two notches of the polygon with holes. The number of pairs of notches that can both be broken into convex vertices by adding a single portal depends strongly on the geometry of the polygon with holes.

Also, since at most two diagonals are essential for any notch as stated by Hertel et al. [30], then any subdivision without inessential diagonals is within four times of the minimum subdivision (which gives us the upper bound). The lower bound is given by the best case scenario where we only need one portal to join pairs of notches.

In the case of subdivision based on diagonals, there may be more than one edge created per concave vertex, since it is possible that no vertex of the geometry fall within the area of interest. If we create an edge outside this area, we will split the current concave vertex into two regions, one convex and one concave, therefore, we will still need additional edges to guarantee that all the final regions are convex. With our heuristic based on the definition of an Area of Interest we narrow down the number of notches that need two portals. Moreover even in the exclusive situation of notch-portal where two portals are needed, we also evaluate whether the initial portal is not needed anymore (similar to the case of deleting inessential diagonals). When this occurs, we are then also creating only one more portal in the total portal count.

3.5 Results

The method presented in this chapter generates *navigation meshes* that can successfully be used for path finding and driving local movement between cells. The convex space partition generated in all cases contains a number of cells lower than the number of notches in the geometry. Since calculating the optimal convex space partition for a simple 2D polygon with holes is NP-hard, we consider

that any algorithm that can guarantee a maximum number of cells equal to the number of notches can be considered a good near-optimal subdivision.

Another advantage of our method is that we obtain NavMeshes without degenerated polygons, and almost no ill-conditioned polygons. Therefore our NavMeshes can be used with any local movement technique guaranteeing natural looking movement of the characters.

Even though the time complexity of our method was not the main concern, the *NEOGEN-2D* can generate NavMeshes with a temporal cost of $O(r \cdot n)$, where r is the number of notches, and n the number of vertices.

We have tested scenarios with increasing numbers of vertices. For each scenario, we have fixed the α parameter of Convexity Relaxation to be 5° (this means that a vertex needs to have an internal angle greater than 185° in order to be considered a notch) and for the τ parameter, we have applied 5 different values ranging from not applying convexity relaxation at all to the maximum value allowed by our implementation (0, 0.25, 0.5, 0.75 and 1). The obtained results are organized into three different tables, depicting the number of real notches taken into account to generate the convex partition (table 3.1), the total number of portals generated (table 3.2) and the number of cells of the resulting *NavMesh* (table 3.3). For further clarity, we also show the percentage of notches left on average after applying convexity relaxation (Figure 3.8).

For the results provided in this paper we have used scenarios with the number of vertices ranging from 52 to 3012. We have included scenarios from the literature both with holes and without holes, to show how our algorithm works well in both cases. Figure 3.12 shows some of the scenarios with the decomposition given by our algorithm without convexity relaxation. For further quantitative evaluation of our results against state of the art methods, we refer the reader to the detailed comparative evaluation presented in the paper by Toll et al. 2016 [96].

Let us study the ratio C/r, where C is the number of cells in the decomposition, and r is the number of notches. From the Optimality discussion in Section 3.4, we can extract that the Optimal decomposition occurs when $C/r \in [0.5, 2]$. Figure 3.9 shows the ratio obtained for the scenarios in our testbed. The orange shaded area indicates the optimal bound based on the number of notches. The graph shows the ratio for the original case without applying convexity relaxation ($\tau = 0$) and the resulting ratio as the τ value increases. From this graph, we can clearly see that our results appears to be always under the value 1, which corresponds to our algorithm generating at most one cell per notch. This value would be in the middle of the optimal bound. We can also observe in this graph the benefit of using convexity relaxation. In our experiments we could observe how convexity relaxation can allow us to obtain results even lower than the minimum bound of optimality, when considering the initial number of notches.

If we plot the number of cells against the number of notches that were not discarded due to convexity relaxation, we can see how our results still holds below the ratio 1 (See figure 3.10).

Figure 3.11 shows an example of a NavMesh obtained with our method. The associated video shows the result generated step by step as well as its final *Cell-and-Portal Graph.* [1]

Finally, our automatic NavMesh generator system has been successfully integrated into *Ninja Flag*, a tactical multiplayer online game, inspired by the famous outdoor sport called *Capture The Flag*, that we have developed.

To determine how a character moves from one cell to another and to describe its behavior inside a convex cell, we use several steering behaviors [74]. Attractors are set based on the agents' projection over the portals, as they move within a cell. This avoids agents sharing the same attraction points when crossing and leads to natural looking movement.

Overlapping is solved by integrating the physical library *Bullet* [8]. To keep track of characters within a cell at all times, we employ *Bullet's GhostObjects*, which are special physic bodies that do not interact with the rest of the standard physics bodies of the simulation, but they track an updated list of the objects they are in contact with.

Map	Vertices	Notches (cr=0)	Notches (cr=0.25)	Notches (cr=0.5)	Notches (cr=0.75)	Notches (cr=1)
circle1	52	32	32	32	20	20
circle2	80	48	48	32	20	20
funny	100	42	39	30	23	20
box100	100	56	56	35	35	28
guitar	144	76	48	27	23	19
debug	200	79	18	5	1	1
circle3	256	128	36	36	20	20
bird	275	102	27	16	14	9
superior	518	172	60	34	27	21
crazybox1	812	728	584	530	458	440
nazca_heron	1036	272	78	42	34	29
nazca_monkey	1204	374	186	125	99	86
crazybox2	3012	2212	845	624	495	441

Table 3.1: Number of notches for the testbed environments, applying different values of convexity relaxation.

[1] *http://www.lsi.upc.edu/~npelechano/videos/MIG2011_NavMesh.avi*

Map	Vertices	Portals (cr=0)	Portals (cr=0.25)	Portals (cr=0.5)	Portals (cr=0.75)	Portals (cr=1)
circle1	52	31	31	31	20	20
circle2	80	41	41	28	17	17
funny	100	63	60	48	37	30
box100	100	54	54	34	34	28
guitar	144	59	38	23	20	16
debug	200	140	24	4	1	1
circle3	256	126	36	36	20	19
bird	275	99	26	16	14	9
superior	518	179	69	34	33	22
crazybox1	812	634	492	444	387	369
nazca_heron	1036	258	73	42	31	29
nazca_monkey	1204	362	181	123	94	85
crazybox2	3012	2157	816	612	486	433

Table 3.2: Number of portals for the testbed environments, applying different values of convexity relaxation.

Map	Vertices	Cells (cr=0)	Cells (cr=0.25)	Cells (cr=0.5)	Cells (cr=0.75)	Cells (cr=1)
circle1	52	28	28	28	17	17
circle2	80	38	38	25	14	14
funny	100	53	50	40	31	26
box100	100	48	48	28	28	22
guitar	144	57	36	21	18	14
debug	200	108	22	5	2	2
circle3	256	123	33	33	20	19
bird	275	100	27	17	14	9
superior	518	168	58	28	25	21
crazybox1	812	562	420	372	315	297
nazca_heron	1036	258	73	43	32	30
nazca_monkey	1204	361	181	123	94	86
crazybox2	3012	2085	744	540	414	361

Table 3.3: Number of cells for the testbed environments, applying different values of convexity relaxation.

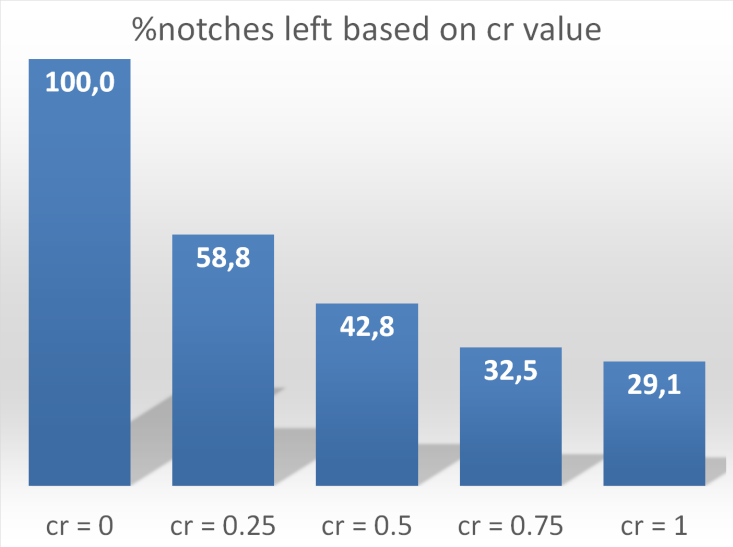

Figure 3.8: Percentage of notches left on average after applying convexity relaxation with $cr \in [0, 1]$

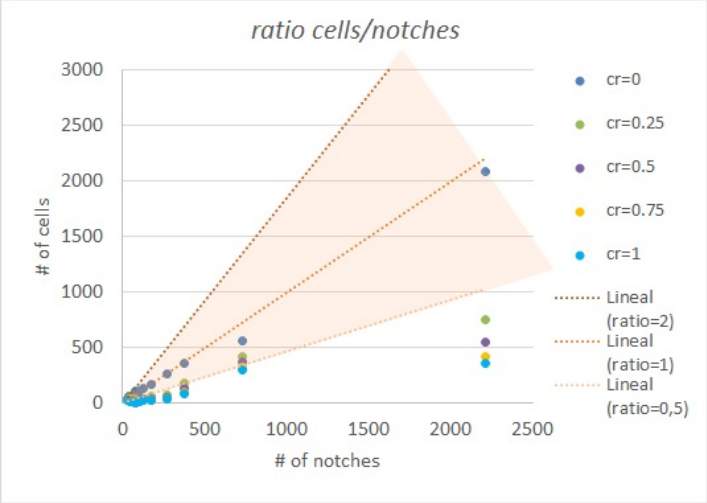

Figure 3.9: Ratio C/r as the convexity relaxation increases, when considering the original number of notches in the environment.

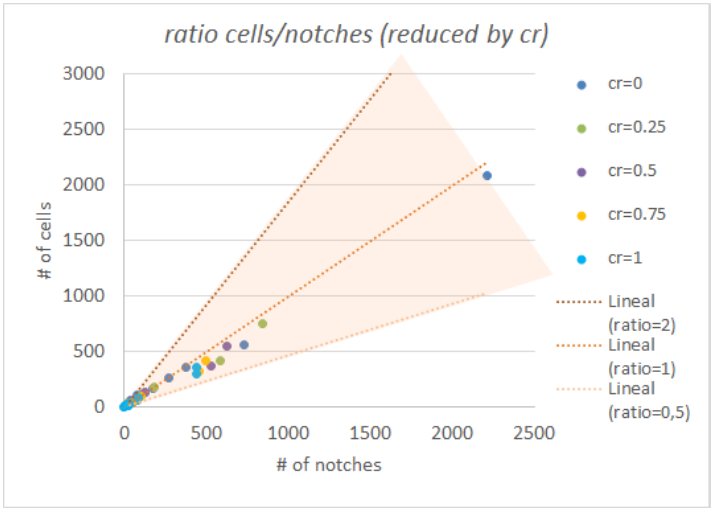

Figure 3.10: Ratio C/r as the convexity relaxation increases, and we consider only those notches that are not discarded due to convexity relaxation.

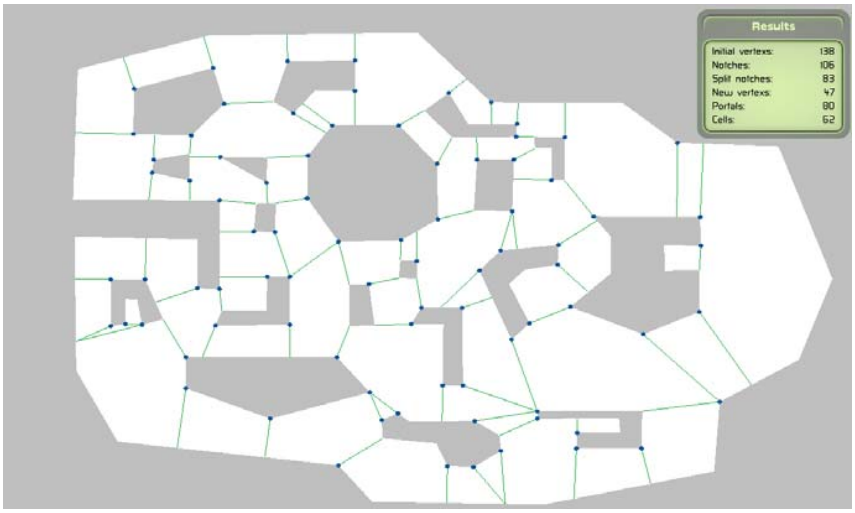

Figure 3.11: Example of a NavMesh with 106 notches and 62 cells. Green lines represent portals.

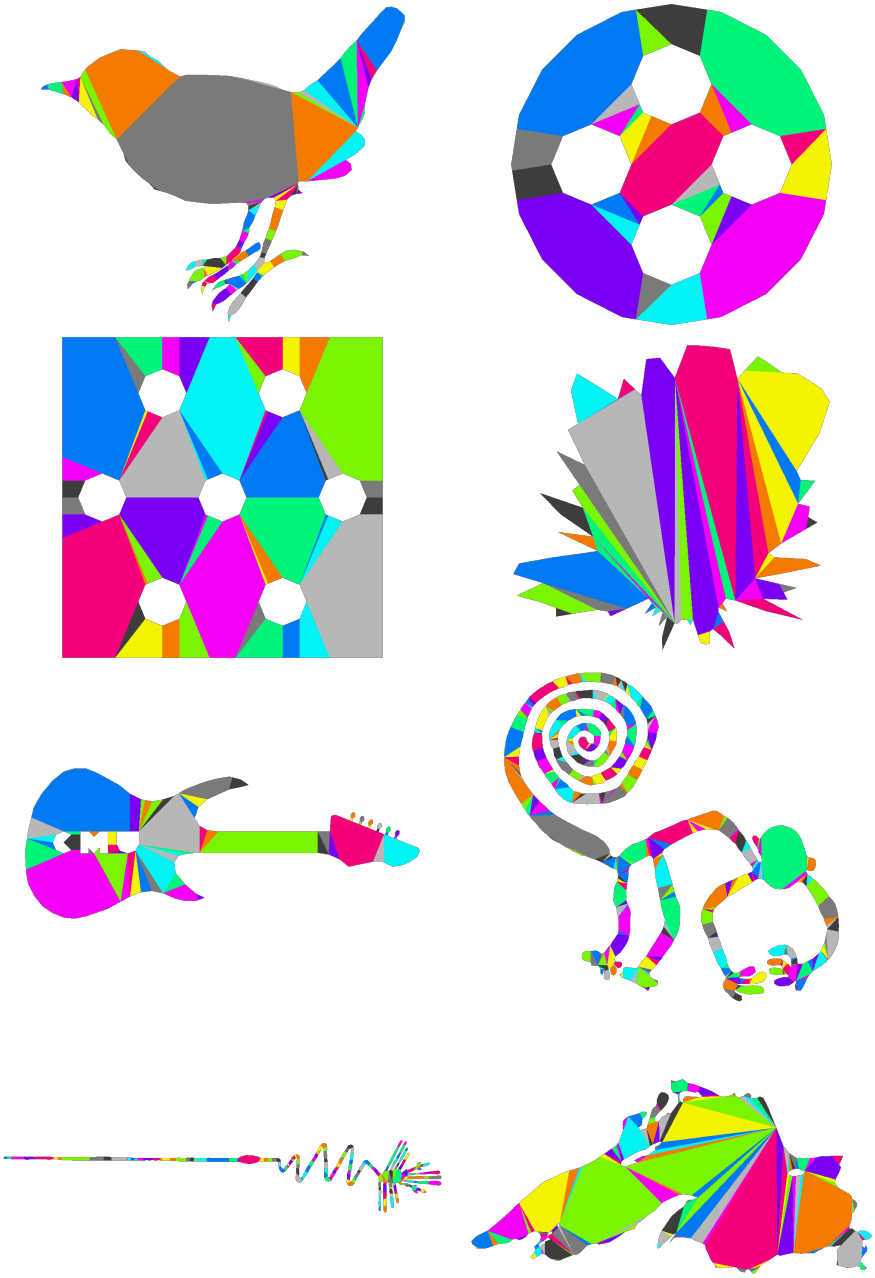

Figure 3.12: Testbed of scenarios

3.6 Conclusions

The method described in this chapter provides an automatic convex cells subdivision for any simple polygon with our without holes. The polygons can represent the floor plan of a given environment, with holes representing static objects such as walls.

We have introduced a novel algorithm which focuses on the idea of sequentially splitting notches into convex areas instead of being limited to some preliminary triangle subdivision. Since our approach is based on subdividing the original polygons with segments, instead of diagonals, we achieve on average a smaller number of convex cells in the environment than previous work in the literature based on diagonals. Even though the optimal solution cannot be guaranteed for the case of simple polygons with holes, our algorithm follows a number of heuristics that help us to get as close as possible to the lower bound of the optimal range. These heuristics are: (1) The concept of *Area of Interest* driving the creation of portals allow us to break most notches using only one portal (thus creating a cell per notch); (2) Having the fist step of the algorithm focused on creating notch to notch portals whenever possible, which allows to break two notches with just one portal (thus creating one cell for each pair of nodes); (3) creating two portals per notch only when encountering the situation of the closest element to a portal being a notch with both endpoints outside the *Area of Interest*; and finally (4) a convexity relaxation method to further reduce the number of cells. Note how (1,2,3) guarantee a result within the bounds of optimality, and (4) can reduce the number of cells even bellow the bounds of optimality. This has allowed us to obtain a navigation mesh generator that creates the lowest number of cells when compared to other state of the art methods. For a detailed quantitative comparison we refer the reader to the comparative paper by Toll et al. [96].

Our goal with this first contribution was to develop a method that could create navigation meshes with the smallest number of cells possible. This comes at the cost of having more complex cells, i.e. more vertices per cell. It is important to emphasize that in general the cost of path finding depends on the size of the graph, so our solution offers a good general navigation mesh for this purpose. However there are other contributions in the literature that have focused their effort in methods that obtain higher performance by having a compact representation of the navigation mesh. In those cases it is preferable to have simpler cells (for example triangles) despite having larger graphs. So in the end it depends on the path finding algorithm chosen and on the implementation, but for the general case of simply computing A* (or any variant of A*) performance benefits from having smaller graphs.

Note that in this chapter we have introduced an algorithm to decompose a single polygon with holes into convex (or almost convex) cells. Therefore if we had the case of a game scenario composed of several islands, the current method would need to be extended to handle such environments. One straight forward

solution would be to run our algorithm for each island, and then include special portals that allow the character to move between navigation meshes.

Chapter 4

Computing NavMeshes for 2.5D geometry

In the previous chapter we have presented a novel method to obtain a near-optimal navigation mesh from a floor map given as a polygon with holes. But this is rarely the case in a video game or any virtual reality application. Most of the time, the given scenario will be a complex 3D geometry given as a large set of objects. Each of this objects, together with the walkable surface will be most likely given as triangle mesh or a polygon soup. Therefore the immediate next step in this thesis, was to develop a method to be able to process such input and convert it into a polygon with holes so that we can automatically obtain the desired navigation mesh.

In this chapter we present a novel, robust and efficient GPU based technique to automatically generate a *navigation mesh* for complex 3D scenes. Our method consists of two steps: firstly, starting with a 3D scene representing a complex environment of one floor with slopes, steps, and other obstacles, it automatically generates a 2D representation based on a single polygon (floor) with holes (obstacles). This step can handle degeneracies of the starting 3D scene model, such as interpenetrating geometry. Secondly, a novel method that exploits the GPU efficiency is used to automatically generate a near-optimal convex decomposition which will represent the cell and portal graph of the environment. Such convex decomposition is a 2D representation of the walkable areas of the environment with portals indicating the crossing borders. In the results section we will show how the presented technique is not only more robust than previous CPU methods, but also how for the tested environments with up to 1000 vertices we obtain results up to five times faster.

The main contribution of this chapter is a novel GPU based method to generate a *NavMesh* for a given 3D scene, representing a complex environment of a single floor. Our method has two main steps: firstly it abstracts away

the information of the 3D model that represents the scene (with its slopes, steps and other obstacles) to automatically convert it into a 2D representation based on a single simple polygon (floor) that can contain holes (obstacles). Secondly, it automatically generates a near-optimal convex decomposition of this 2D representation. Our method is robust against degeneracies of the starting 3D model, such as interpenetrating geometry.

Unlike the previous chapter that worked directly processing vertices of the input geometry (exact method), the one in this chapter does not need a clean geometry to work with. The filtering process acts as a sampling that allows to generate a simple polygon with holes from a polygon soup. This is a very important advantage because very often the geometry obtained from modeling tools is far from perfect. So typical artifacts that we can find include intersections, gaps, non-manifolds or degenerate triangles. Sampling methods such as Recast and ours, allow us to handle imperfect geometry. The usual tradeoff when compared to exact methods, is that the resulting *navigation mesh* is not as adjusted as in exact methods. However as we will show in the results section, our algorithm obtains a *NavMesh* with a very good adjustment to the input geometry.

4.1 Converting a 3D World into 2D Polygons

As most *NavMesh* generation methods, the navigation mesh is constructed in 2D. However, especially in the case of video games, the virtual world is typically generated using a 3D software modeler. Since we want the method to be fully automatic, the first step of our method transforms the 3D input data into a 2D representation. In particular, the input required by the *navigation mesh* Generator consists of a single polygon defining the floor, with the vertices given in counter-clockwise order, and holes representing the static obstacles with the vertices given in clockwise order. The 2D Abstraction step is subdivided in several stages, as can be seen in figure 4.1.

4.1.1 Normal and Depth Map Extraction

The first stage of the pipeline takes the 3D model of the scene as input and performs a render of the model from a top view, using an orthographic camera. A texture is created using the fragment shader, that stores the normal per fragment (red, green and blue channels) and its normalized depth (alpha channel). Figure 4.2 shows the resulting Normal-and-Depth Map for a given scene.

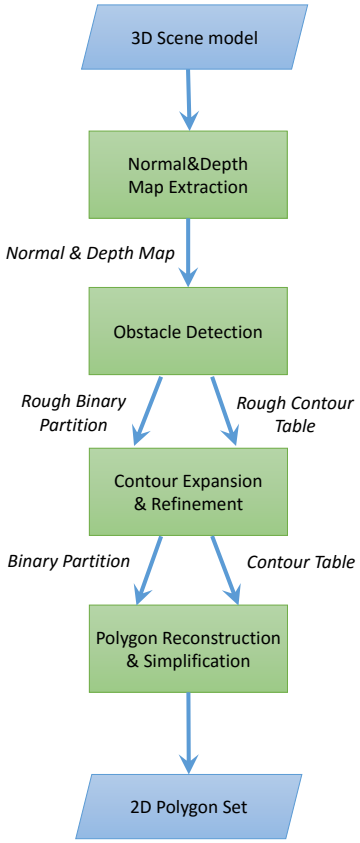

Figure 4.1: This figure describes the data flow of the pipeline of the 2D Abstraction step to convert from the 3D world to the 2D representation.

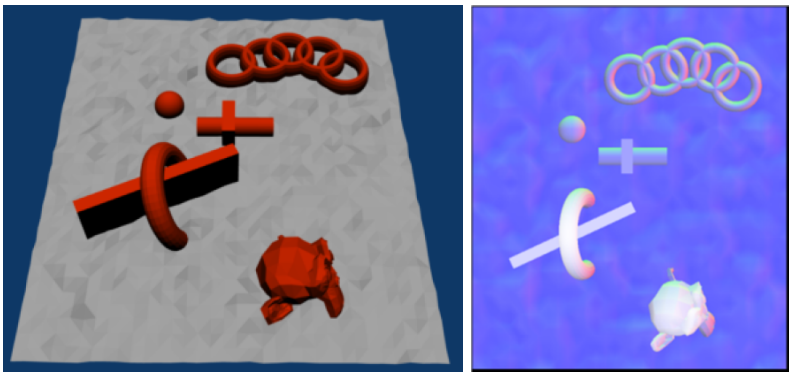

Figure 4.2: The initial 3D scene (Left) and its Normal-and-Depth Map generated with the shader (Right).

4.1.2 Obstacle Detection

The target of this stage is to identify walkable space (floor) vs. non-walkable (obstacles). The obstacle detection is solved using a flood fill algorithm, where the seed is introduced by the user over a walkable area (notice that this is the only input required by the user). The Normal-and-Depth map is used to determine if a neighboring fragment is similar to our current fragment. Two adjacent fragments are similar if the character can overcome the angle formed by their normals and the difference of depth. These parameters are configured through the application and depend on the walking abilities of the characters. If the neighbor fragment is reachable from the current one, then it belongs to the walkable area; otherwise it belongs to the frontier of an obstacle (contour).

The output of this stage are a Rough Binary Partition (RBP) and a Rough Contour Table (RCT). The former is a binary image representing the walkable areas (white pixels) and the obstacles (black pixels), and the latter is a table containing those pixels marked as contour (black pixels in the RBP that have at least one white neighbor). In Figure 4.3 we can see the binary partition with the walkable areas and the obstacles. Notice that the torus is treated as a solid obstacles seen from above and thus the floor underneath it, will not be treated as walkable.

4.1.3 Contour Expansion and Refinement

In order to ensure a one pixel wide continuous contour with an area greater than zero (i.e. no obstacles of size one pixel or line obstacles) the RBP and RCT need to be further refined.

This stage is subdivided into two sub-steps. Firstly, the contour is expanded by iterating over all the pixels in the Contour Table marking as contour those

Figure 4.3: The Rough Binary Partition resulting of the Obstacle Detection stage.

adjacent pixels that in the binary partition belong to the floor, i.e. white pixels. The target of this sub-step is to avoid future degeneracies such as having obstacles mapped into a single vertex.

The Contour Refinement step removes those contour pixels that have end up completely surrounded by black pixels i.e. pixels of an obstacle, and hence, they do not belongs to the frontier of an obstacle anymore. Figure 4.4 shows these two steps over a given obstacle.

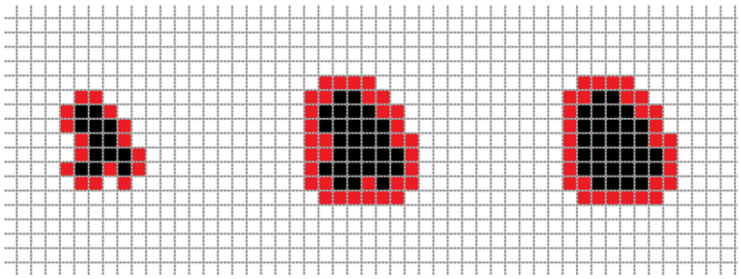

Figure 4.4: The initial contour of an obstacle (Left); the expanded contour (Center); the refined contour (Right).

At the end of this process we obtain the final Contour Table and Binary Partition adequate to carry out polygon reconstruction.

4.1.4 Polygon Reconstruction and Simplification

This step will generate the 2D model representing the floor and obstacles to feed the *NavMesh* generator. Firstly, the pixels on the Contour Table are sorted by its x coordinate, i.e., they are sorted from left to right. If the x coordinate of two contour pixels is the same, they are sorted by the y coordinate from top to bottom. Each contour pixel is considered a vertex of a polygon and then a simplification method is used to reduce the final number of vertices. Initially all contour pixels are marked as not-visited.

The algorithm proceeds by iterating over all the pixels on the Contour Table, until it finds the first not-visited contour pixel. The order of the Contour Table guarantees that this pixel is the most left one of a polygon on the Binary Partition. When reconstructing the floor, the vertices have to be given in counter-clockwise order, so for the most left contour pixel, we have to start moving to the S, SE or E neighbor pixel that is contour. If we are reconstructing an obstacle, the vertices have to be given in clockwise order, so we have to start moving to the N, NE or E. Figure 4.5 exemplifies the process of reconstructing an obstacle from its most left contour pixel C. In this case, the vertices have to be given in clock-wise order, so the neighbor chosen is the one marked with E.

Once the first neighbor has been decided, the process continues by selecting and setting as visited at each iteration the contour pixel that is closest to the current one, that has not been visited yet. In this case, all the adjacent neighbors of the current pixel are checked. The Contour Expansion and Refinement stage ensures that we always have an unequivocal neighbor contour pixel to choose as next. It also ensures that every reconstructed polygon has an area greater than 0, and that we do not have degeneracies such as obstacles reconstructed as a single point. The process of reconstructing a polygon ends when the start pixel is reached and the process of reconstructing all the polygons finishes when all the pixels on the Contour Table have been marked as visited.

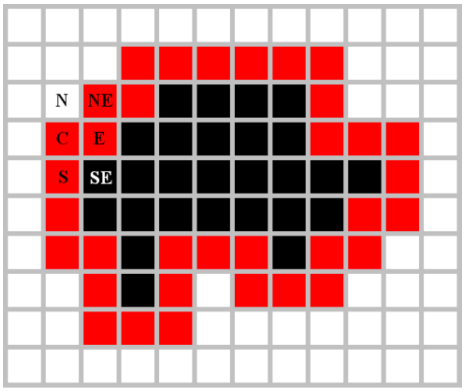

Figure 4.5: The most left contour pixel C of an obstacle and the potential neighbors that can be chosen as next.

To reduce the total number of vertices per polygon, the first straight forward simplification consists of eliminating all vertices that belong to segments aligned horizontally, vertically or with 45^o angle and are not end points. This pre-simplification step is done during the reconstruction process. Next the Ramer-Douglas-Peucker Algorithm [72, 11] is applied in order to further simplify the polygon (figure 4.6).

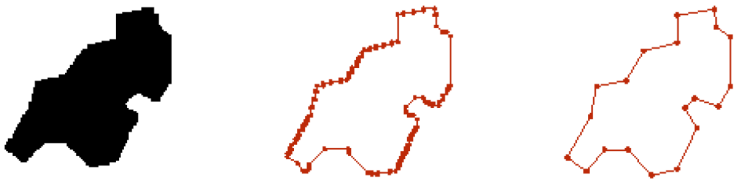

Figure 4.6: A polygon on the Binary Partition (Left); A high density pre-simplified polygon (Center); The final simplified ¡polygon (Right).

4.2 Automatically Generating NavMeshes

The algorithm presented in the previous Chapter 3, which was fully implemented on the CPU, consisted on identifying the notches (vertices with an angle greater than π) and transforming them into convex vertices. Notches are vertices that cause concavities in the geometry, and our algorithm is able to transform them into convex vertices, providing a near-optimal convex partition of the input polygon. The transformation from concave to convex is performed by creating a portal between each notch and the closest element (vertex, edge or portal) lying inside the Area of Interest of the notch. The Area of Interest is determined by the area formed by prolonging the incident edges of the notch. The method ensures that in the cases where the closest element is a vertex or an edge, only one portal needs to be created per notch. In the cases when the closest element is a portal with neither endpoints lying inside the Area of Interest, it is required to create two portals to transform the notch into a convex vertex. In this section we present a novel GPU version of the aforementioned CPU solution which aims to improve the efficiency of computing the closest element to a notch.

4.2.1 The GPU based version

The CPU solution presented in Chapter 3 has a cost of $O(n \cdot r)$, where n=*number of vertices* and r=*number of notches*. So if r is similar to n, the algorithm to generate *NavMeshes* has a $O(n^2)$ cost to solve the problem. This is not an important handicap a priori, since the *NavMesh* construction is normally an offline process, but it becomes an issue when dealing with dynamic environments that require continuous updates of the *NavMesh*, as it happens in video games.

The proposed method based on GPU, starts by assigning a unique identifying color to each edge of the scene. Then, the 2D scene is rendered from the point of view of every notch, with the parameters of the camera set based on the characteristics of the notch. The position of the camera is given by the position of the notch, the FOV of the camera is equal to the angle formed by the prolongation of the edges that define the Area of Interest of the notch and the forward direction of the camera is defined as the sum of the unitary vectors that define the Area of Interest. Once the camera has been configured, the scene is rendered and the result is stored on a one-dimensional texture that contains those elements visible from the point of view of the notch, as can be seen in figure 4.7.

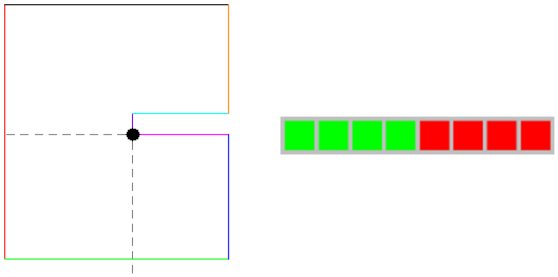

Figure 4.7: A simple scene with all edges drawn with a unique color (Left) and the texture generated from the point of view of the notch (Right).

To recover the edges visible from the notch, we check every pixel of the texture. The color of such pixel identifies the edge. Then, we determine which of those edges is the closest one to the notch and we create a portal with its best candidate. We cannot simply read the depth of each pixel, because we need Euclidean distances to the notch.

A critical parameter that affects directly the performance of the algorithm is the *zFar* of the camera. To avoid rendering an unnecessary number of elements that are occluding each other, the *zFar* is dynamically updated. Initially the *zFar* is set to $1/10^{th}$ of the diagonal of the bounding box of the scene. After each render, if no element has been rendered, the zNear is set to the current *zFar* and the *zFar* is doubled in order to carry out another render. This process continues until at least one element has been found that lies on the Area of Interest of the notch with the current *zFar*. Notice that this implies several renders for some notches, but we have found empirically that the *zFar* converges towards an optimal value that results in the most efficient render for a large number of the notches in the given scene. The entire process is described on figure 4.8.

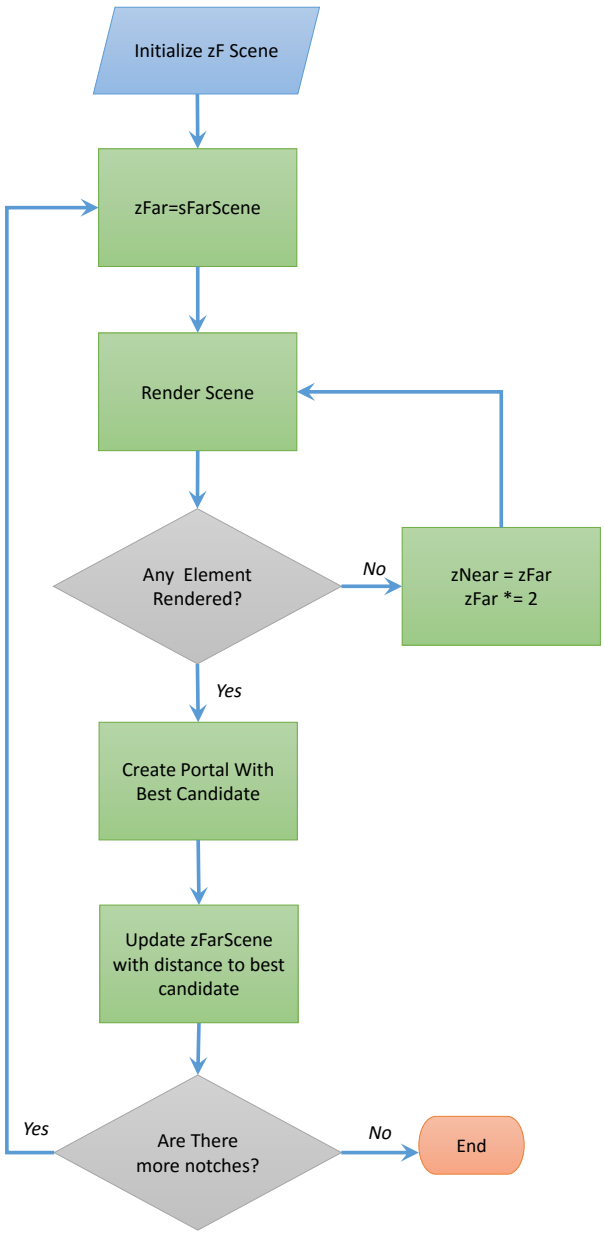

Figure 4.8: Diagram that describes the GPU based version.

4.2.2 The Portal Vertex-Portal case

The most complicated case that our algorithm must handle is when the closest element to the notch is a previously created portal. In order to avoid intersections between portals, we create a portal between the notch and one of the endpoints of the portal that is inside the Area of Interest of the notch. If neither endpoint lies within the Area of Interest, then two portals are created to join the notch with each of the endpoints of the previous portal.

In order to avoid intersections with the geometry, the CPU version checks if any of the existing edges already intersects the portal that we want to create with either endpoint of the closest portal. In GPU mode this problem is solved by using one extra render step. The new Area of Interest is defined as the one delimited by the segments that join the notch with the endpoints of the previous portal as can be seen in figure 4.9.

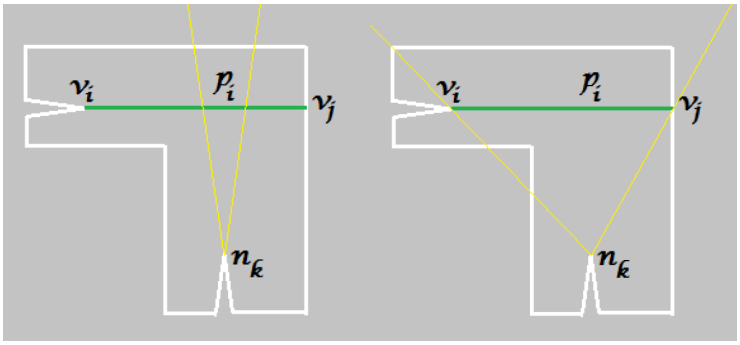

Figure 4.9: The original Area of Interest of a notch n_k (Left). When the closest element is a portal p_i, the new Area of Interest is defined by the notch and the endpoints of the portal (Right).

The camera parameters are thus updated accordingly and a new render of the scene is performed. Then the algorithm checks for intersections between the segments joining the notch with the endpoints and the edges that appear on the new render. Notice that the GPU version needs to check against a reduced number of edges unlike the CPU version that checks against all edges in the scene.

4.3 Results

In this chapter we have presented a framework to obtain *Navigation Meshes* from 3D complex environments consisting of one layer where we could carry out navigation for characters. Firstly a method has been described to abstract the 3D geometry into a 2D simple polygon with holes. The results shown in figures 4.10 and 4.11 demonstrate the robustness of the method to deal with

environments containing complex obstacles. The method provides the flexibility of being adjusted to the walking abilities of the characters, to determine the height of the steps, and the angle of a ramp that a character can easily overcome. Secondly we described a novel GPU based approach to speed up the search for closest element to a notch.

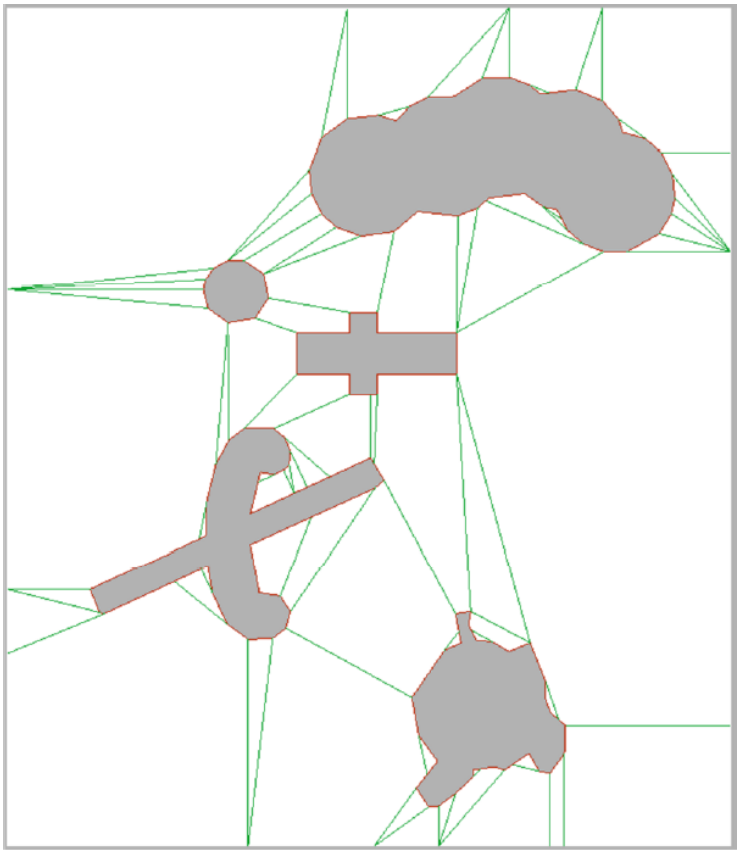

Figure 4.10: The resulting NavMesh of the scene described in figure 4.2.

The experimental results have been obtained on a NVIDIA GeForce 8800 GTX and an Intel Core 2 Quad Q6700 at 2.66GHz with 8 GB of RAM. We have tested the new GPU based algorithm presented in this chapter to generate automatically *NavMeshes* against the previously proposed CPU version.

To test the overall performance of the algorithm, we created 10 scenarios of increasing complexity ranging from 23 vertices to 965. The algorithm applied dynamic *zFar* calculation. Figure 4.12 shows the time taken by both CPU and GPU implementations. As we can see, the time taken by the CPU version to solve the problem increases quadratically, whereas the GPU version increases nearly linearly with the number of vertices of the environment. Notice that the

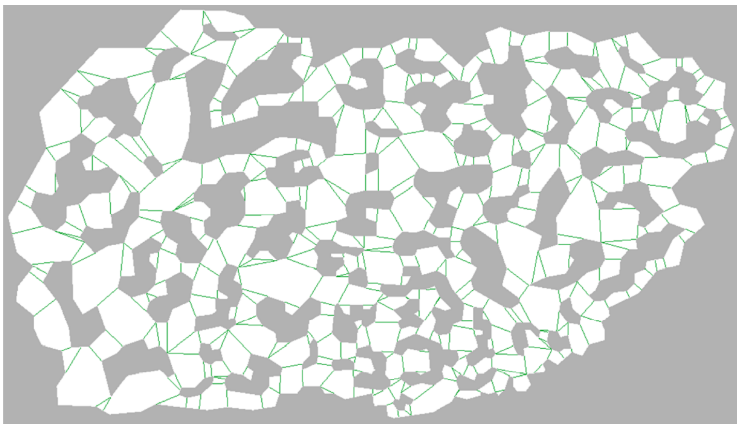

Figure 4.11: The resulting Navigation Mesh for the scene containing 965 vertices and 650 notches.

GPU algorithm can be quadratic in the worst case, but in practical scenarios it performs with nearly linear time, since it applies geometry culling using an octree to render the 1D texture for each notch.

The experimental results were obtained performing renders over a viewport of 1x32 pixels that were then mapped onto a texture. The size of the texture (and thus the viewport) used is important for the overall performance of the algorithm, as can be seen in figure 4.13. We have found empirically that a size of 32 pixels for the texture is adequate to correctly identify the closest element. This comparison table was obtained with the largest scene of 965 vertices, since for smaller scenes the difference is less significant.

The value of the *zFar* chosen for performing the render from each notch has also an impact on the performance, since it determines how many segments get discarded at an earlier stage of the graphics pipeline. A large *zFar* will guarantee that all segments visible from the given notch are rendered, but with a high cost, whereas a small *zFar* will result in faster renders but may not render segments that are visible and relevant for creating the *NavMesh*. The optimal *zFar*, is thus the shortest one that allows the closest element to be rendered without rendering many additional segments that are far away and thus either not visible or simply not relevant.

In order to calculate the optimal *zFar*, we have carried out an experiment with the largest scenario. Empirically we found that for the given scenario, the optimal *zFar* was 2. Figure 4.14 shows the time results of generating the *NavMesh* with increasing *zFar* starting with the optimal value 2 (any value under 2 would not guarantee that the closest element is found for all notches).

Our goal with this experiment was to test whether the automatic method for calculating the *zFar* dynamically would solve the problem in similar times.

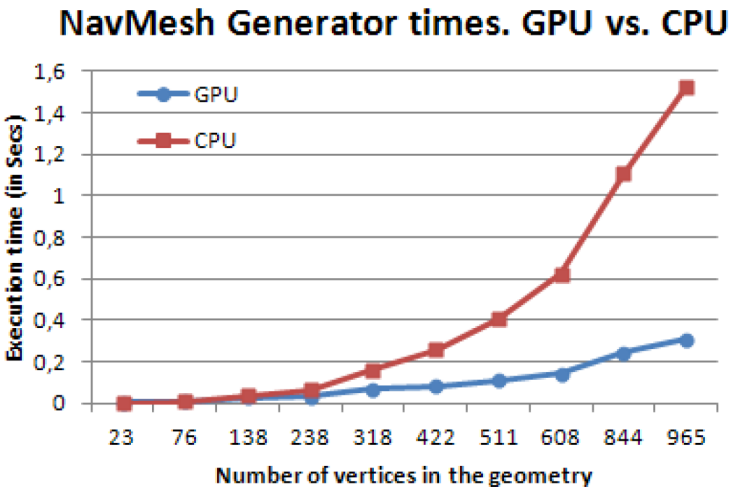

Figure 4.12: Time comparison between the CPU and GPU versions of 10 scenarios with different complexity.

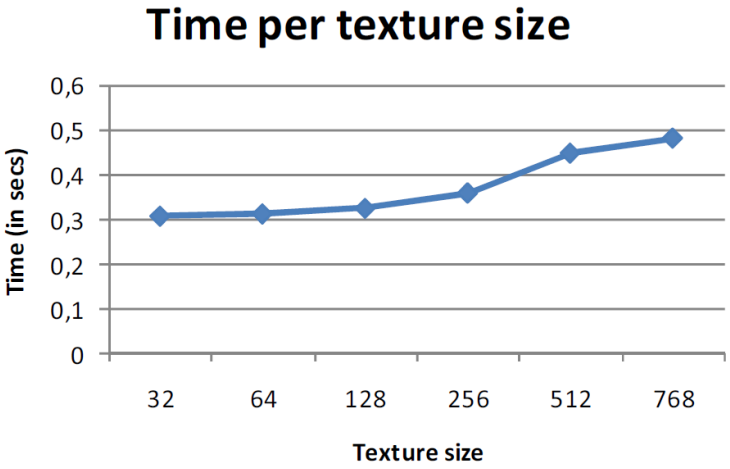

Figure 4.13: Time spend to solve the most complex test environment, for different sizes of texture.

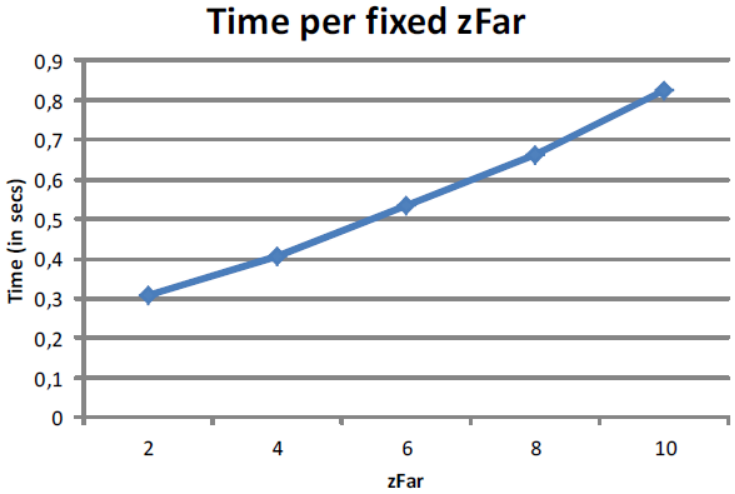

Figure 4.14: Time spend to solve the most complex test environment, for different values of fixed zFar.

Therefore we then tested that same scenario with the method presented in Section 4.2.1. The resulting time was 0,312 seconds, with an average *zFar* of 1,25 which is automatically calculated and changes dynamically when necessary. This shows that our automatic method achieves time results similar to the optimal *zFar* calculated manually.

4.4 Conclusions

In this chapter we have presented a novel GPU based method to automatically compute a *navigation mesh* for a complex 3D scene, representing a single floor plant. Our method has two main steps: first a 2D abstraction is constructed from the 3D model. Then the *NavMesh* is computed using this 2D abstraction.

Additionally, we have presented a GPU version of the same algorithm described in Chapter 3. Results show that the GPU based version is more efficient and scalable than the CPU version, as it makes use of the underlying space partition used by the rendering engine.

The method presented in this chapter works with complex 3D scenes representing a single layer with 3D geometry (meaning that we can handle ramps, steps, holes, etc,). If the original scene consisted of more levels, the user would need to manually subdivide it, treat each level independently and connect its *NavMeshes* manually. The natural extension to this work, that will be explained in the following chapter, is to add the capability to deal with several floors automatically to handle also multi-layered scenes. The advantage of the method

is that it can handle complex 3D geometry, with degenerated triangles, intersecting geometry, and even non-manifolds and automatically recover a simple polygon with holes.

Chapter 5

Computing NavMeshes for multi-layered 3D environments

In the previous chapter, we presented a fully automatic method that takes as an input a 3D scene representing a single floor plan, and abstracts a representation consisting of a 2D simple polygon with holes that fits the input required by the underlying 2D *NavMesh* generator. This chapter further extends the work in order to support 3D multi-layered environments such as buildings, which are basically a concatenation of several floor plans.

The main difficulty when handling multiple layers in complex 3D geometry, is to clearly identify each layer. Our approach to solve this problem consists of creating a voxelization similar to the one done by Recast. But unlike Recast that uses such voxelization to generate the *NavMesh*, we only use the voxelization to split the geometry into layers and then use the voxels corresponding to each layer as a filter to extract an accurate contour of the underlying geometry.

The main contribution of this chapter is a novel GPU based method to generate a CPG for a given 3D scene (with slopes, steps and other obstacles) represented simply as a polygon soup. The algorithm starts by performing a GPU voxelization of the geometry to classify the different layers and calculate a *cutting shape*, *CS*. The *CS* is a depth filter used by the fragment shader to flatten each layers' geometry into a 2D high resolution texture encoding the depth map of each layer. Then each layer is encoded as a single simple polygon with holes which is the required input for the *NavMesh* generator (see Chapters 3 and 4). The *NavMesh* generator provides a convex decomposition with a number of cells close to the optimal value, since for most cases it only needs to create one portal per notch of the polygon with holes. Finally all the layers' CPG are automatically linked together to obtain the final CPG of the entire

scene. Figure 5.1 shows the flow of the algorithm in a very simple but illustrative scenario.

As we will see in the results, among the benefits of this method is the fact that it can still handle complex geometry with intersections, degenerate polygons or holes among others.

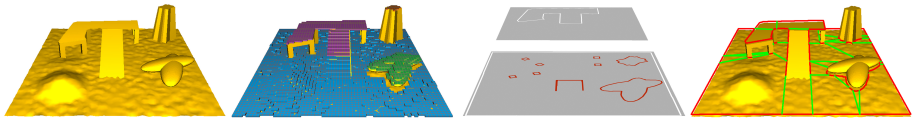

Figure 5.1: Near optimal navigation mesh construction for a simple example of a 3D environment with 2 layers. From left to right we can see the original scene, the result of the layer extraction step after the coarse voxelization, the 2D floor plan of each layer, and finally the near optimal navigation mesh.

To the best of our knowledge this is the first fully automatic architecture that can provide an accurate navigation mesh, with an almost optimal number of cells, ready for path planning from just a polygon soup (with degeneracies such as intersections and holes). Previous work studied is either not fully automatic, not robust to degeneracies or produces an over-segmented CPG. The presented system is efficient enough to allow the scene modeler to make changes and observe the impact on the final navigation mesh at interactive rates and is of high importance to the fields of video games, robotics, movies and character simulation.

5.1 Algorithm Description

Our system entitled *NEOGEN-ML* (the ML sufix comes from *Multi-Layered*), takes as an input a polygon soup describing a multi-layer 3D environment and provides as an output a CPG that can be directly used for character navigation without any manual work required by the user. The user only needs to specify four input parameters:

- maximum step height, h_s, indicates the maximum difference in terrain height that the character can overcome.
- maximum walkable slope angle, α_{max}, indicates the maximum angle of the slope that the character can walk up or down.
- height of the character, h_c.
- walkable seed, s_w, introduced by the user to indicate one point of the geometry where the characters can navigate.

Note that the first two are character navigation skills and thus could be calculated automatically from the set of animation clips available, and the third one can also be automatically calculated.

In figure 5.1 we can see an example with some intermediate results provided by *NEOGEN-ML*, and the final CPG. Figure 5.2 shows the steps followed by the *NEOGEN-ML* framework in order to generate a navigation mesh from a 3D multi-layer environment.

The first step of the presented method consists of performing a GPU *coarse voxelization*. This step classifies the voxels based on whether they are empty, positive, or negative. Geometry does not need to be axis aligned, and the discretization at this level will not affect in any way the final result. This GPU voxelization is then processed by the *Layer Extraction and Labeling* step to classify the voxels into layers.

Once the layers have been detected, the *Layer Refinement* step performs a high resolution orthogonal render of each individual layer in order to obtain a 2D image of the layer that preserves the original geometry. From this 2D image it calculates the floor plan of each layer. Finally a near optimal convex decomposition of each floor plan is carried out and linked to the adjacent layers in order to obtain the CPG of the scene.

In our method, the voxel size is not used directly to define walkable areas, but to serve as a filter to process the underlying geometry. Therefore the discretization at this level does not have an impact on the adjustment of the cells to the geometry. The only thing that matters is not to have an overhanging walkable area over another within the same voxel, which is guaranteed by having the voxel height equal to the characters' height.

5.1.1 GPU coarse voxelization

GPU voxelization is employed to speed up the *navigation mesh* generation which is of great importance for the artist modeling the scene. By achieving interactive rates in this process we greatly help in the time consuming task of creating and modifying scenarios.

The voxelization method is an extension of the GPU method described in [12], where they perform in just one rendering pass a voxelization based on a slicing method. A grid is defined by placing a camera above the scene and adjusting its view frustum to enclose the area to be voxelized. This camera has an associated viewport with (w, h) dimensions. The scene is then rendered, constructing the voxelization in the frame buffer. A pixel (i, j) represents a column in the grid and each voxel within this column is binary encoded using the k^{th} bit of the RGBA value of the pixel. Therefore, the corresponding image represents a $w \times h \times 32$ grid with one bit of information per voxel. This bit indicates whether a primitive passes through a cell or not. The union of voxels corresponding to the k^{th} bit for all pixels defines a slice. Consequently,

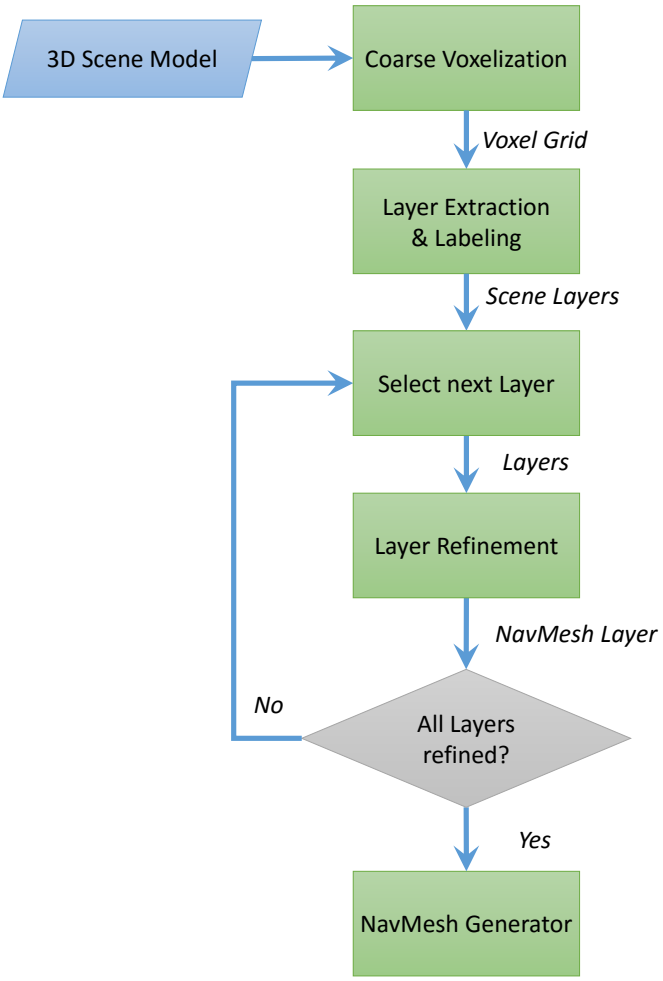

Figure 5.2: Multilayer framework for the automatic navigation mesh generator.

the image/texture encoding the grid is called a *slicemap*. When a primitive is rasterized, a set of fragments are obtained. A fragment shader is used in order to determine the position of the fragment in the column based on its depth. The result is then $OR - ed$ with the current value of the frame buffer.

Since faces near-parallel to the viewing direction do not produce any fragment, we have extended the previous technique by carrying out three separated *slicemaps*, one for each viewing direction (along the X, Y and Z axis respectively). A final *slicemap* is then created by merging these separated *slicemaps* into a single one.

In this step, we need a coarse voxelization that provides information regarding the walkable areas of the scene. Each voxel will be of size $w \times w \times h_s$ where w can be a user input, or else can be automatically initialized as the diameter of the cylinder enclosing a character, and h_s is the maximum step height. Each voxel will be classified into positive, negative or empty, therefore we will use two *slicemaps*, one to store positive fragments and the second one to store negative fragments. This can be done in a single pass using *Multiple Render Target (MRT)* and a shader that outputs each fragment in the corresponding texture depending on its classification.

For each drawn fragment we classify the voxel as:

$$V_{ijk} = \begin{cases} positive & \cos(\alpha_{max}) > (\vec{n} \cdot \vec{UP}) \\ negative & otherwise \end{cases}$$

where α_{max} is the maximum walkable angle, $\vec{n}$ is the normal of the polygon corresponding to that fragment and $\vec{UP}$ is the world up vector that in the coordinates system we use corresponds to $(0, 1, 0)$.

The main problem is that *positive* and *negative* surfaces can fall in the same voxel. Since in our application we are only interested in detecting the voxels with walkable surfaces, we will classify those voxels also as *positive*. The refinement step performed later on will solve the ambiguity. Figure 5.3 shows the decomposition into positive and negative voxels of a simple scene.

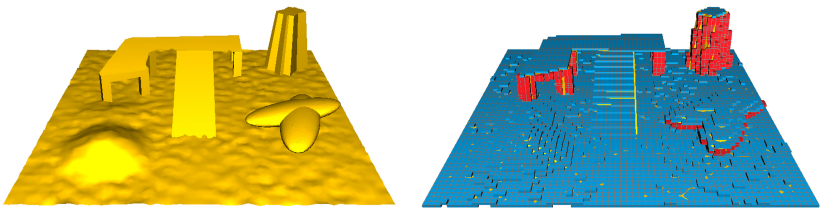

Figure 5.3: Original scene and its corresponding GPU coarse voxelization. Red indicates negative voxels and blue positive voxels (i.e. where the slope is within the walkable capabilities of the character).

Currently the maximum resolution for the coarse voxelization is $128 \times 128 \times 128$. This resolution is limited by having *MRT* that can render into 8 textures in a single pass and we need 2 for each *slicemap* (*positive* and *negative*). Therefore we can count on 4 textures to encode each *slicemap*, with 8 bits/channel and 4 channels (RGB).

The size of the environment that we can currently handle is restricted by the resolution of the voxelization step. However it could easily be extended to larger environments by subdividing the scene into smaller regions to fit with the resolution of the voxelization, treat each of these regions as an individual multi-layered map applying the current method, and finally join the resulting *NavMeshes* into a single one.

5.1.2 Layer extraction and labeling

After the coarse voxelization step, we know which voxels of the scene contain potentially walkable geometry. The potentially walkable area is formed by all those voxels:

$$W_A = \cup\{V_{ijk} = positive\}$$

In this step, we need to split the potentially walkable area W_A into layers, as well as eliminate all those voxels that are unreachable due to the character's maximum step height h_s, or the character's height, h_c. A Layer, L_i, will be composed by a set of *connected accessible* voxels such that it is not possible to have in the same layer V_{ijk} and $V_{ijk'}$ (i.e. it can contain at most one voxel per column of the voxelization). An *accessible* voxel is a *positive* voxel where the character can stand without colliding with any geometry above:

$$V_{ijk} = \begin{cases} accessible & V_{ij(k+\{1..n\})} = empty, n = \left\lceil \frac{h_c}{h_s} \right\rceil \\ non\ accessible & otherwise \end{cases}$$

Two *accessible* voxels V_{ijk} and $V_{ijk'}$, (where $k \leq k'$) are *connected* when the distance in both i and j is at most 1 and $V_{ijk''} = positive, \forall k'' = [k+1, k'-1]$, when $k' - k > 1$.

An ordered flooding algorithm is performed to extract all the different layers and assign them layer IDs, L_{id}. Initially *accessible* voxels are stored ordered from bottom to top and assigned an invalid L_{id}. Starting from the most bottom *accessible* voxel, an L_{id} is created and its *connected accessible* voxels are checked for an L_{id} propagation step, in which we can encounter the following three cases:

- The voxel has an invalid L_{id}: the current L_{id} will be assigned, as long as there is no voxel below it that already has the current L_{id}. Otherwise the voxel will remain with an invalid L_{id} until the flooding method reaches it.

- The voxel has a valid L_{id} different from the current one: A layer merging can be carried out between the two layers if there are no voxels from one layer in the same column as a voxel from the other layer.

- The voxel already has the same L_{id}: in this case nothing needs to be done.

The flooding algorithm proceeds iteratively from bottom to top. Figure 5.4 shows the result of the layer extraction step.

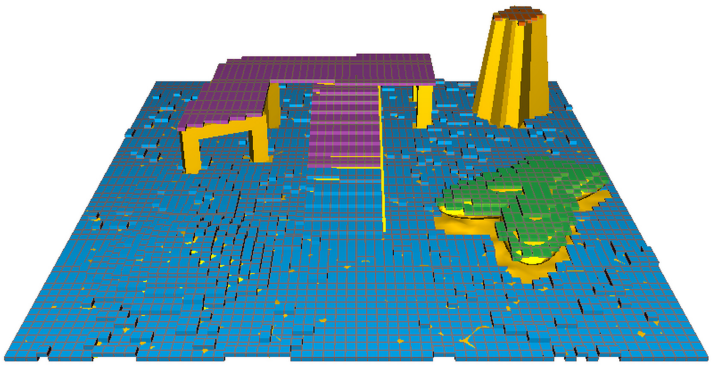

Figure 5.4: Result of the layer extraction step. Each color indicates the set of voxels belonging to the same layer.

Finally, layers that are unreachable for the seed s_w provided by the user are eliminated. Figure5.5 shows the result of the layer connectivity step.

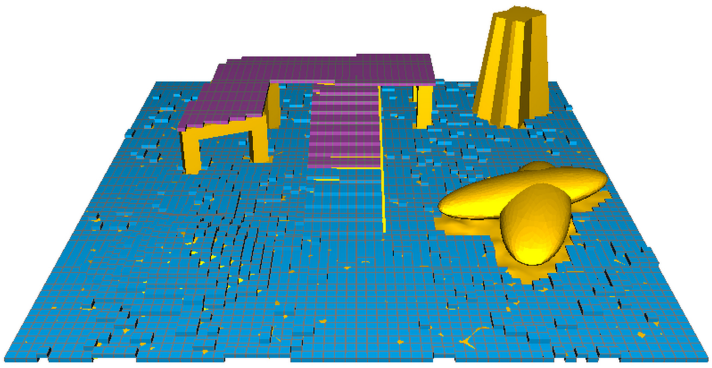

Figure 5.5: After the layer connectivity step, unreachable layers are eliminated (for this example, the orange and green layers are eliminated.

5.1.3 Layer refinement

At this stage, we have subdivided the real walkable space into Layers. The *navigation mesh* could be computed from this representation, but since we want to obtain a fine adjustment to the obstacles, we need to further increase the resolution. The goal is to obtain a 2D high resolution floor plan of each layer. This is a key step of our algorithm, since it will solve any ambiguities contained in the voxels due to the coarse voxelization step, and provide as an output an accurate high resolution description of the original geometry. In order to do this we have implemented a fragment shader that for each layer will only render the geometry that corresponds to such layer. Once the fine floor plan is rendered we can calculate the polygon bordering the layer, as well as the obstacles. We will now explain in detail each of the sub-steps followed by the Layer Refinement process as shown in figure 5.6.

5.1.3.1 Layer contour expansion

For each layer obtained from the coarse voxelization we have two types of voxels: *accessible* and *obstacle*. It is possible though that *obstacle* voxels neighbors of *accessible* voxels partly contain walkable geometry. Contour expansion is thus computed in order to consider these voxels for the cutting shape calculation. Figure 5.7 shows the result of this step around obstacles or floating geometry.

5.1.3.2 Cutting Shape

The *Cutting Shape*, CS, is calculated from the accessible voxels of each layer. The CS can be seen as a shape that wraps each layer in order to filter the geometry that should be rendered into a 2D high resolution texture to obtain the floor plan of each layer.

This CS stores for each pixel the depth of the top of the accessible voxels with the offset h_c and the type of voxel. The output texture stores:

- Channel R: the type of voxel (1 for accessible voxel, 0.5 for voxels which contain a portal between layers, and 0 for non-accessible voxel).
- Channels GBA: the depths of the cutting plane for each column of the voxelization grid.

5.1.3.3 Depth map extraction

An orthogonal top view camera is defined enclosing the scene. The depth map of the layer is calculated with a fragment shader, using the cutting shape as a filter. The fragment shader will discard a fragment (i, j) if it satisfies any of the following conditions:

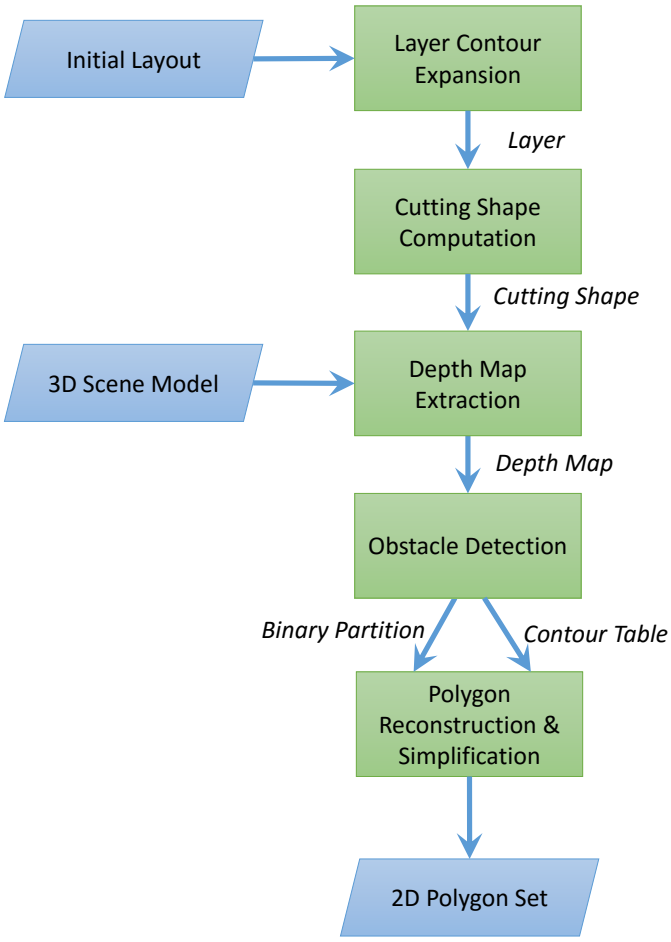

Figure 5.6: Layer refinement diagram.

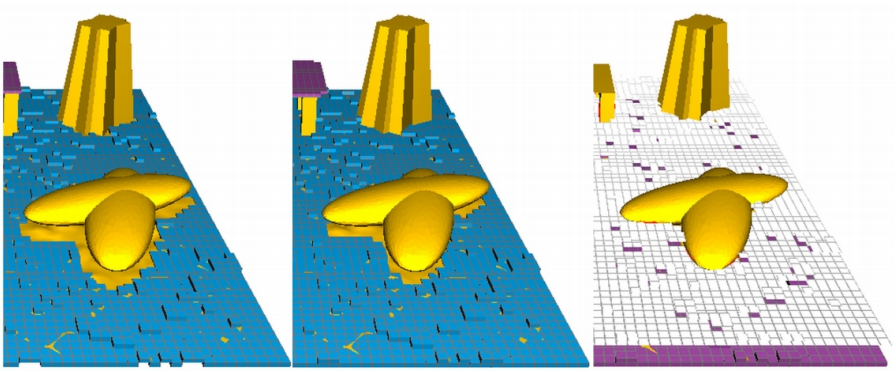

Figure 5.7: From left to right we can see a close up of the scene. On the left, the output of the layer extraction, in the center the result of the voxel expansion step to better capture the original geometry around obstacles, and on the right the calculated *Cutting Shape* (in white).

1. $texture_{ij}^{CS}.R = 0$. If channel R of the cutting shape contains a 0, it means it is a non-accessible voxel.
2. $fragment_{ij}.depth < texture_{ij}^{CS}.GBA$. If the current fragment's depth is smaller than the cutting shape depth (stored in channels GBA) then those fragments do not belong to the geometry of the current layer, but to some other higher layer.
3. $\cos(\alpha_{max}) > (\vec{n} \cdot \vec{UP})$. This means that the current surface, with normal $\vec{n}$ cannot be overcome by the character for the given maximum walkable slope angle α_{max}.

In any other case, the fragment is passed to the next step of the graphics pipeline. Since the CS can intersect with vertical walls, and those walls will not produce any fragment on the rasterization process, we need to deal with near-perpendicular polygons separately in order to not to lose any details of the original geometry. This is done by storing those polygons and then for each layer rendering polygons that intersect with *accessible* voxels directly onto the depth map of the corresponding layer. Figure 5.8 shows the result of this process for the bottom and top layer of the example scene. Notice that black areas indicate non-walkable space (obstacles, or empty) and in grey scale we can see the depth of the walkable areas. Notice how the ramp has been split between the two layers.

5.1.3.4 Obstacle detection and polygon reconstruction

To identify floor vs. obstacles, a flood fill algorithm is performed over the depth map. Obstacles are detected when the difference in height between neighboring

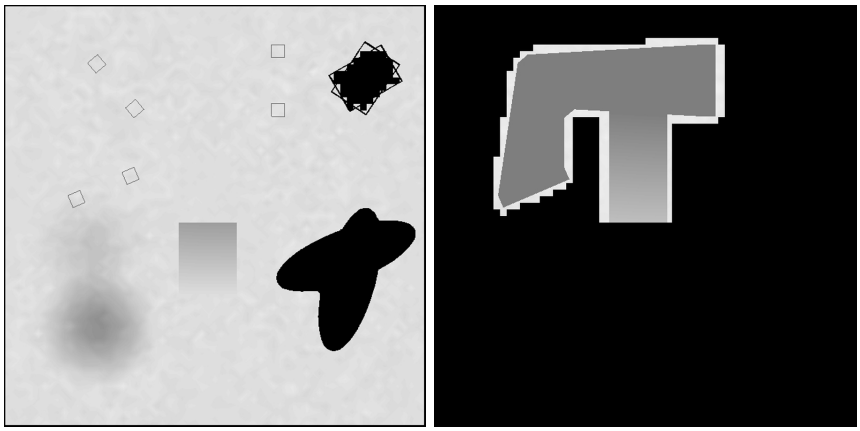

Figure 5.8: Depth maps for the bottom and top layers of the example scene.

pixels is bigger than the character's step height, h_s. The output of this flooding is a binary file where 1 means floor, and 0 means obstacle. Pixels belonging to the contour of an obstacle are considered vertices of the obstacle shape. To reduce the final number of vertices, we apply the Ramer-Douglas-Peucker algorithm [11].

Note that we are reconstructing the shape of each polygon from depth map, therefore the resolution used to generate this map will have an impact on the quality of the reconstructed shapes. Finer resolution will provide a tighter adjustment to the original geometry with a higher computation cost whereas coarser resolution will be faster but provide a lower quality adjustment (i.e. vertices of the polygons may be slightly displaced from the vertices of the original geometry).

5.1.3.5 *NavMesh* Generation

The final step of the algorithm consists of generating a navigation mesh of the scene. The floor plans generated previously are in the input format required by the underlying navigation mesh generator described at Chapter 3: a single polygon with holes representing floor vs. obstacles. The resulting CPG is a near optimal convex subdivision of the space, since the results obtained show that the final number of cells is close to the minimum value of the range where the optimal solution lies (as proven in [63]).

Since our method provides a decomposition that is always guaranteed to be within the optimal bound, and in fact our experimental results show a tendency towards the lower values of this bound, we can determine that our algorithm provides a near-optimal decomposition.

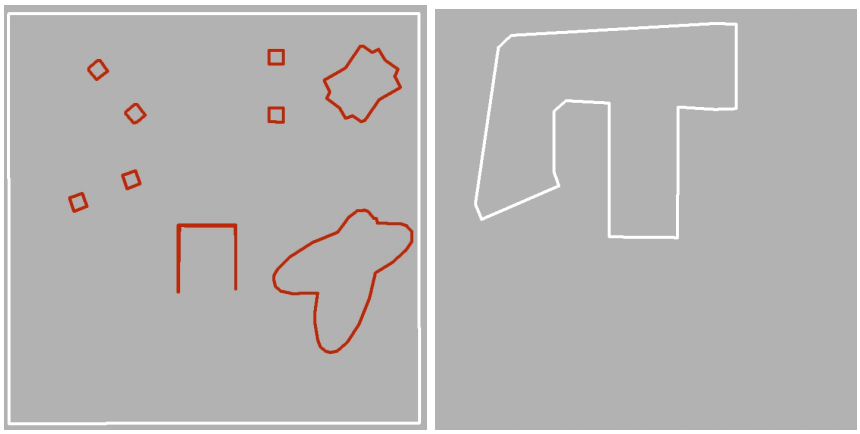

Figure 5.9: Floor plans of bottom and top layers. White polygons provide the shape of each layer, and red polygons represent obstacles.

5.1.3.6 Merging layers

During the layer extraction step, the red channel (R) of the cutting shape texture stores the value 0.5 when a pixel could belong to boundary between layers. This information is used to determine the portals that join a layer with its neighboring layers.

This process consists of for each boundary pixel, find the two closest vertices between neighboring layers and merge them into a single vertex. The position of the merged vertex is calculated as the average between the two positions of the vertices (note that the resolution of the cutting shape may lead to numerical differences in positions and thus the need to calculate the average position). This can have the undesirable effect of adding an extra notch to the map. However this effect can be easily mitigated with the convexity relaxation method.

The algorithm iterates through these possible portal edges to merge them together. After the merging process is finished, the navigation mesh could end up with some non-essential portals. A non-essential portal is defined as a portal that if removed will not leave a notch in the navigation mesh. For example in Figure 5.10 we see that a portal has been created in the center of the ramp to join the bottom and top layers (black dotted line) that could be removed merging the cells of the ramp. Therefore the final step of the algorithm consists of searching for non-essential portals around the merged areas.

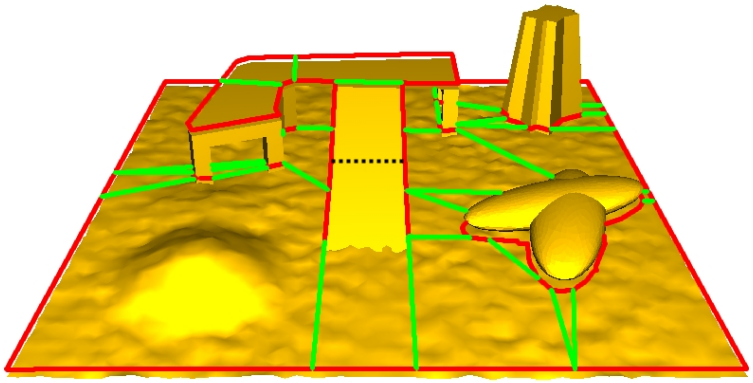

Figure 5.10: The merging step will join the top and bottom Navigation Meshes into one, and will also eliminate the non-essential portals around merged areas (black dotted line).

5.2 Results

We have tested our algorithm in several scenarios of increasing complexity and number of vertices. *NEOGEN-ML* has successfully generated the *navigation mesh* for all the multi-layered 3D environments tested. In table 5.1 we have a summary of some different scenarios in which we have tested our algorithm. Figures 5.11 and 5.12 show the visual results obtained. It is important to mention that our algorithm is robust against intersecting geometry, cracks and holes (which would be treated as obstacles). This is a very important advantage, since it makes easier the task of designing scenarios.

scene	#Fig	#triangles	#layers	#cells	τ_{cr}
map1	1	18,431	2	29	0.5
map2	16	7,308	3	86	0.5
map3	15	19,510	4	50	0.75

Table 5.1: Summary of the scenes tested (with references to the Figure number in the chapter) with the number of triangles and layers for each scene and the final number of cells generated by *NEOGEN-ML* for the given convexity relaxation threshold, τ_{cr}.

In figure 5.13 we show the time taken by *NEOGEN-ML* to output the *NavMesh* for each scene (tested with Intel Core 2 Quad Q9300 @ 2.50GHz, 4GB of RAM and a GeForce 460 GTX). Even though the execution time of the algorithm is not a major goal, it is important that the process run as fast as possible, to make it easier for the designer to make changes to the geometry and see the impact on the resulting navigation mesh at interactive rates.

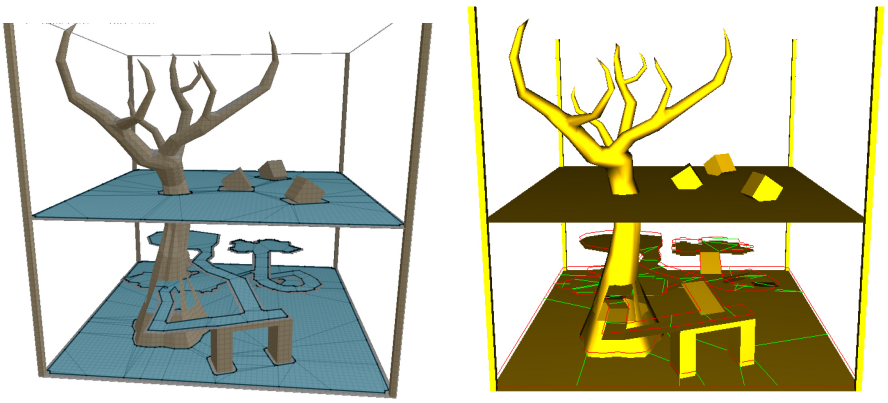

Figure 5.11: Comparison between the results given by Recast (left) and *NEOGEN-ML* (right) in map3. As we can see, our method calculates a navigation mesh with a much lower number of cells. Note also that the shape of obstacles is perfectly adjusted by our cell decomposition (cells rendered slightly above the geometry for clarity), without increasing unnecessarily the number of cells in the final CPG.

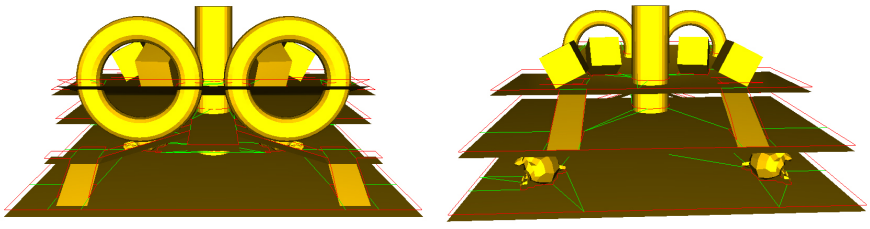

Figure 5.12: Two views of the Navigation Mesh for the scene map2.

We also compare our results against Recast, since it is one of the most widely used open source tools for the computation of *NavMeshes* in complex virtual applications. As we can observe, our method takes considerably less time to compute the Navigation Mesh, and this difference increases with the size and complexity of the environment. Regarding the number of generated cells, for map1 Recast created 121 cells, whereas *NEOGEN-ML* needed only 53 (without convexity relaxation) and can be further reduced to 29 (with $\tau_{cr} = 0.5$).

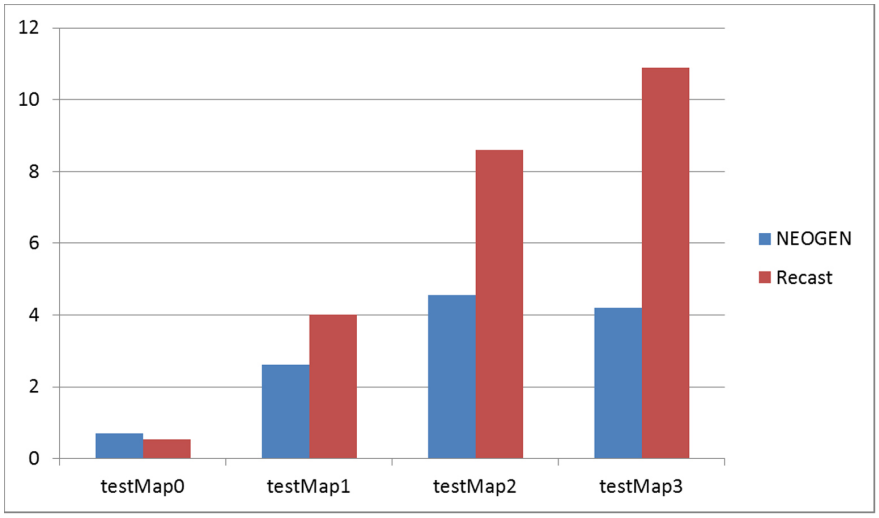

Figure 5.13: Computation time (in seconds) taken for each of the tested scenes.

Another advantage of our method is that it creates cells that adjust tightly to the geometry. Figure 5.14 compares the adjustment to the contour offered by NEOGEN-ML and Recast. As we can observe, *NEOGEN-ML* provides a better adjustment to the contour of the obstacles, and hence, our description of the walkable space is more accurate as can be seen in the contour around the columns (notice that the small visual offset between the cell limits and the contours in our method is due to the cell edges being rendered slightly above the terrain). The voxelization used in Recast provides a coarse approximation of the geometry that can be refined by reducing the size of the voxels, but at the cost of generating significantly more cells (this issue arises regardless of the character's size used to calculate clearance through Minkowski sum). *NEOGEN-ML* offers a tight fit to the geometry without adding unnecessary cells.

Our results show that our system is faster than previous work in the literature, with minimal user input. In fact the information required from the user could be limited to the walkable seed s_w. The rest of the input elements (h_c, h_s, α_{max} and τ_{cr}) could be automatically extracted from the walking abilities of the characters.

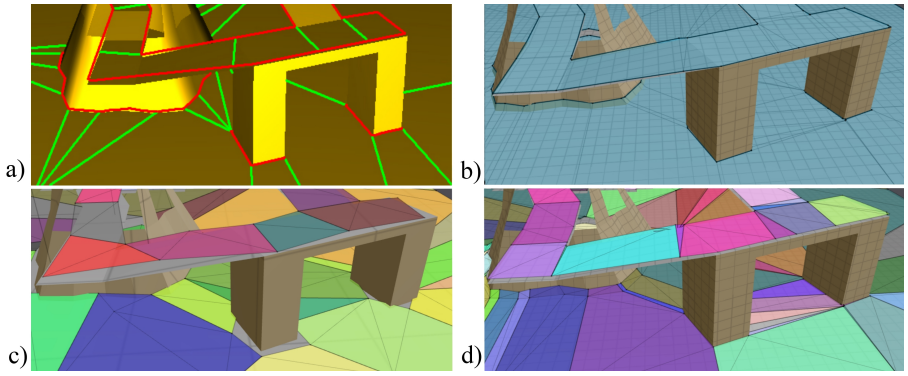

Figure 5.14: Adjustment to the geometry of the *NavMesh* in our method (a) and Recast (b,c,d). Recast results are calculated with agents radius being 0 (to eliminate the object enlargement carried out by Recast to account for clearance). In (b) we can observe a good adjustment to the geometry when the original cell size for the voxelization is 0.1, which has the drawback of creating too many cells (see (d) where cells are shown with different colors), and taking too long to compute. When the cell size is big (0.9), the number of cells generated by Recast drops (c), but at the cost of losing precision in the adjustment to the geometry and even producing intersections (notice the bottom of the columns). In our result (a) we combine small number of cells with tight adjustment (cell borders drawn slightly above the geometry for clarity).

Finally, it is worth to mention that we have participated in the comparative study of Navigation Meshes carried out by Wouter van Toll and Roland Geraerts [96], where *NEOGEN-ML* is compared against other state of the art software such as LCT, ECM, CDG, Recast and a Grid. Results show that *NEOGEN-ML* produces the smallest graph to represent the *navigation mesh* in most of the scenarios being evaluated. This means that our resulting Cell-and-Portal Graph (CPG) contains a lower number of nodes ($|V|$) and edges ($|E|$) compared to the rest of implementations, which can improve the efficiency of pathfinding queries. Note however that some methods achieve speedups by having a dedicated implementation that takes advantage of compact representations. By compact they refer to the size of the information stored per cell. In those cases they benefit from having simple cells consisting of three or four edges. Our work focuses on providing smaller graphs to reduce the cost of path finding independently of the implementation details. Tables 5.2 and 5.3 shows an extract of those results for 2D maps and multi-layered maps respectively (note that a 2D map can be seen as a multi-layered map consisting of a single layer, so *NEOGEN-ML* is still valid on this type of geometry). For a complete set of results and metrics, please see the aforementioned study [96].

Environment	Method	CPG Complexity	
		$\|V\|$	$\|E\|$
Military	LCT	120	134
	ECM	58	72
	CDG	1168	2078
	Recast	101	115
	NEOGEN-ML	52	66
	Grid	36844	72755
University	LCT	732	812
	ECM	329	409
	CDG	3309	4369
	Recast	402	481
	NEOGEN-ML	261	341
	Grid	8363	15460
Zelda	LCT	564	608
	ECM	289	344
	CDG	3579	4233
	Recast	321	376
	NEOGEN-ML	205	260
	Grid	5681	10193
Zelda2x2	LCT	2248	2472
	ECM	1148	1372
	CDG	5636	8850
	Recast	1281	1505
	NEOGEN-ML	820	1044
	Grid	22663	40658
Zelda4x4	LCT	9007	9911
	ECM	4580	5484
	CDG	11996	16564
	Recast	5105	6009
	NEOGEN-ML	3289	4193
	Grid	90591	162537
City	LCT	2553	2732
	ECM	1442	1621
	CDG	3451	5278
	Recast	1527	1706
	NEOGEN-ML	1164	1343
	Grid	207575	408383
Maze8	LCT	26	25
	ECM	30	29
	CDG	68	59
	Recast	14	13
	NEOGEN-ML	12	10
	Grid	31	30
Maze16	LCT	81	80
	ECM	84	83
	CDG	310	319
	Recast	42	41
	NEOGEN-ML	39	36
	Grid	126	124
Maze32	LCT	363	362
	ECM	358	357
	CDG	1084	1138
	Recast	184	183
	NEOGEN-ML	168	157
	Grid	511	510
Maze64	LCT	1422	1421
	ECM	1392	1391
	CDG	4805	4488
	Recast	602	572
	NEOGEN-ML	636	591
	Grid	2045	2041
Maze128	LCT	5674	5673
	ECM	5567	5566
	CDG	25135	44910
	Recast	2590	2480
	NEOGEN-ML	2574	2410
	Grid	8184	8176

Table 5.2: CPG complexity of each evaluated NavMesh generation algorithm on the 2D virtual scenarios.

Environment	Method	CPG Complexity	
		$\|V\|$	$\|E\|$
ParkingLot	ECM	61	68
	CDG	779	873
	Recast	60	66
	NEOGEN-ML	24	31
	Grid	1866	3493
Library	ECM	216	218
	CDG	1758	2173
	Recast	111	113
	NEOGEN-ML	74	76
	Grid	3191	5801
OilRig	ECM	603	629
	CDG	4858	6316
	Recast	324	353
	NEOGEN-ML	253	279
	Grid	75898	147409
Neogen1	ECM	438	444
	CDG	1017	1651
	Recast	103	114
	NEOGEN-ML	193	202
	Grid	4519	8494
Neogen2	ECM	390	403
	CDG	2170	2911
	Recast	198	213
	NEOGEN-ML	295	312
	Grid	9430	17962
Neogen3	ECM	439	439
	CDG	2070	3319
	Recast	275	276
	NEOGEN-ML	218	218
	Grid	9430	17962

Table 5.3: CPG complexity of each evaluated NavMesh generation algorithm on the multi-layered virtual scenarios.

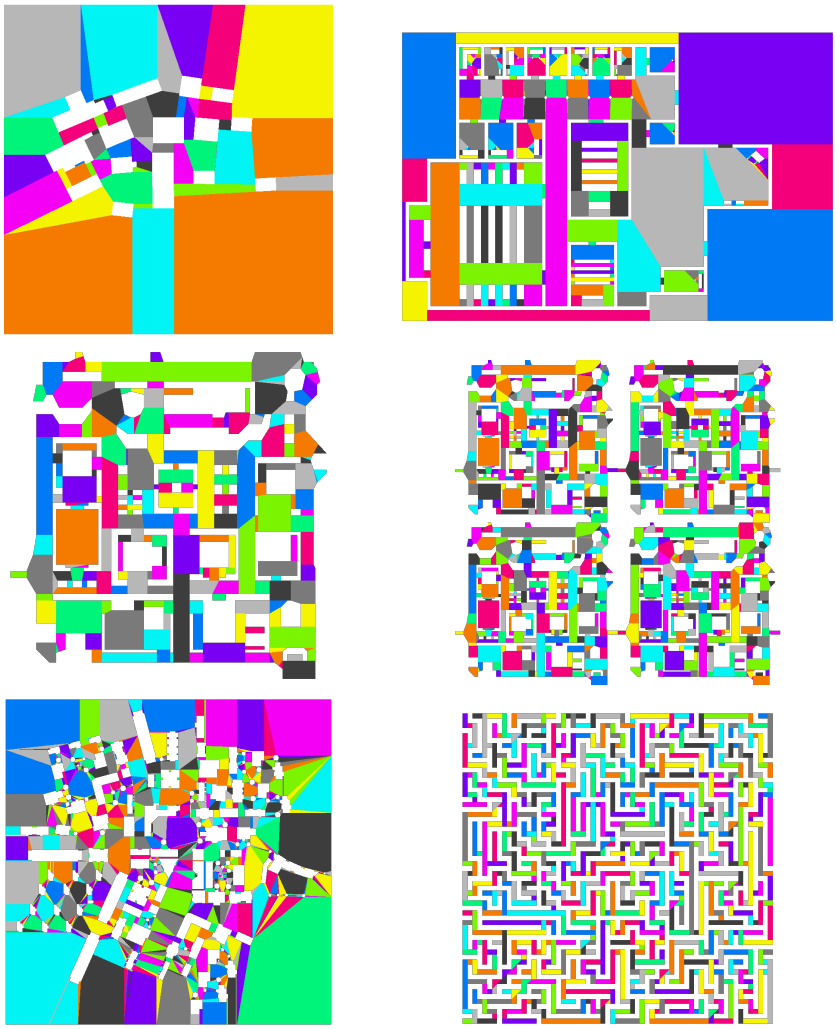

Figure 5.15: The resulting NavMesh using NEOGEN-ML for some of the 2D test maps (Military, University, Zelda, Zelda2x2, City and Maze64).

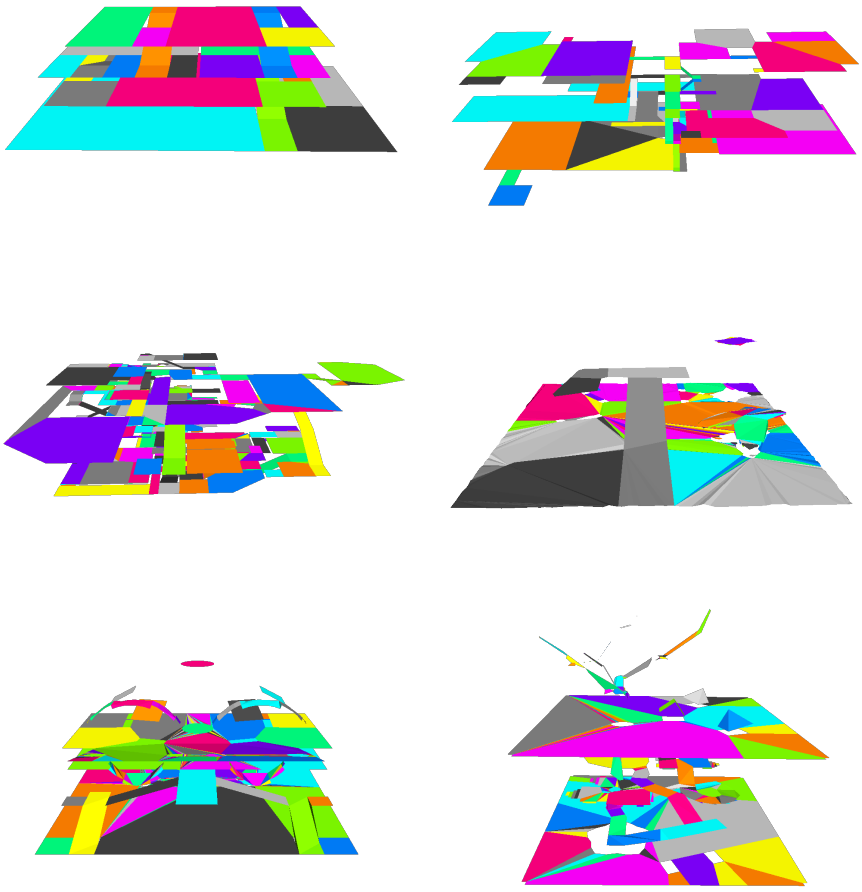

Figure 5.16: The resulting NavMesh using NEOGEN-ML for some of the multi-layered test maps (ParkingLot, Library, OilRig, Neogen1, Neogen2, Neogen3).

5.3 Conclusions

We have presented a novel fully automatic system entitled *NEOGEN-ML*, to compute a Near Optimal convex decomposition from a multi-layered complex 3D environment given as a polygon mesh. Our method can be divided into the following steps: firstly, a coarse voxelization of the scene is done in order to obtain a first approximation of the walkable area. Secondly, the potentially walkable area is subdivided into layers, using an ordered flooding process, and the layers that are not connected with the user seed, s_w, are discarded. Then, each remaining layer is further refined by using the fragment shader at higher resolution and the *NavMesh* is computed. Finally, all those individual *NavMeshes* are merged into a single one, that represents the walkable space of the entire scene.

The results show that *convexity relaxation* is a powerful tool to reduce the final number of cells, especially when the scenario contains many rounded objects, and hence, the resulting CPG fits well with the requirements of an application that needs a real-time response as it keeps a low number of cells, while maintaining an accurate adjustment to the original geometry.

Finally, we have compared our results against a well known tool used in many popular games, Recast, and show how we obtain better results in terms of number of cells, adjustment to obstacles and computational times.

The strongest feature of *NEOGEN-ML* is the fact that it can handle non-perfect geometry, i.e: polygons with collinear edges, intersecting geometry, small gaps due to bad adjustments, etc. This is a useful feature, because usually the available SW to generate virtual environments, introduces this kind of artifacts. Therefore other methods in the literature that require "clean" geometry, need to pre-process or manually fix the geometry before creating the *navigation mesh*. So *NEOGEN-ML* shares this advantage with Recast, but with the additional benefits showed in this paper. However, our system depends on a voxelization step which introduces limitations concerning for example the size of the graph. Additionally the filtering step depends on the resolution and thus on the voxel size. In the following chapter we will start by providing detailed descriptions and figures of where the limitations of our method may arise, and we will use this as the basis for the final contribution of this thesis in the area of navigation meshes.

Chapter 6

Computing NavMeshes for arbitrary 3D environments

In the previous chapter we presented a novel method, entitled *NEOGEN-ML*, to automatically generate *NavMeshes* for any given 3D multi-layered virtual environment represented as a polygon soup. The main advantage of *NEOGEN-ML* is that it provides a tight adjustment to the input geometry while keeping the total number of cells within the bounds of optimality. Moreover, the use of the GPU helps to deal with any type of geometry described as a simple polygon soup that may contain degeneracies, such as cracks, holes and even non-manifold edges in a straight forward way. However, the implementation presented in previous chapter has certain limitations, mainly:

1. The voxel size can have an impact on the classification of the original geometry into layers. If the voxels were too large, there could be scenarios where it would be difficult to clearly classify the voxels into different layers, however a voxel size too small would increase unnecessarily the computational time. We have determined empirically that a good estimate for the voxel size is the character's radius.

2. Our experimental results show that *NEOGEN-ML* works best for building-like environments where there are typically flat floors, stair and ramps. It also works well with outdoor environments as long as the changes in geometry are not very drastic (e.g. too many consecutive bumps and non-smooth surfaces with a variety of heights and layers). An example where we could get a bad partition in layers would be a cave with many holes, stalactites, or stalagmites as the shown in figure 6.1. This particular example would be difficult to capture by our voxelization due to the coarse representation in the Y axis.

3. It is also important to keep in mind that the tight adjustment is only guar-

anteed on the 2D projection of each layer, which means that the polygons reconstructed do not store height information. As we show in Figure 6.2 the cells in the navigation graph are 2D, thus losing the information regarding the bumps in the terrain. However, for the purpose of having characters moving through this navigation meshes, height fields can be used to store the elevation of the terrain.

4. Finally, the resolution of the depth map will have an impact on the adjustment of the reconstructed polygons to the original geometry. In Figures 6.2 and 6.3 we can see examples of the consequences of using a coarse resolution. The figures show how a coarse resolution may introduce extra notches in the polygon reconstruction phase.

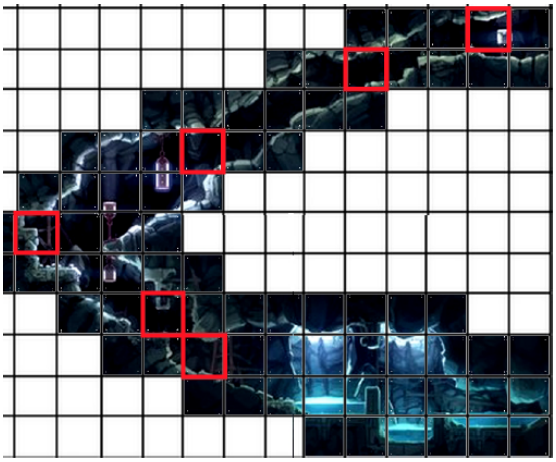

Figure 6.1: Example of complex cave-like environment that would be difficult to process due to the lower resolution used in the Y axis. The grid shows an example of voxelization, where voxels with both ground and ceiling geometry have been highlighted in red.

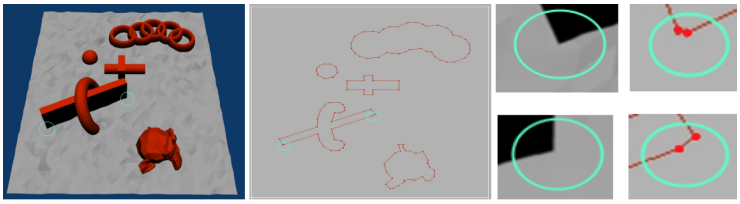

Figure 6.2: Examples of cases where a coarse depth map can lead to extra notches appearing around obstacles during polygon reconstruction. Another undesirable outcome may be polygons that are not an exact adjustment to the original geometry.

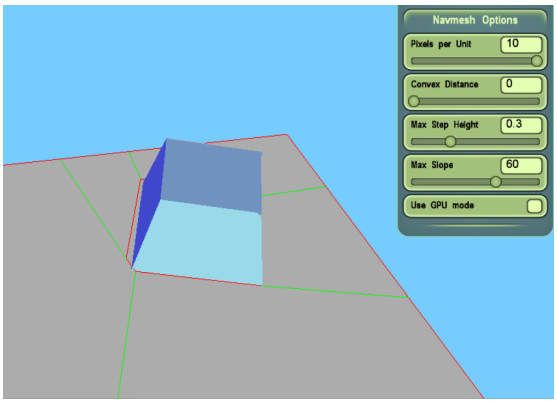

Figure 6.3: Polygon reconstruction over the original geometry, where we can see another example of poor adjustment due to a coarse depth map.

The main contribution of this chapter is a new method and implementation entitled *NEOGEN-3D* that is not restricted only to 3D multi-layered environments but it is able to automatically generate a near optimal *NavMesh* for any given arbitrary 3D environment. The original *NEOGEN-ML* approach, consisted on simplifying the 3D environment to a set of 2D simple polygons with holes, which works as the input for the core *NavMesh* generator algorithm presented in 3. However, this simplification that flattens each layer, implies that we lose valuable information regarding the Y component of the input geometry. The new method and implementation described in this chapter is a novel approach that works always on the 3D space so the input geometry is respected at all stages. Therefore, the resulting *NavMeshes* perfectly adjust to the contour of the obstacles while respecting the bumps present on the terrain. This new approach overcomes the limitations of the original *NEOGEN-ML*, while keeping all its benefits. Although at the time of writing this is work in progress, we have acquired promising preliminary results that deserve to be highlighted.

6.1 Algorithm Description

The process of our algorithm is depicted in figure 6.4. Firstly, we take the input geometry and create a special internal data structure, called the *Terrain*, that contains all the information required for the following steps. Next, a first approach of the walkable space is constructed by determining which faces have a slope low enough for a character to be able to overcome them. This walkable space is further refined by applying the constraints provided by the geometry above it, i.e: when height of the geometry above it is too low for the characters to walk underneath it. This step gives an exact 3D partition into floor and holes that is finally used to compute the *NavMesh* of the whole scene. Note that

unlike the previous *NEOGEN-ML* that processed the input geometry (filtering and projection) to obtain a simplified representation, this new algorithm works directly on the 3D input geometry, thus creating a partition without any kind of simplification. This allows us to obtain a navigation mesh that adjusts exactly to the input geometry, as it respects its original vertices and edges. The tradeoff with this method is that we need to treat every possible degeneracy of the input geometry, which has a cost in terms of the complexity of the implementation.

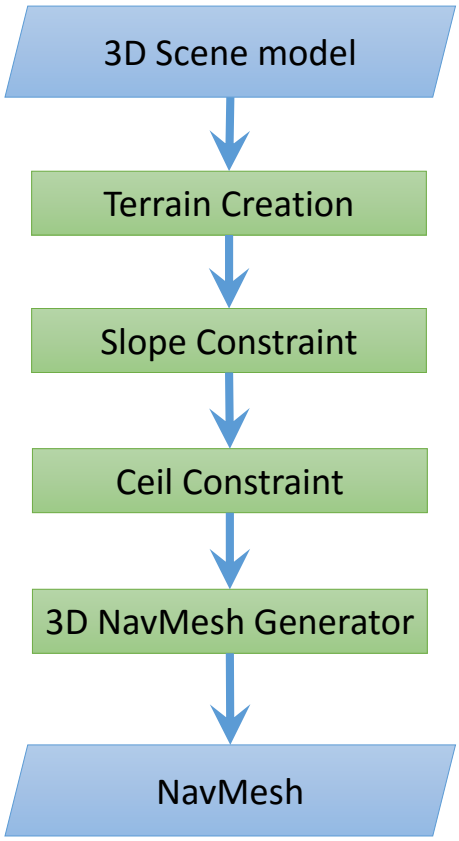

Figure 6.4: *NEOGEN-3D* framework data flow.

6.1.1 Constructing the Terrain

The initial step takes as an input a 3D triangle soup representing the scene an outputs a special data structure, called *Terrain*, that gathers all the information required for the subsequent steps. The *Terrain* contains a set of *Vertices* (3D positions in the space), *Edges* (oriented segments with an start and end *Vertex*)

and *Faces* (ordered lists of *Edges* that lie on the same plane).

We compute the area of each input triangle. If the area equals zero, then we have a degenerated triangle (its vertices are almost collinear) and it is ignored for the purpose of finding walkable surfaces. Otherwise, it is a valid triangle and a new *Face* is created, with the corresponding *Edges* and *Vertices*.

A second step simplifies the Terrain by joining adjacent coplanar *Faces* into a new single *Face*. Two faces are adjacent if they share the same *Edge* (i.e., same endpoints) but in opposite direction (to guarantee that they have their front facing the same direction). By doing this, we improve the computational cost of the following steps since we are reducing the total number of *Faces*. Figure 6.6(a) shows the resulting *Terrain* of a sample scene.

6.1.2 Slope Constraint

In this step we determine which faces have a slope that can be overcome by the walking abilities of the autonomous character. Therefore, each *Face* f of the *Terrain* is classified as:

$$f = \begin{cases} positive & \cos(\alpha_{max}) > (\vec{n} \cdot \vec{UP}) \\ negative & otherwise \end{cases}$$

where α_{max} is the maximum walkable angle, $\vec{n}$ is the normal of the face and $\vec{UP}$ is the world up vector, that by convention we use $(0, 1, 0)$.

This step gives us a first coarse approximation of the walkable and non-walkable areas (see figure 6.6(b)). However, this partition must be further refined in order to take into account the height of the character and the distance between the walkable area and the geometry above it.

We define a walkable Face with the symbol, φ, and represents a face of the terrain with a positive value in the previous equation.

6.1.3 Ceil Constraints

For each *walkable Face* φ_i, we compute its *ceil constraints*, i.e., the set of *Faces* (walkable or not) that are at a distance in the Y direction that is less than or equal to the character's height. Each ceil constraint c_j must be refined in order to determine the exact subpolygon that limits φ_i on the Y direction. To do so, we compute the distance from each of the vertices of c_j along the -Y direction onto the plane defined by φ_i. Then, for each edge of c_j, we can compute the exact subsegment that limits φ_i applying linear interpolation (i.e. calculate the point along the edge with the smallest distance to the plane, so that the character can walk underneath). The resulting subsegments are joined in order to obtain the final subpolygon of c_j that limits φ_i along the Y direction, and

we call this a refined ceil constraint, c'_j. Figure 6.5(left) illustrates this process, from the cell constraint B we compute the refined constraint B'. An Octree space partition is used in order to compute the *ceil constraints* efficiently.

Each refined *ceil constraint* c'_j is projected onto the plane defined by φ_i to calculate $proj(c'_j, \varphi_i)$ and then the boolean difference operation between φ_i and $proj(c'_j, \varphi_i)$ is applied in order to get the exact walkable contour of φ_i as shown in figure 6.5(right). Note that this operation can also create holes if $proj(c'_j, \varphi_i)$ is completely contained in φ_i.

Once all the *walkable faces* have been processed, the result is an exact 3D partition into walkable space (floor) and holes as depicted in figure 6.6(c), which serves as an input to construct the final *NavMesh* of the scene.

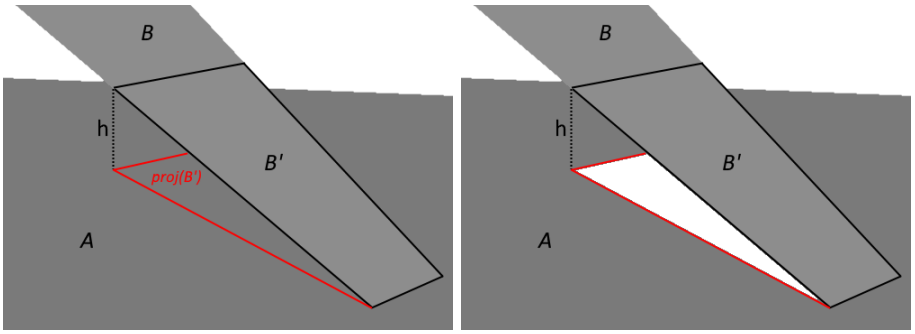

Figure 6.5: Face B is a *ceil constraint* of face A, so B' is the resulting subpolygon of B with all points in the polygon being at a distance along the Y axis smaller than or equal to the character's height h (left). B' is projected onto A and the boolean difference operation $A - proj(B')$ is applied (right).

6.1.4 Constructing the NavMesh

The method used to construct the *NavMesh* is conceptually the same algorithm that we have described in chapter 3, but we have generalized it in order to work in the 3D space. Therefore, there is no need of projecting onto the 2D space and no information is lost in the process. With the extended 3D method presented in this chapter, we obtain an exact navigation mesh in $\mathbb{R}^3$. Cells in this new navigation mesh will consist of a set of one or more adjacent triangles instead of a 2D convex polygon. An interesting feature of the new cells is that their projection onto the XZ plane would be a convex polygon.

In terms of character local motion our new cells in $\mathbb{R}^3$ allow characters to walk in a straight line between any two points located within the cell because the projection of the cell onto XZ is convex, and the changes in the Y direction between adjacent triangles have passed the maximum slope test presented in section 6.1.2.

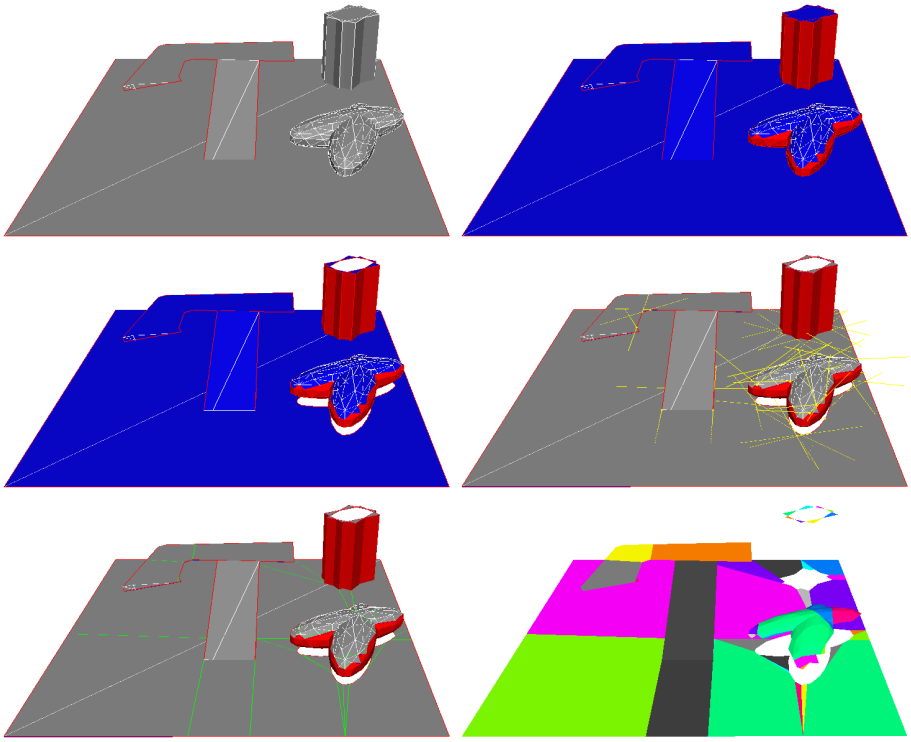

Figure 6.6: *Terrain* construction (a). Original coplanar and adjacent faces have been merged into a single face; the result of applying the *slope constraint* (b). *Walkable faces* are colored in blue, whilst *obstacle faces* are colored in red; the exact floor and holes partition after applying the *ceil constraints* (c); the resulting notches with its *Area of Interest* (d). *Convexity relaxation* is set to 0.5; the portals (in green) resulting after breaking the previously detected notches (e); the final *NavigationMesh*, where each color identifies a single cell (f).

6.1.4.1 Creating the portals

For each vertex of the partition, we determine if it is a notch by checking the inner angle α_i. Since we are working in $\mathbb{R}^3$, we need to define the internal angle which we calculate in $\mathbb{R}^2$.

Given edges $e_{i-1,i}$ and $e_{i,i+1}$, we compute the planes $\Pi_{i-1,i}$ and $\Pi_{i,i+1}$ as the planes that are perpendicular to the XZ plane, where $\Pi_{i-1,i}$ contains $e_{i-1,i}$ and $\Pi_{i,i+1}$ contains $e_{i,i+1}$. The internal angle α_i is the angle between the planes $\Pi_{i-1,i}$ and $\Pi_{i,i+1}$ on the side of the walkable surface.

If $\alpha_i > \pi$ then we have a notch, v_i and a portal needs to be created in order to turn it into two convex vertices. The *Area of Interest* of a notch is the virtual area delimited by the semiplanes supported by the edges incident to the notch that causes the concavity ($\Pi_{i-1,i}$ and $\Pi_{i,i+1}$). Notice that if we create a portal with any element inside this area, we automatically break the concavity (definition and proof similar to the one given in section 3.2). The *convexity relaxation* concept (see Section 3.3) is also applied in order to discard some notches. Figure 6.6(d) shows the notches with $\tau_{cr} = 0.5$.

For each notch v_i, we look for the closest edge in the list of possible candidate edges, ε, that lies inside its *Area of Interest*, I_i. Those edges are the ones defining the faces φ sharing the notch plus the edges of the holes contained in those faces. Note that we must discard those edges at a distance 0 from the notch, i.e., the incident edges to the notch.

The break notch function proceeds differently depending on four possible cases:

1. The closest element is an endpoint, v_j of an obstacle edge of the partition. In this case, we create a *vertex-vertex* portal and if the other vertex is also a notch, we check whether after creating the portal, the notch condition is broken in this second notch so that it will not be necessary to process it.

2. The closest element is a point in between the endpoints of an obstacle edge of the partition. In this case, we create a *vertex-edge* portal by subdividing the edge at the closest point to the notch (the orthogonal projection of the notch over the obstacle edge, which implies creating a Steiner boundary point, s_i).

3. The closest element is a previously created portal, p_k. In this case, we create a *vertex-vertex* portal. This case has to be treated differently depending on the location of the previous portal endpoints, $p_k[0]$ and $p_k[1]$ relative to the *Area of Interest*, I_i of the notch v_i. Independently of which of the following cases applies, after breaking the notch we need to evaluate whereas p_k is still an essential portal.

   (a) First, we check if any of the endpoints of the portal is inside the area

of interest of the notch:
$(p_k[0] \in I_i) or (p_k[1] \in I_i)$
In such case, it is safe to create a portal, p_i between the notch, v_i and the endpoint lying inside I_i. This will break the notch.

(b) If both endpoints lie inside the *Area of Interest*:
$(p_k[0] \in I_i) \& (p_k[1] \in I_i)$
we choose the closest one to v_i to create the new portal p_i.

(c) If both endpoints are outside the *Area of Interest*:
$(p_k[0] \notin I_i) \& (p_k[1] \notin I_i)$
we split the notch v_i into two notches v_{i_1} and v_{i_2}, sharing the same position than the original notch. The *Area of interest*, I_{i_1}, of v_{i_1} is defined by the supporting ray $p_k[0] - v_i$ and the closest ray limiting the previous *Area or interest*, I_i. Similarly, the area of interest, I_{i_2}, of v_{i_2} is defined by the ray $p_k[1] - v_i$ and the closest ray limiting the previous *Area or interest*, I_i. See figure 6.7 to better illustrate this case. The break notch function is called for both notches v_{i_1} and v_{i_2}.

4. The closest element is a walkable edge that connects to another walkable face. In this case, we repeat the algorithm by selecting the edges of this face (and its holes) as possible candidates until case 1, 2 or 3 is reached.

   Note that by doing this, we are using the floor and holes partition as a space partition itself, so we are overcoming the main drawback of the original method explained in Chapter 3, as the cost of computing the closest element is no longer n^2.

Whenever a portal is created, all the faces intersected by the portal are split into two new faces, as each portal will delimit two adjacent *Cells*. Figure 6.6(e) depicts the resulting portals. Algorithm 4 contains the pseudocode for the portal creation of *NEOGEN-3D*. Note that the functions $createPortalVertexVertex()$ and $createPortalVertexEdge()$ have not been included since they are straight forward.

Algorithm 4 Portal creation algorithm for *NEOGEN-3D*

```
1: procedure PORTALCREATION(Notch v_i, EdgeList ε)
2:     c ← computeClosestElementInAOI(v_i, ε)
3:     if c is vertex then
4:         createPortalVertexVertex(v_i, c)
5:     else if c is EdgeObstacle then
6:         createPortalVertexEdge(v_i, c)
7:     else if c is EdgePortal then
8:         createPortalVertexPortal(v_i, c, ε)
9:     else
10:                                                  ▷ c is EdgeWalkable
11:        ε ← c.getOtherFace().getEdges()
12:        portalCreation(v_i, ε)
```

Algorithm 5 Portal Vertex-Portal algorithm

```
1: procedure CREATEPORTALVERTEXPORTAL(Notch v_i, Portal p_k, EdgeList
   ε)
2:     if isInAOI(v_i, p_k[0]) & isInAOI(v_i, p_k[1]) then
3:         q ← closestPoint(v_i, p_k[0], p_k[1])
4:         createPortalVertexVertex(v_i, q)
5:     else if isInAOI(v_i, p_k[0]) then
6:         createPortalVertexVertex(v_i, p_k[0])
7:     else if isInAOI(v_i, p_k[1]) then
8:         createPortalVertexVertex(v_i, p_k[1])
9:     else
10:            ▷ Create subnotch v_{i_1} and call to main portalCreation procedure
11:        r_1 ← p_k[0] − v_i
12:        n_1 ← createNotch(v_i, r_1, getClosestLimit(r_1, I_i))
13:        portalCreation(v_{i_1}, ε)
14:            ▷ Create subnotch v_{i_2} and call to main portalCreation procedure
15:        r_2 ← p_k[1] − v_i
16:        n_2 ← createNotch(v_i, r_2, getClosestLimit(r_2, I_i))
17:        portalCreation(v_{i_2}, ε)
18:    if isNonEssentialPortal(p_k) then
19:        removePortal(p_k)
```

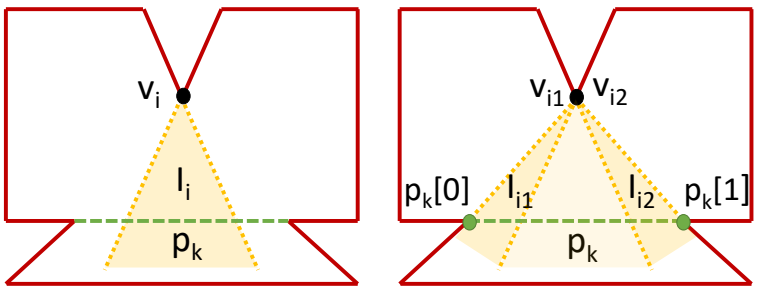

Figure 6.7: The closest element of the notch v_i is a previously created portal p_k (left). In order to avoid *T-joints*, we split the original notch v_i into two virtual notches v_{i_1} and v_{i_2}, sharing the same position as v_i, but the *Area of Interest* of each notch is defined by the endpoints of p_k (right).

6.1.4.2 Creating the Cells

The last step consists of grouping the resulting walkable faces into *Cells*. Start with the first face that has not been assigned to a *Cell* yet, the algorithm proceeds recursively adding all the adjacent walkable faces that are separated by a current walkable edge (non-portal). The process is repeated until all the walkable faces have a *Cell* assigned. Figure 6.6(f) shows the resulting cells in a simple environment.

Note how the resulting cells perfectly match the variations of the *Terrain* and are also perfectly adjusted to the obstacles (as they are built based on the original polygons of the input geometry).

6.2 Results

NEOGEN-3D is work in progress that we expect to have finished soon, so this section contains only a discussion and some preliminary qualitative results. We do not show other metrics such as time performance nor memory efficiency, as we are currently focusing on obtaining a robust implementation against all possible geometry that we could encounter. Once all degeneracies are correctly handled we will polish the implementation to further speed up the method. Therefore in this current chapter we have not provided quantitative metrics as they will not be relevant until the implementation is fully finished. However as

indicated in this chapter, we have reduced the cost of the algorithm greatly by using the connectivity described by the input geometry itself as a way to achieve significant speedups at the step of searching for closest elements in the geometry. In terms of the relationship between number of notches and resulting cells, the results are similar to the ones presented in previous *NEOGEN* versions as the core ideas in terms of partitioning the geometry, still hold in *NEOGEN-3D*.

Our current implementation has proven to be robust against intersecting geometry and it can also deal with degenerated input polygons (i.e., models containing triangles where some vertices are almost collinear). An example of geometry with intersecting geometry can be observed in the example scenario used to highlight each *NEOGEN-3D* step (figure 6.6). Notice that the obstacle at the top-right corner is in fact formed by two intersecting cubes. Moreover, the cross-like obstacle is modeled as two separated and intersecting meshes, as well as the ramp is intersecting also with the floor. *NEOGEN-3D* is able to deal with those degeneracies and correctly generates the *NavMesh* for this type of scenarios. However, it is currently restricted to manifold geometry. We are focusing our efforts on handling also non-manifold geometry, thus producing a truly general system. Note how unlike *NEOGEN-ML* where we could filter out degeneracies of the input geometry (even non-manifolds) in a straight forward way by using the GPU, our new *NEOGEN-3D* works directly over the input geometry (what is known as an exact method). This leads to a more complicated implementation where all degeneracies need to be carefully treated to obtain a robust code.

Finally, We are also preparing a paper for submission that will include the final results of *NEOGEN-3D* with a robust version of the code. In figure 6.6(f) we have already shown the resulting *NavMesh* for a simple scenario. This figure already allows us to show the differences in the final *NavMesh*. With *NEOGEN-3D* the resulting cells adjust to the geometry also along the Y axis, thus achieving a complete 3D adjustment (see Figure 6.8 for a comparison between *NEOGEN-ML* and *NEOGEN-3D*). The previous *NEOGEN-ML* system presented earlier in this Dissertation, could only guarantee tight adjustment in 2D (along X and Z axis) since each floor plan was computed by a projection method onto 2D planes. Although the 2D level of adjustment is enough for most scenarios, as we have shown through a large number of scenarios, *NEOGEN-3D* presents yet one more dimension of adjustment which will make it suitable for a much larger number of scenarios. Figure 6.9 shows the results of using *NEOGEN-3D* on a floor plan represented as a polygon with holes, a multi-layered environment and a general 3D map, thus demonstrating that it is our most general approach.

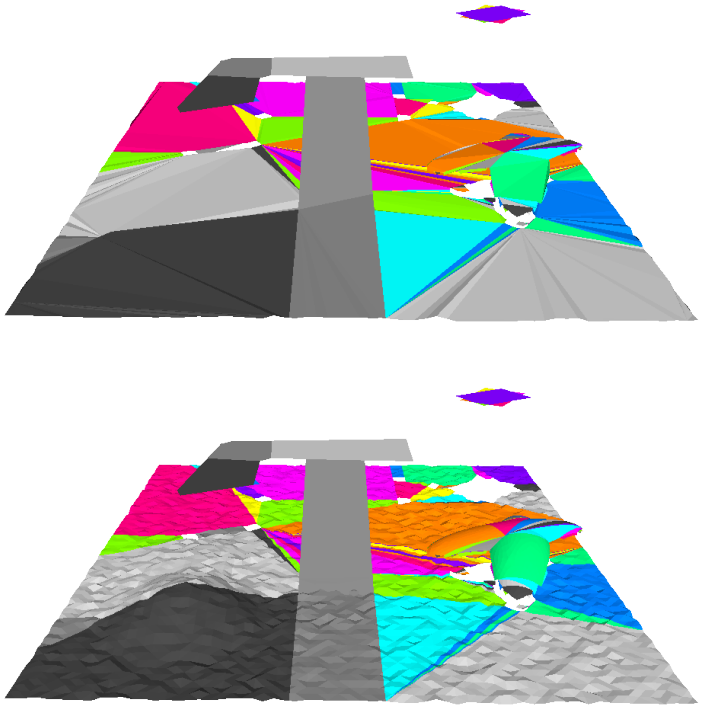

Figure 6.8: The resulting NavMesh using *NEOGEN-ML* (top) and *NEOGEN-3D* (bottom) on the same map. Notice how the later one offers a more rich and detailed NavMesh, perfectly adjusting also on Y and respecting the bumps of the terrain.

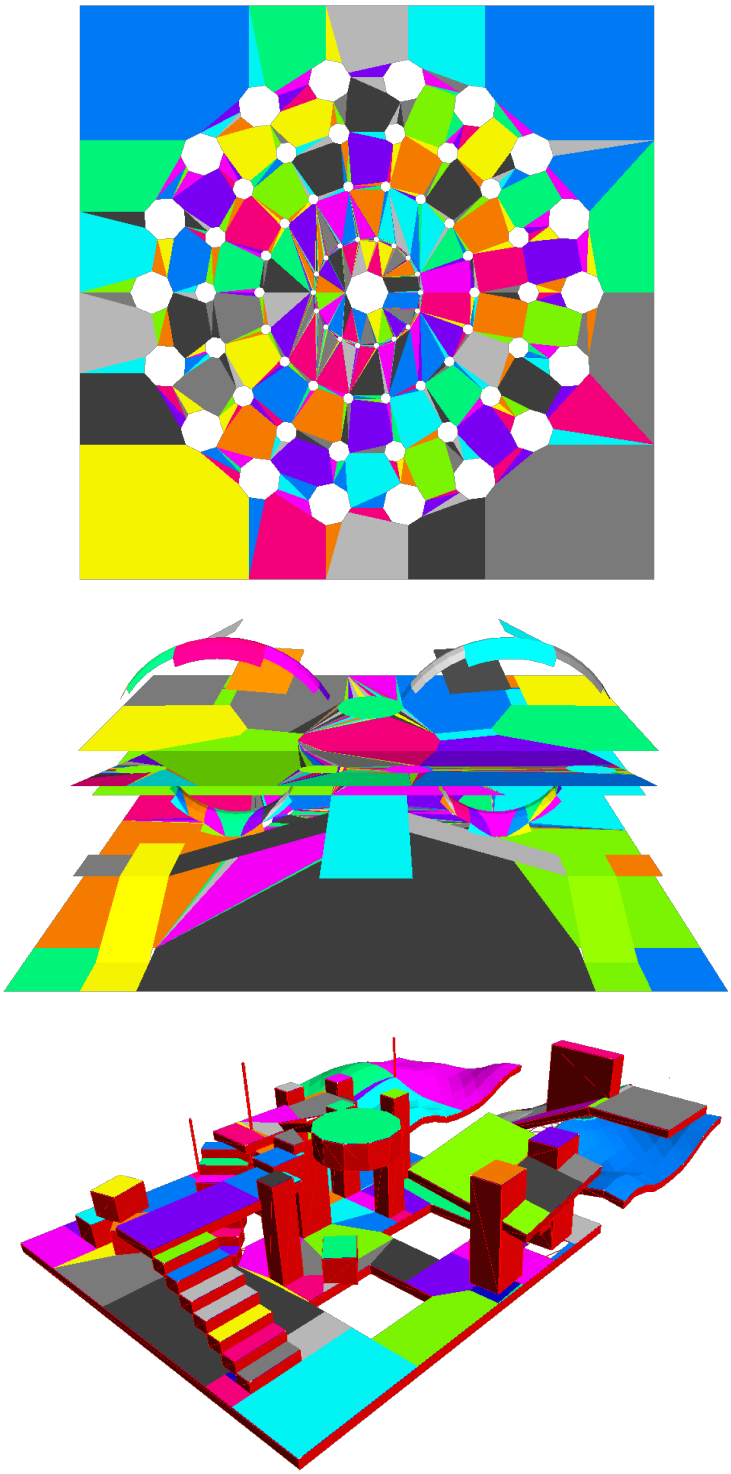

Figure 6.9: The resulting *NavMesh* for a 2D simple polygon with holes (top), a multi-layered environment (center) and a general 3D scene (bottom) using *NEOGEN-3D*.

6.3 Conclusions

In this chapter we have presented *NEOGEN-3D*, a new algorithm with also a completely new implementation that overcomes all the limitations of the previous *NEOGEN-ML* algorithm described in Chapter 5. This novel approach is able to automatically compute a *Navigation Mesh* for any 3D environment with arbitrary topology that may contain degeneracies such as intersecting geometry.

Our new approach has been developed from scratch to avoid the main limitations due to the voxelization and projections steps, while keeping all the benefits of the previous version (such as: near-optimal partition, convexity relaxation and tight adjustment to geometry). The *NavMesh* generated with the new algorithm contain a near-optimal number of cells, avoids the creation of degenerated cells and the *convexity relaxation* concept allows us to further reduce the number of cells. In addition, the face connectivity described by the 3D model is used as a space partition when computing the closest element to a notch, improving the cost of the algorithm by an order of magnitude.

Contrary to most of the reviewed previous work, our method ensures that the created *Navigation Mesh* perfectly follows the contour of the obstacles. Furthermore, the shape of the cells also adjusts perfectly to the terrain and maintains its bumps and variations. This is useful in order to avoid problems when locating the cell where a character stands. Therefore, our data structure can be used not only for path planning, but also for those methods that needs a precise definition of the walkable space, for example, to solve the problem of setting the feet of the characters in the ground.

Chapter 7

Computing Exact Arbitrary Clearance for NavMeshes

In Chapters from 3 to 6 we have described our contributions in the field of *navigation meshes*, by developing algorithms to automatically create data structures encoding the free space of the scene by splitting it into convex polygons, known as cells. A Cell-and-Portal Graph (CPG) is obtained where a node represents a cell of the partition and a portal is an edge of the graph that connects two adjacent cells. Then, given a start and a goal position, paths can be calculated through a variant of the classic A^* algorithm. Finally, at every step of the simulation, a *local movement* algorithm is applied in order to guide the agent through the obtained path by computing intermediate goal positions (commonly known as *way points*) that connect the different nodes of the path. In this chapter we describe our novel *local movement* algorithm, which takes into account clearance and makes a dynamic *way point* assignation for each character. We also extend the *NavMesh* with clearance information to adapt global navigation to the dimensions of the characters.

There are two frequent artifacts in crowd simulation caused by *navigation mesh* design. The first appears when all agents attempt to traverse the *navigation mesh* and share the same way points through portals, thus increasing the probability of collisions with other agents or queues forming around portals. The second is caused by way points being assigned at locations where clearance is not guaranteed, which causes the agents to either walk too close to the static geometry, slide along walls or get stuck. To overcome these issues we use the full length of the portal and propose a novel method to calculate way points dynamically.

This chapter presents a novel system to guarantee character trajectories with clearance that make the most of the available free space in the *NavMesh*. We present three contributions. The first allows computing paths with any desired

amount of clearance for cells of any shape (even with concavities). Secondly, we propose a new method for computing the exact region of a portal that guarantees the desired clearance condition. Finally, way points are dynamically assigned for each character over the portal with clearance, based on current trajectory and destination, therefore guaranteeing that agents in a crowd will have different way points assigned over the same portal.

7.1 Clearance Value of a Cell

When simulating a variety of characters, it is convenient to be able to calculate the shortest route for the characters based on their size. If we think of applications such as video games, this would allow a skinny character to escape from a large monster by running through a narrow passage. The algorithm implemented must also be efficient, as for a large scenario the paths for all characters need to be calculated within a small fraction of a second. In order to be able to calculate paths with clearance in real time it is necessary to pre-compute and store clearance values per cell. This can be a difficult problem to tackle especially if the *NavMesh* allows for small concavities in the cells as in our case. We propose an algorithm to compute per cell clearance that works for any given polygon, without the need of being strictly convex.

Given a cell $\mathcal{C}$, we define a cell cross as the pair ($\mathcal{P}_1$, $\mathcal{P}_2$) of $\mathcal{C}$, where $\mathcal{P}_1$ is the entry portal and $\mathcal{P}_2$ is the exit portal. We classify the obstacle edges of the cell into edges to the left (*stringLeft*) and edges to the right (*stringRight*) in respect to the path that crosses the cell from the entry portal to the exit portal (see Figure 7.1). Note that it is not necessary to have strictly convex cells, as cells generated by *NEOGEN* are allowed to have certain concavities depending on the *convexity relaxation* threshold chosen when creating the mesh (see Section 3.3).

The algorithm examines every portal endpoint and notch (i.e., a vertex such that its internal angle is greater than π) present in *stringLeft* and determines the closest edge in *stringRight*. The distance between the notch and the closest edge is the clearance value of this notch. Note that in this case, the endpoints of each string must be treated as if they were notches. If the closest edge to the notch is a portal edge, the algorithm recursively checks the distance between the notch and the edges lying in the adjacent cell through the portal. The clearance value of the left string cl_L is the minimum of those distances. To compute the clearance value of the right string cl_R, we proceed in the same way. Finally, the clearance value of the described path is computed as follows:

$$cl\left(\mathcal{P}_1, \mathcal{P}_2\right) = min\left(cl_L, cl_R\right) \tag{7.1}$$

It is only necessary to check the distance of the notches of the string against the edges of the opposite string, as in the case of a convex vertex, the distance

to the opposite string must be greater than or equal to the clearance value of the cell. This process is done off-line once the *NavMesh* of the virtual scenario has been generated and, for each cell, we store in a table the clearance value of every possible cell cross. This is because it is possible to have a cell with three or more portals, where an agent with a large radius can walk for example from portal $\mathcal{P}_1$ to $\mathcal{P}_2$, but not from portal $\mathcal{P}_1$ to $\mathcal{P}_3$ (see Figure 7.2).

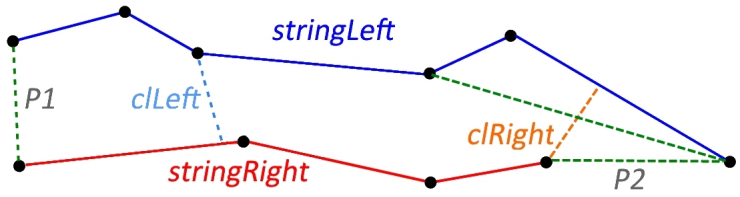

Figure 7.1: Clearance calculation for a given cell.

Algorithm 6 Clearance computation for a given Cell and Cross

```
1: function computeClearanceCrossCell(Cell c, Portal p1, Portal p2)
2:     stringLeft ← computeVertexStringLeft(p1, p2)
3:     stringRight ← computeVertexStringRight(p1, p2)
4:     cl_L ← computeClearanceStringString(stringLeft, stringRight)
5:     cl_R ← computeClearanceStringString(stringRight, stringLeft)
6:     return min(cl_L, cl_R, p1.length, p2.length)
```

Algorithm 7 Vertex string to Vertex string clearance computation

```
1: function computeClearanceStringString(Vertex[] V1, Vertex[] V2)
Ensure: V2.length ≥ 2
2:     c ← ∞
3:     for all v_i ∈ V1 do
4:         if isNotch(v_i) then
5:             for j ← 0 ... V2.length − 2 do
6:                 tmp ← pointToSegmentDistance(v_i, V2[j], V2[j + 1])
7:                 if tmp < c then
8:                     c ← tmp
9:     return c
```

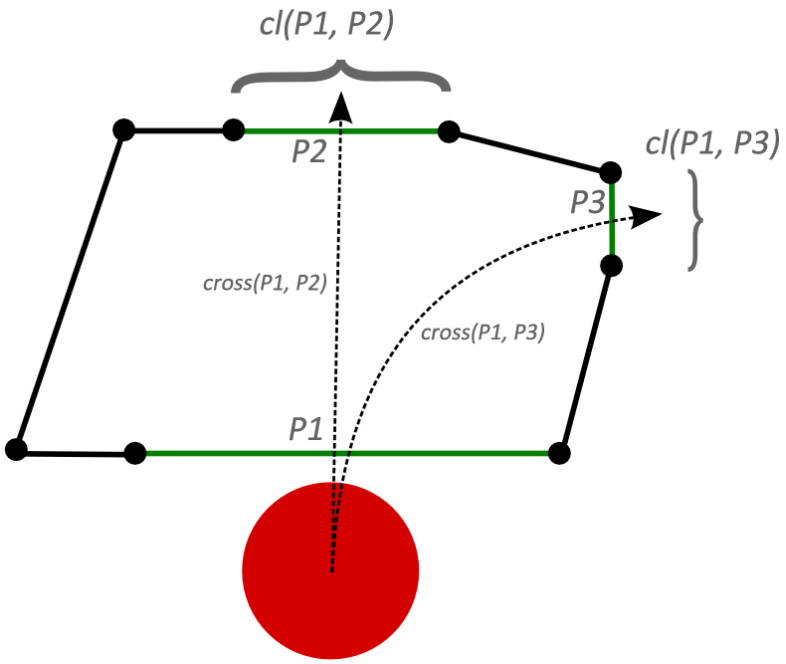

Figure 7.2: Example of different clearance depending on the crossing path through a cell.

7.2 Finding Portals with Enough Clearance

In order to avoid artifacts such as characters bouncing, sliding or getting stuck on the edges of the geometry, we should only assign way points that have enough clearance (i.e. that they have the required distance from the static geometry for the character to traverse the portal without collision). Note that fixing way points to the center of the cell does not always guarantee collision free traversals, as shown in Figure 7.3.

As we discussed in the related work chapter, there has been a large amount of work on computing shortest paths with clearance or paths for disks of arbitrary radius. However our goal was not to compute one single crossing point over portals that guarantees the shortest paths with clearance, but instead computing the sub-segment of possible crossing points. By doing this, we allow the local movement algorithm to have the flexibility of assigning any crossing point over the sub-segment with the guarantee that it will be collision-free.

Let $\mathcal{C}_A$ be the cell where the character is currently located, $\mathcal{C}_B$ be the next cell in the path and $\mathcal{P}$ the portal that joins both cells. We want to calculate the sub-segment $\mathcal{P}'$ of $\mathcal{P}$ such that all points in $\mathcal{P}'$ have enough clearance.

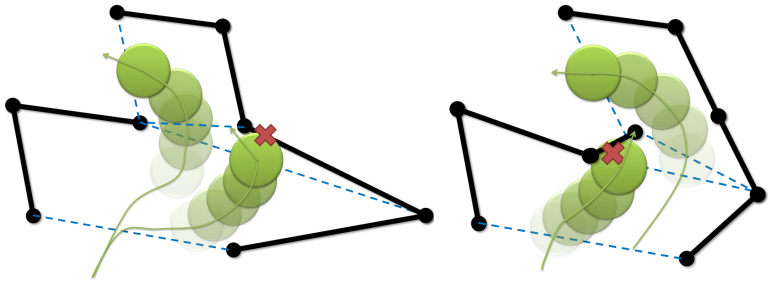

Figure 7.3: Examples of fixed way points that cause collisions. On the left, the way point is fixed at the center of a cell which causes a collision with the geometry. On the right the way point is assigned at a distance r of the portal endpoint also causing collision.

The algorithm for finding portals with enough clearance proceeds by reducing the size of the original portals based on the following three cases:

1. Limitations given by the endpoints of the current portal.

2. Limitations given by the endpoints of the portals that must be crossed to go from the current cell to the target cell in the path (usually portals in either $\mathcal{C}_A$, $\mathcal{C}_B$ or their neigboring cells).

3. Limitation given by obstacle edges of the adjacent cells (or neighbors).

Cases 1 and 2 assume that the endpoints of portals are located over obstacles as occurs in most navigation meshes. In the case of grid based navigation meshes or when T-joints exist between portals, this would not be the case for certain portals and thus the algorithm should only consider those endpoints that are located over obstacles. The complete pseudocode to treat cases 1,2 and 3 is described in Algorithms 8, 9 and 10 respectively.

Case 1: The algorithm starts by displacing each endpoint of $\mathcal{P}$ a distance of r units towards the center of the portal as we can see in Figure 7.4. The resulting sub-segment $\mathcal{P}'$ has enough clearance only if the other edges in $\mathcal{C}_A$ and $\mathcal{C}_B$ are at a distance greater than or equal to r from $\mathcal{P}'$. If not , this sub-segment must be further refined to guarantee collision-free traversability.

In order to further shrink portal $\mathcal{P}'$ based on cases 2 and 3, we need to consider portals and edges as if they were defined for each cell in counter-clock wise order (Figure 7.5 depicts this situation). $\mathcal{C}_A$ and $\mathcal{C}_B$ are thus two polygons with vertices given in counter-clockwise order. $\mathcal{P}$ can then be treated as two identical overlapping segments given in opposite order depending on which cell they belong to. We refer to them as $\mathcal{P}_{AB}$ for the oriented edge that belongs to $\mathcal{C}_A$, and $\mathcal{P}_{BA}$ for the oriented edge that belongs to $\mathcal{C}_B$.

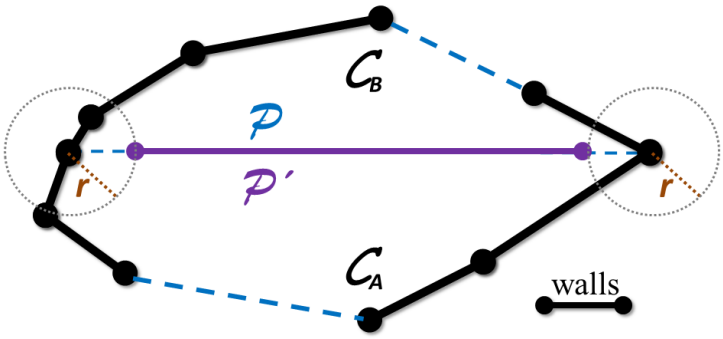

Figure 7.4: Example portal $\mathcal{P}$ that connects $\mathcal{C}_A$ and $\mathcal{C}_B$. The shrunk portal $\mathcal{P}'$ is initialized by displacing the endpoints of the original portal $\mathcal{P}$ a distance r towards its center.

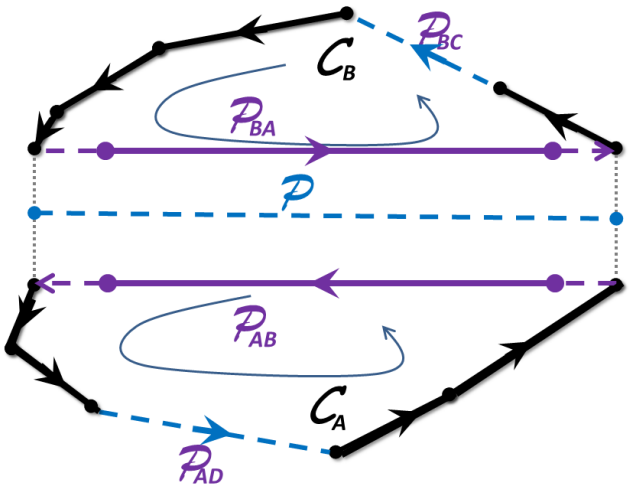

Figure 7.5: $\mathcal{C}_A$ and $\mathcal{C}_B$ separated by $\mathcal{P}$ are in fact two independent polygons with their vertices oriented in counter-clockwise order, so $\mathcal{P}$ is the overlapping of $\mathcal{P}_{AB}$ and $\mathcal{P}_{BA}$.

Case 2: Let $\mathcal{C}_B$ be an intermediate cell on the character's path (i.e., a cell that is neither the starting cell of the path nor the final one). In this case we have to cross the cell by crossing two portals, an entry portal $\mathcal{P}$ and an exit portal $\mathcal{P}_{BC}$ (portal that connects cell $\mathcal{C}_B$ with the next cell in the path $\mathcal{C}_C$). In such a situation, it is possible that the endpoints of $\mathcal{P}'$ are determined by the endpoints of any exit portals in $\mathcal{P}_{E1}, ..., \mathcal{P}_{En}$, where $\mathcal{P}_{Ei}$ indicates a portal in the sequence of portals that needs to be crossed to go from $\mathcal{C}_B$ to the final cell in the path $\mathcal{C}_{Goal}$. This occurs when one (or both) endpoint of a portal $\mathcal{P}_{Ei}$ is at a distance less than or equal to the desired clearance value from the entry portal $\mathcal{P}$. To handle this situation, we check if a circumference (of radius=agent's clearance) centered on the endpoints of the first exit portal $\mathcal{P}_{BC}$ intersects with $\mathcal{P}'$. If this intersection exists, we update $\mathcal{P}'$ accordingly and the process continues iteratively by checking the next exit portal. The algorithm stops when we find the first exit portal $\mathcal{P}_{Ei}$ that fails the test (none of its endpoints determines the endpoints of $\mathcal{P}'$) or when $\mathcal{C}_{Goal}$ is reached.

We take the polygons with oriented edges from Figure 7.5, and define $\mathcal{P}_{BA}[0]$ as the origin of the oriented portal $\mathcal{P}_{BA}$, and $\mathcal{P}_{BA}[1]$ as the end. As the portals are also given in counter-clockwise order, we can state that the origin of any portal can only limit the clearance of the end of the portal for which we are calculating clearance, so $\mathcal{P}_{BC}[0]$ can only shorten ${\mathcal{P}'}_{BA}[1]$, and $\mathcal{P}_{BC}[1]$ can only shorten ${\mathcal{P}'}_{BA}[0]$. The algorithm to further shorten ${\mathcal{P}'}_{BA}$ continues through the following two cases:

- If the circumference centered on $\mathcal{P}_{BC}[0]$ intersects ${\mathcal{P}'}_{BA}$ at a single point, then ${\mathcal{P}'}_{BA}[1]$ is set to be this intersection point.
- If the circumference centered on $\mathcal{P}_{BC}[0]$ intersects ${\mathcal{P}'}_{BA}$ at two points, then ${\mathcal{P}'}_{BA}[1]$ is set to be the intersection point that is furthest from $\mathcal{P}_{BA}[1]$.

Similarly, a circumference centered on $\mathcal{P}_{BC}[1]$ is checked for intersections against ${\mathcal{P}'}_{BA}$ to determine if ${\mathcal{P}'}_{BA}[0]$ needs to be updated. Figure 7.6 shows the result of the algorithm using an example cell. Figure 7.7 illustrates the importance of respecting the ordering of the portals when calculating portals with clearance. Even though in both cases the characters can walk through the portals, in the first case (Figure 7.7 top) the way points assigned over the portals would continuously push the characters to collide with the static geometry.

Case 3: The final case to consider takes into account whether any obstacle edge limits the clearance of the portal. This can happen when an edge or portal of the current cell is at a distance smaller than the clearance value of the portal that we are shrinking. In the case of portals, the process must be repeated recursively.

Given a cell $\mathcal{C}_X$, with a set of vertices in counter-clockwise order $\{v_0, v_1, ..., v_n\}$, where each consecutive pair of vertices in the sequence defines an oriented edge of the cell, i.e: $\vec{e}_{(i,i+1)}$ is the edge starting in vertex v_i and ending in vertex

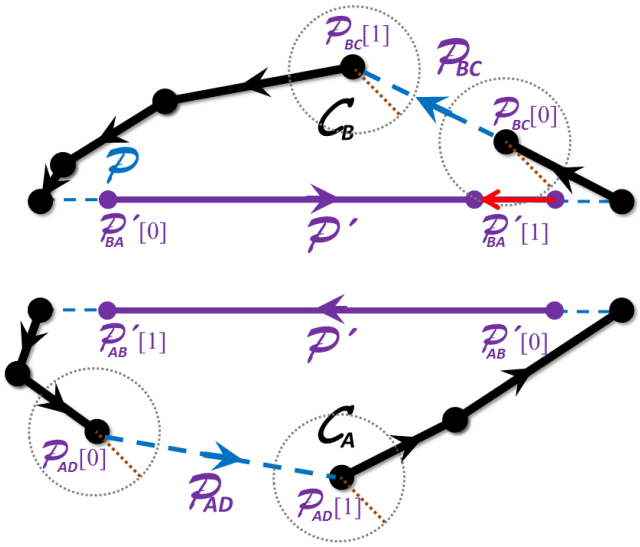

Figure 7.6: The endpoint $\mathcal{P}'_{BA}[1]$ of the entry portal is determined by the endpoint $\mathcal{P}_{BC}[0]$ of the exit portal, as the circumference centered on $\mathcal{P}_{BC}[0]$ intersects $\mathcal{P}'_{BA}$. The other end of $\mathcal{P}'_{BA}$ is not modified, as the circumference centered on $\mathcal{P}_{BC}[1]$ does not intersect $\mathcal{P}'_{BA}$.

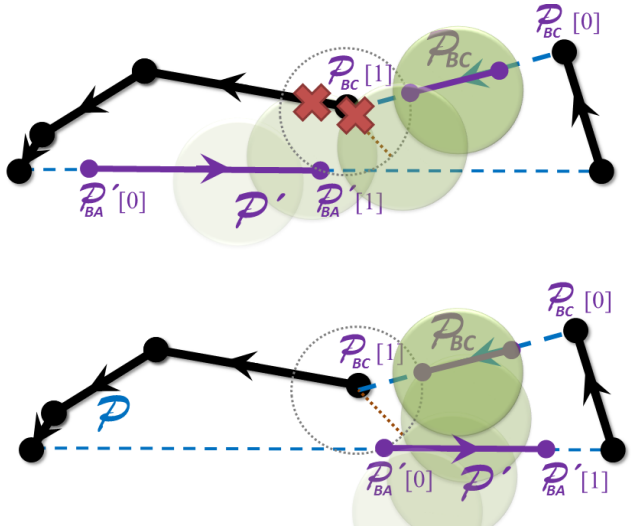

Figure 7.7: On the top we can see an example of what the character's trajectory would be if $\mathcal{P}'$ was not shrunk respecting the direction of the portals. The trajectory leads to a collision with the static geometry. On the bottom we can see the trajectory when clearance is calculated correctly.

v_{i+1}, for $i = [0, n-1]$. We define the shrinking direction of an edge, $\vec{s}_{(i,i+1)}$, as the unit vector perpendicular to the edge with its direction pointing towards the interior of the cell (Figure 7.8).

The algorithm proceeds by displacing each edge $\vec{e}_{(i,i+1)}$ a distance of r units along its shrinking direction, $\vec{s}_{(i,i+1)}$, if the shrinking direction points towards the portal (otherwise there is no chance of intersection). After displacement we obtain $v'(i)$ and $v'(i+1)$ as the results of displacing vertices v_i and v_{i+1} . For each displaced edge $\vec{e'}_{(i,i+1)}$, we calculate its intersection against $\mathcal{P'}_{BA}$, and if such an intersection exists, the corresponding endpoint of $\mathcal{P'}_{BA}$ is updated depending on the direction of $\vec{e'}_{(i,i+1)}$as follows:

1. if the edge is an obstacle edge:
   (a) If $v'_{(i)}$ is on the side of $\mathcal{C}_B$ and $v'_{(i+1)}$ is on the side of $\mathcal{C}_A$, then $\mathcal{P}_{BA}[0]$ is set to be the intersection point.
   (b) If $v'_{(i)}$ is on the side of $\mathcal{C}_A$ and $v'_{(i+1)}$ is on the side of $\mathcal{C}_B$, then $\mathcal{P'}_{BA}[1]$ is set to be the intersection point.
2. if the edge is a portal leading to $\mathcal{C}_D$:
   (a) If $v'_{(i)}$ is on the side of $\mathcal{C}_B$ and $v'_{(i+1)}$ is on the side of $\mathcal{C}_A$, then repeat the algorithm for the edges in $\mathcal{C}_D$ to update $\mathcal{P}_{BA}[0]$ if necessary.
   (b) If $v'_{(i)}$ is on the side of $\mathcal{C}_A$ and $v'_{(i+1)}$ is on the side of $\mathcal{C}_B$, then repeat the algorithm for the edges in $\mathcal{C}_D$ to update $\mathcal{P'}_{BA}[1]$ if necessary.

Figure 7.8 shows this process over the example scenario with a magnified view of the area of interest. The same process is performed for $\mathcal{P'}_{AB}$ and finally, $\mathcal{P'}$ is computed as the resulting sub-segment of the intersection between $\mathcal{P'}_{AB}$ and $\mathcal{P'}_{BA}$. Every point in $\mathcal{P'}$ is guaranteed to have enough clearance. Figure 7.9 shows the result of the algorithm.

To accelerate the computation of the shrunk portal, we store the result of the transformation for a particular value of clearance in a table. The next time that the portal needs to be shrunk, the table is checked for that particular clearance value so it does not need to be computed again.

In general, Case 3 will always be the most restrictive and thus the key calculation, however there can be exceptions such as illustrated in Figure 7.10 where case 3 does not limit the clearance of the portal. Therefore all three cases are necessary, as if we simply use distance from endpoints we would fail to generate natural paths in certain scenarios.

In Figure 7.11 we show an example where the recursive step would be necessary to compute exact clearance over the portal. Without recursivity the clearance on the left extreme of the portal would be given by the end point on the left hand side of the neighboring portal, but with recursivity it is further reduced to the new intersection point a.

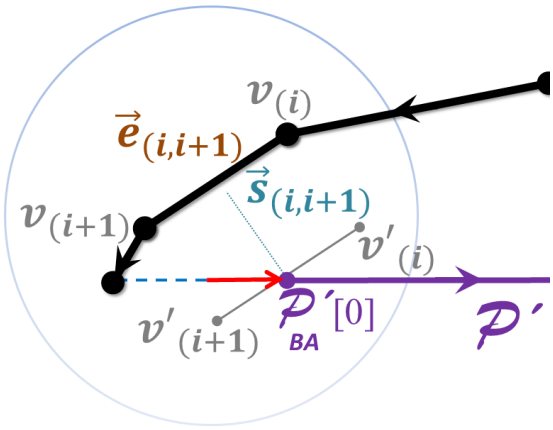

Figure 7.8: Close up of the top left of Figure 7.6 with the shrinking process due to displacing edges.

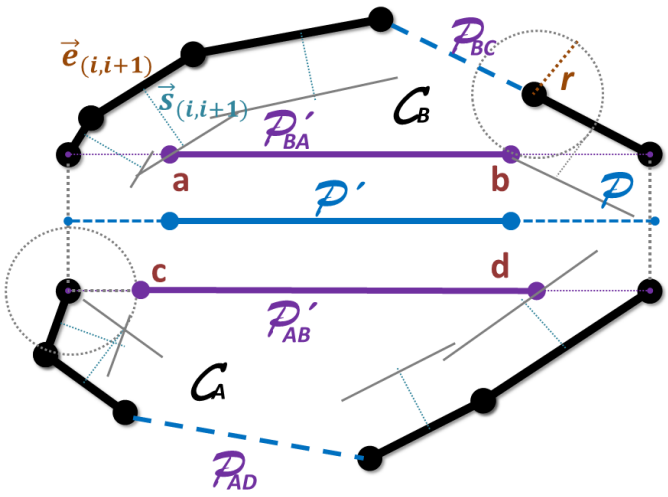

Figure 7.9: Final result $\mathcal{P}'$ after calculating the merging of the intermediate solutions $\mathcal{P}'_{AB}$ and $\mathcal{P}'_{BA}$. The resulting shrunk portal before merging illustrates the application of the three cases: Case 1 can be seen in c, Case 2 results in b and case 3 in a and d. $\mathcal{P}'$ is given in this example by the most limiting endpoints which are a and b.

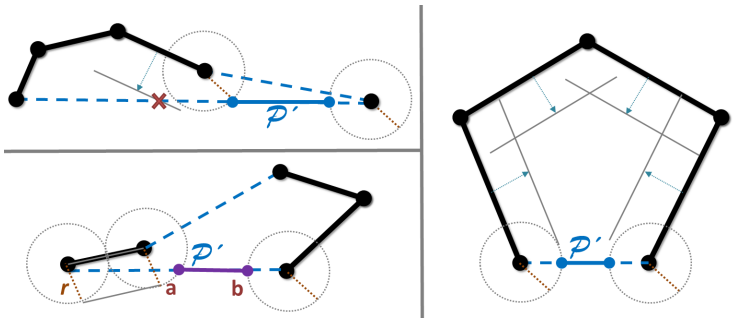

Figure 7.10: These examples show different situations where portal clearance is not defined simply by Case 3, and thus Cases 1 and 2 are necessary.

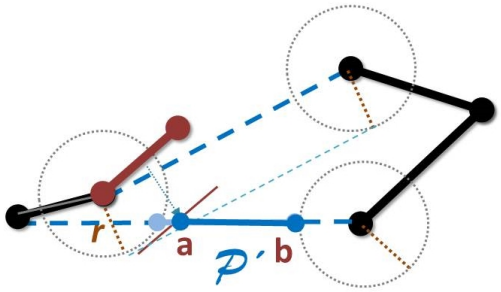

Figure 7.11: Example where the recursive step is necessary to compute clearance correctly.

Algorithm 8 Algorithm for computing portals respecting a minimum value of clearance r

1: **function** SHRINKPORTAL(Cell c, Portal p, Real r)
2: ▷ Case 1
3: $v_1 \leftarrow p[0] + p.direction \times r$
4: $v_2 \leftarrow p[1] - p.direction \times r$
5: ▷ Case 2
6: $v_1, v_2 \leftarrow shrinkPortalByPortalEndpoints(c, v_1, v_2, r)$
7: ▷ Case 3
8: $v_1, v_2 \leftarrow shrinkPortalByCell(c, v_1, v_2, r)$

Algorithm 9 Algorithm for computing portal shrinkment due to the influence of other portals r

```
1: function shrinkPortalByPortalEndpoints(Cell c, Vertex v1, Vertex v2, Real r)
2:     while c ≠ c_goal do
3:         p ← getPortalToNextCellInPath(c)
                                              ▷ Check if p[0] influences v2
4:         tmp ← segmentCircleIntersection(v1, v2, p[0], r)
5:         if tmp.isIntersection then
6:             if tmp.isSingleIntersection then
7:                 v2 ← tmp.intersectionPoint1
8:             else
9:                 v2 ← furthestPoint(v2, tmp.intersectionPoint1, tmp.intersectionPoint2)
                                              ▷ Check if p[1] influences v1
10:        tmp ← segmentCircleIntersection(v1, v2, p[1], r)
11:        if tmp.isIntersection then
12:            if tmp.isSingleIntersection then
13:                v1 ← tmp.intersectionPoint1
14:            else
15:                v1 ← furthestPoint(v1, tmp.intersectionPoint1, tmp.intersectionPoint2)
16:        c ← getNextCellInPath(c)
17:    return v1, v2
```

Algorithm 10 Algorithm for computing portal shrinkment due to the influence of obstacle edges in a cell r

```
1: function shrinkPortalByCell(Cell c, Vertex v1, Vertex v2, Real r)
2:     for all Edge e ∈ c.edges do
3:         p_dir ← (v2 − v1)
4:         if p_dir.dot(e.shrinkDirection) < 0 then
5:                 ▷ e is an edge pointing towards the portal we are shrinking
6:             v'_i ← e[0] + e.shrinkDirection × r
7:             v'_{i+1} ← e[1] + e.shrinkDirection × r
8:             tmp ← segmentSegmentIntersection(v1, v2, v'_i, v'_{i+1})
9:             if tmp.isIntersection then
10:                if e.isPortal then
11:                    v1, v2 ← shrinkPortalByCell(e.getNextCell(), v1, v2, r)
12:                else
13:                        ▷ e is an obstacle edge. Determine the point that needs to be updated.
14:                    if p_dir.shrinkDirection.dot(e.direction) < 0 then
15:                        v1 ← tmp.intersectionPoint
16:                    else
17:                        v2 ← tmp.intersectionPoint
18:    return v1, v2
```

7.3 Critical Radius:

All our calculations are required to perform in real time and we have already described an approach to speed up the system by storing information about clearance for agents of different radii. Other agents with radius similar to those already stored can then look up the information from a table instead of re-calculating. An additional technique implemented to speed up the process consists of pre-calculating a critical radius, ρ. The critical radius is defined as the maximum radius for which clearance depends exclusively on keeping a distance ρ from the portal endpoints. It is calculated by computing the minimum distance to an obstacle edge with its shrinking direction pointing towards the current portal. During run time, only agents of radius larger than ρ need to compute the portal clearance algorithm described in this section. Agents with radius r below ρ only need to keep a distance of r from the portal endpoints. As the critical radius is calculated off-line, this provides a speed up of 1.15 times faster on average during the real-time calculations. This speed up has been calculated over a variety of scenarios, most of them handmade to fully test the method. However in most of the scenarios obtained with *NEOGEN*, portal clearance is influenced exclusively by the portal endpoints, and thus the number of portals for which the full clearance algorithm needs to be executed will be minimal.

7.4 Dynamic Way Points

The method used to steer the character from one cell to another is a key aspect to create natural routes in *navigation meshes*. When way points are assigned at a fixed position, usually the center of the portals, animation artifacts arise (Figure 7.12). The most common artifacts are line formation among characters that move in the same direction, and bottlenecks caused by characters crossing cells in opposite directions and being forced to pass through the same point. A typical approach in video games consists of setting the way points at a distance r from the closest endpoint of the portal (where r is the radius of the character). This solution provides slightly more natural paths since paths are apparently shorter and at least two way points are available for each portal, but it does not completely solve the problem. Our work focuses on dynamically calculating way points over the shrunk portal (Figure 7.13).

Our dynamic way point assignation is based on the position of the character within the cell. First of all, we check if the goal position of the character is visible from its current position (i.e., the segment joining the current and the goal position of the character only produces an intersection with portal edges). In that case, the attractor point is simply the goal position. If the segment does intersect with at least one obstacle edge, we need to compute a way point over the next portal in the path to steer the character towards the next cell of the path. Our target is to avoid characters having the same attractor point,

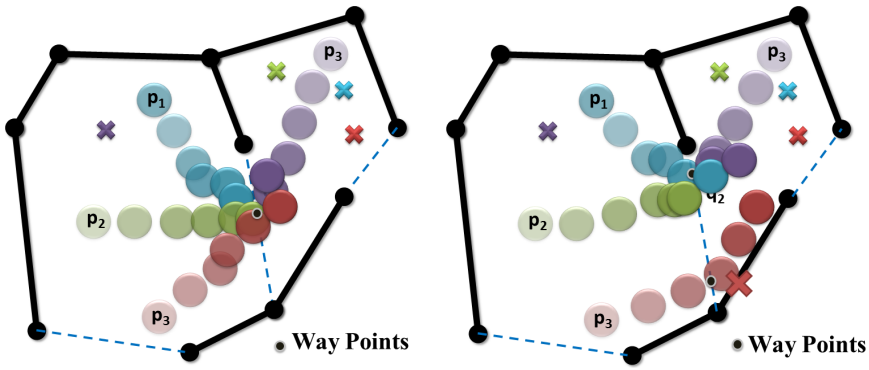

Figure 7.12: Typical lining up artifacts and bottlenecks when way points are set at either the center (left) or the closest endpoint of the portal (right).

so we compute the orthogonal projection point $\mathbf{q}$ of the current position of the character $\mathbf{p}$ over $\mathcal{P}'$, where $\mathcal{P}'$ is the shrunk portal after applying the algorithm described in Section 7.2 over the portal $\mathcal{P}$. If $\mathbf{q}$ lies outside the limits of $\mathcal{P}'$, then the furthest endpoint of $\mathcal{P}'$ with respect to the current position of the character is selected as a temporal attractor, until $\mathbf{q}$ is valid.

The position of the characters is given by the local movement algorithm used to steer them. This algorithm will naturally move characters away from each other to avoid collision. Each character's position approaching a portal will be different, so their projection over the portal will also be different making it virtually impossible for two different characters to share the same attractor point over the portal if the characters are at risk of colliding.

We have determined empirically that in the case of $\mathbf{q}$ being invalid, the furthest endpoint of $\mathcal{P}'$ is a better candidate as a temporal attractor than the closest one. This is because when the steering attractor is the closest endpoint, the character tends to move too close to the walls, producing a bad quality route.

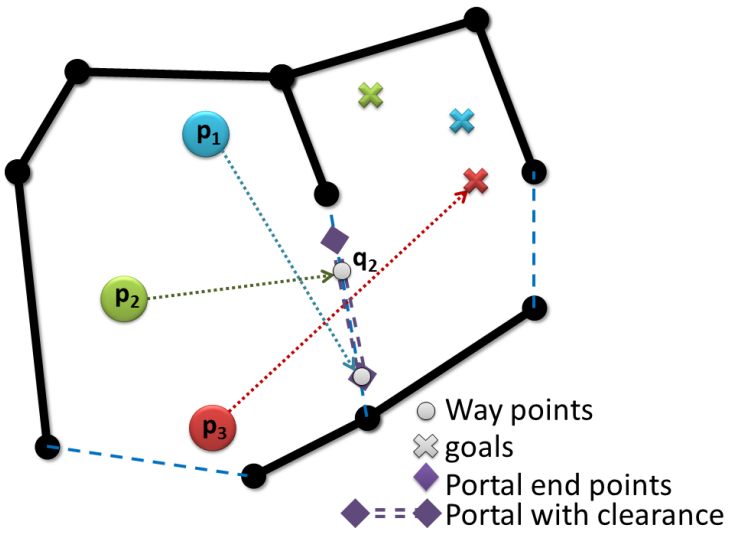

Figure 7.13: The attractor point of the red character is its own goal since it is visible from its current location. The green character has its orthogonal projection, $\mathbf{q}_2$, over the portal as its way point, whereas the blue character has the farthest away endpoint of the portal assigned as its way point, since its current orthogonal projection lies outside the portal with clearance $\mathcal{P}'$.

7.5 Local Movement

The local movement algorithm is based on a simple steering behavior with some extension to include physical forces as described in HiDAC [67]. Collision detection and repulsion forces between agents are calculated using the Bullet Physics Engine [8]. We have also used this library to perform calculations to speed up the detection of agents crossing portals. Agents move towards their next assigned dynamic way point while avoiding the static geometry and other moving obstacles. In order to keep track of the cell in which the character is located we have taken advantage of some of the features that the Bullet Physics Engine offers. By assigning a rigid body to the floor of each cell, we can efficiently compute the intersection between the character and the cells using Bullet's space partitioning.

This solves artifacts that usually appear when agents approach their assigned way point, and end up moving back and forth trying to reach the threshold distance to the target point. With our technique, a portal can be crossed at any point independently of the distance to their next assigned way point.

Note that with the method described above to detect when agents cross portals, we improve the local movement of both centered and dynamic way points. Traditional center way points require the agents to be a certain distance from the way point in order to assign the next portal. In many cases this leads to agents moving back and forth around portals as they attempt to reach a specific distance from an attractor. In our implementation this is not strictly necessary, as agents may cross portals despite not having reached their next way point. When this happens, they are immediately assigned to a new way point in the next portal without losing track of their current cell information. This avoids a common problem that arises in many simulations where the agents only update their current cell when they have reached their assigned way point, and thus agents may end up "lost".

7.6 Results

In order to evaluate the results obtained with our algorithm, we have carried out both qualitative and quantitative analysis. We have examined whether our clearance method combined with dynamic way points achieves a better use of space, and whether the performance of our algorithm is sufficient to work with large groups of agents in real time whilst computing paths with clearance and collision free way points.

Figure 7.14 (and the accompanying videos[1]) shows a comparison between using traditional way points (WP) at the center of portals and our method with dynamic way points (DWP) for two example scenarios. The first scenario is

[1] www.lsi.upc.edu/~npelechano/videos/C&G2014_Clearance.mov

shaped as a donut and the second is shaped as a cross with static obstacles randomly located. The local movement algorithm is the same for all scenarios, and it is based on a simple rule based model with collision avoidance, steering towards attractors (way points) and collision response. For each character, a random cell of the environment is selected as its destination cell. A path finding algorithm based on A* calculates the sequence of cells that the character needs to walk through to go from its current cell to the destination. Way points are assigned over portals connecting consecutive cells. Once a character reaches its destination cell, a new one is randomly assigned. Characters are considered to cross a portal as soon as the Bullet Physics Engine [8] detects that the character has arrived in the next cell of the path. Dynamic way points make better use of the space, use straight trajectories whenever possible and offer more natural looking trajectories for the characters, even when using a very simple rule based model for their local movement. When way points are fixed at the center of portals, we can observe that not only do the paths not make use of the available space but also that they are more chaotic as characters bounce around portals trying to get close to the way point while avoiding each other.

Dynamic way points offer a better distribution of agents over portals which allows more agents to cross portals simultaneously. This increases flow rates through portals since it avoids artificial line formation. For example, in the donut scenario with 200 agents walking in the same direction, we observe 22% higher flow rates.

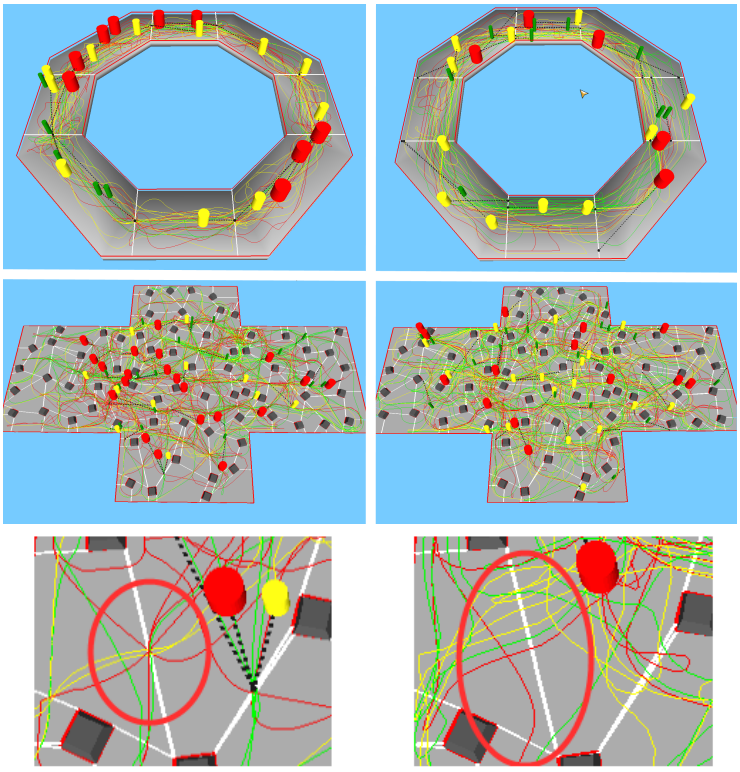

Figure 7.14: Comparison between having way points at the center of portals (on the left) and dynamic way points (on the right) for the donut scenario with 25 agents (top row), large cross scenario with 50 agents (middle row) and close up of the paths crossing a portal (bottom row).

7.6.1 Performance

The following results have been obtained in an Intel Core i7-3770 CPU @ 3.40GHz, 16GB of RAM, NVIDIA GeForce 680GTX . Figure 7.15 compares the time spent per query (microseconds) of different versions of the portal shrinking method:

- **SimpleShrink(-/+)**: A fast and simple method, commonly used on video games and other virtual applications, that simply displaces the endpoints of the portal r units towards its center. The (+) version uses a lookup table to store previously computed shrunk portals, and the (-) calculates it at every simulation step.
- **ExactShrink(-/+)**: Our exact clearance solution described in Section 7.2. The (+) version uses a lookup table to store previously computed shrunk portals while the (-) calculates it at every simulation step.

Each test case consists of a set of queries where, for each query, we randomly chose a cell of the *NavMesh*, a trajectory to cross this cell (i.e. an entry portal and an exit portal) and a clearance value (0.5, 1 or 1.5).

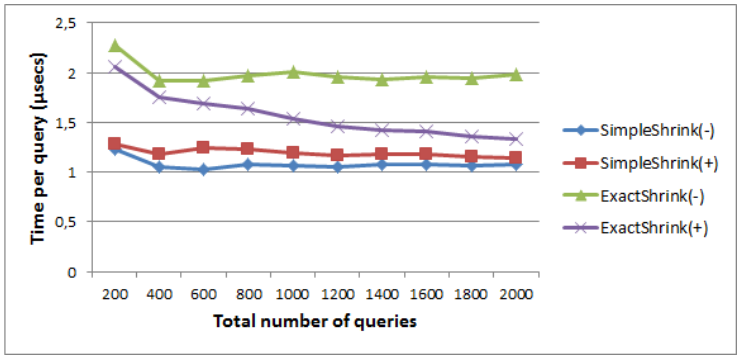

Figure 7.15: Comparison of the time taken per query (in microseconds) as the number of queries increases for the different shrinking techniques.

The results of this experiment highlight the efficiency of our exact clearance method (*ExactShrink(+)*). The efficiency of the algorithm increases with the number of queries as the chance of producing a redundant query is higher, and eventually, every query will be redundant. Results show that for the case of 1000 random queries, the cost of *ExactShrink(+)* is just 1.41 times the cost of the most efficient version (in this case *SimpleShrink(-)*) and 1.2 times for 2000 random queries. This means that the algorithm for calculating portals with exact clearance presented in this chapter (*ExactShrink(+)*) is around 20% more time consuming for 2000 queries than simpler implementations, but it also guarantees that every computed path will have enough clearance with the

static geometry. As the number of queries increases, this percentage is further reduced. For the given example, we get a probability of hit of 50% for 1000 queries, which means that one in two queries does not need to be computed since it is already stored in the lookup table, and 90% probability of hit when it reaches 6000. The time taken by the *ExactShrink(+)* algorithm converges towards the *SimpleShrink(+)* method.

It is also important to emphasize that this increment in time does not have a big impact on the overall simulation since it is insignificant compared to the cost of AI, rendering or physics.

Including the recursive step when calculating clearance makes our method more robust without introducing a noticeable impact on the computational time.

The memory requirements to store the lookup table are minimal, since for each radius size we only need two 3D point coordinates for the corresponding shrunk portal. For example, in the cross scenario with 208 portals, 3 character sizes and 12Bytes per 3D point, the total memory required is less than 15K.

As there are many elements that affect the resulting frame rate of an application, such as: rendering engine, physics library, local movement algorithm, size of the scenario, size of the crowd, and so on, we are not interested in how many characters we can simulate in real time, but in comparing our method for paths with clearance against the standard solution where characters walk towards way points fixed at the center of portals without checking for any kind of clearance against the static geometry. Figure 7.16 shows a comparison of the average frame rate achieved as the number of characters increases with and without our technique, when all the other elements of the simulation stay the same. This graph compares the standard solution (in red) against our technique (in blue). The results are practically the same (less than 5% smaller frame rate on average with our method), meaning that the computational time required to calculate portals with clearance and dynamic way points is insignificant within the overall simulation time. Both simulations can handle up to 500 characters in real time.

Therefore we can claim that the computational cost of our technique is insignificant for the overall simulation time and that it provides results that are perceptually more convincing and make better use of the space, as shown in Figure 7.17 and the accompanying videos.

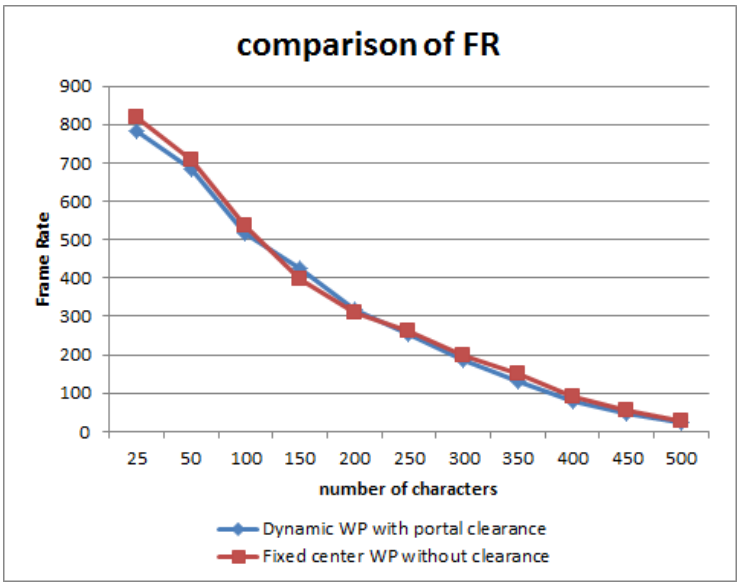

Figure 7.16: Average frame rates obtained in the large "'cross"' scenario as the number of characters increase for our method and a standard solution

7.6.2 Path finding

To show the results achieved by the path finding algorithm with clearance, we can observe in Figure 7.17 the different paths used by the characters depending on their size. The larger characters only traverse those cells with a clearance larger than their radius. Another nice outcome of the presented method is the use of space made by the characters depending on their size. We can observe in the image how as the characters' size decreases, the final emerging trajectories of their color are wider, since their way points are assigned over larger shrunk portals.

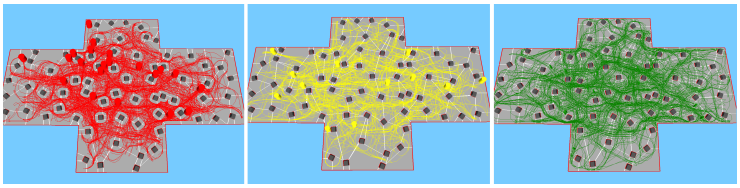

Figure 7.17: Trajectories followed by characters of different size. From left to right, the larger characters (red, $r = 2.0$) will not use the narrower portals and thus they can only walk through 97 of the 130 cells in the *NavMesh*, the medium characters can already get through most of the portals (yellow, $r = 1.5$) therefore being able to walk through 110 cells, and finally the smaller size characters (green, $r = 0.5$) can walk through all the portals having the largest shrunk portals (walkable cells=130).

7.6.3 Comparison of dynamic collisions

To demonstrate quantitatively that having dynamic way points not only provides better visual results independently of the local movement algorithm used, but also drastically reduces the number of collisions by spreading the crowd over the length of the portal, we have run several experiments to compare the average number of collisions for both fixed center way points and dynamic way points.

We account for a collision between two rigid bodies at every tick of the physics engine (60x per second). Collisions are considered when an agent is in contact with the geometry (which also accounts for agents being stuck next to a wall due to a badly located way point)

As shown in Figure 7.18, for up to 100 agents the number of collisions between agents is almost zero, since at low densities there are not many chances of collisions and basic avoidance behavior can steer agents away from collisions. However once the densities start increasing we can observe how even when all the agents move in the same direction, collisions start appearing. As the graph shows, the number of collisions for fixed center WP is much higher than for

DWP, since forcing all the agents to move towards the same point leads to chaotic behavior with loops in the agents' trajectories. This occurs for up to 175 agents for the donut scenario, since from this point onwards the density of agents in the environment is so high that bottlenecks are almost impossible to avoid.

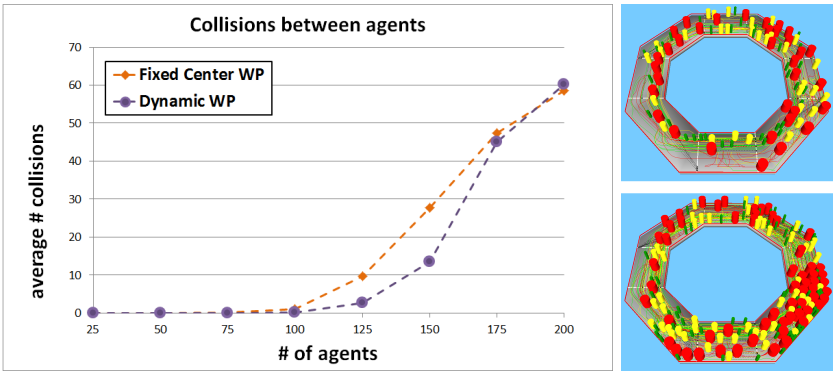

Figure 7.18: Comparing the average number of collisions per second between agents for the donut scenario as the number of agents increases. We compare dynamic way points against fixed center way points. On the top right we show the scenario with 100 agents and on the top bottom with 175 agents

In Figure 7.19 we can observe a comparison between the average number of collisions per clock tick as the number of agents increases for fixed centered versus dynamic way points. Our method to dynamically assign way points achieves a much lower number of collisions between agents which not only reduces artificial bottlenecks in the environment, but also results in smoother and more natural trajectories. As in the donut scenario, once the number of agents increases beyond 125, differences in the number of collisions start emerging between DWP and fixed center WP, until the total number is higher than 225. At this point, the high density of agents makes collisions inevitable, independent of the method used.

While the graphs vary depending on the size of the scenario, length of portals and local navigation method, we observe that in all of our experiments, dynamic way points achieve better results than fixed center WP.

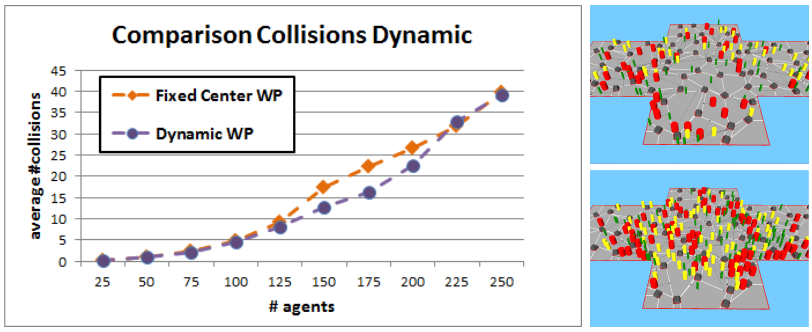

Figure 7.19: Comparing the average number of collisions per clock tick between agents for the cross scenario as the number of agents increases. We compare dynamic way points against fixed center way points. On the top right we have the cross scenario with 125 agents, and on the bottom right the same scenario with 225 agents.

7.6.4 Comparison of collisions against geometry

The main advantage of having exact clearance calculations is that we guarantee that way points will only be assigned over portals where collision free paths exist. To evaluate this quantitatively, we have run several experiments using different scenarios and compared the following methods: (1) dynamic way points (DWP) over portals with exact clearance, (2) DWP over portals with simple clearance, and (3) fixed center way points. For the three methods, the local movement algorithm is the same, and the agents' goal cell is chosen randomly every time they reach their destination. For each case we have counted the number of collisions against the geometry that results from way points being badly assigned.

Obviously the results depend strongly on the quality of the portals created and the overall geometry. To show the potential of our method, we have designed scenarios with several examples of problematic portals (mostly ill-conditioned portals).

Figure 7.20 shows the results of each of the methods in terms of paths followed by agents, and situations where they can easily get stuck trying to walk through a portal that does not guarantee clearance. As shown in Cases 2 and 3, agents may even get completely stuck against the geometry, whereas with our exact clearance method, agents are always steered towards way points that guarantee traversability. This holds even for maps with many ill-conditioned cells, such as the ones created manually for these experiments.

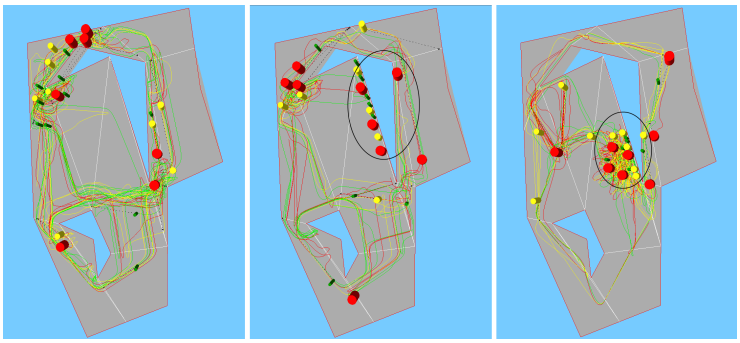

Figure 7.20: Comparing paths between the three methods. From left to right: (1) DWP over portals with exact clearance, (2) DWP over portals with simple clearance, and (3) fixed center way points. The areas where agents get stuck due to an ill-conditioned cell with a portal too close to the geometry (narrow cell) are circled.

The quantitative results in terms of number of collisions against the geometry for this particular scenario are shown in Figure 7.21. The three methods use the same local movement algorithm, therefore the only difference comes from how and where way points are assigned. Our method outperforms previous work with regards to reducing the number of collisions against the geometry. We have performed comparisons for different crowd sizes. We have demonstrated that the differences become less significant as the crowd size increases. This occurs because there is a point where collisions are due to the high density of the crowd and not just the location of way points. In all cases, exact clearance provides the lowest number of collisions against the geometry. If we compare fixed center against dynamic way points with simple clearance, fixed center performs better when it comes to avoiding collisions against the static geometry, since in most cases the center way point will be located at the furthest point from the geometry.

Finally, Figure 7.22 shows the importance of using our exact clearance calculation when there are ill-conditioned cells. In this example we can see the portal calculated with our exact method against the simple method often used in video games. In both cases the segments over the portals that are traversable for each method are shown with a thin blue line. The character for which this clearance has been calculated is also circled in blue. In both examples, a red agent is trying to move from cell A to cell B. Our exact clearance algorithm provides the exact segment over the portal that can be crossed without collisions or errors. In the case of simple clearance, we can observe how the character is being steered towards a position that will lead to the wrong cell and to collisions against the geometry.

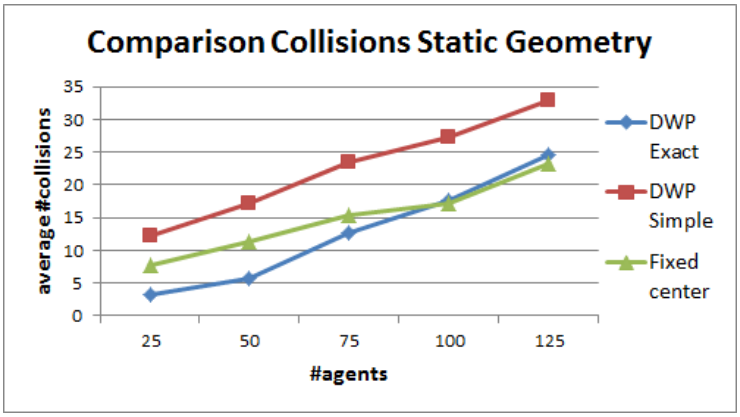

Figure 7.21: Average number of collisions against the geometry for each method tested (collisions counted at each clock tick, which corresponds to 60Hz).

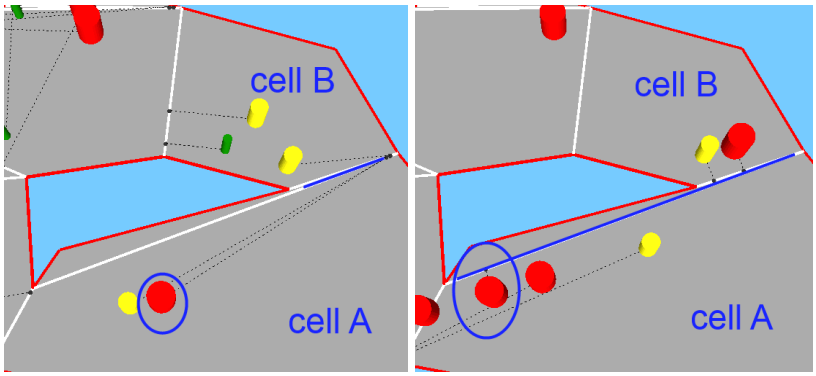

Figure 7.22: Clearance calculated with our exact algorithm (left) and with the simple clearance method (right).

7.7 Limitations of Path finding with Clearance

When generating the *CPG* from a *NavMesh*, a problem may arise that consists of having cycles on the graph due to the characteristics of the agents and the environment. Let us take as an example the situation described in figure 7.23. In this case, there are two possible solutions when computing the path from *start* to *goal*:

- *A, B, F*
- *A, B, C, D, E, B, F*

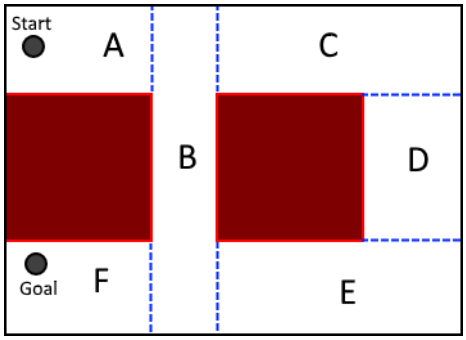

Figure 7.23: An arbitrary *NavMesh* that could be the result of partitioning the environment into cells (letters from *A* to *F*) and portals (dashed blue lines). Depending on the desired amount of clearance, a cycle will occur when computing the path from *start* to *goal* as a character may need to pass twice through the cell *B*.

The chosen path will depend on the desired amount of clearance. However, the second path is not valid, as it contains a cycle, i.e., the agents needs to pass twice through cell *B*. Note that this is not an exclusive problem of *NEOGEN*, but a general problem that may arise in any kind of *NavMesh*, excluding a few examples where the data structures used already guarantee a *CPG* free of cycles by construction [38].

Our solution lies in the observation that although in the second path we pass through the same cell twice, we use a different *cross* each time, i.e., the first time we cross cell *B*, we enter by the portal connecting *A* and *B*, and we exit by the portal connecting *B* and *C*. Similarly, the second time we cross cell *B*, we enter using the portal connecting *E* and *B*, and we exit by the portal connecting *B* and *F*. Therefore, our *A* Node* data structure is defined as a cell cross, instead of simply containing the identifier of a cell as it is usually done. This way the problem of having cycles is solved because a single cross will appear once and only once in the path. The data structure used to represent a *Node* in our *A** version is as follows:

```
struct CrossNode {
  Portal portalEntry
  Portal portalExit
  Cell cell
  Real clearance
}
```

The pseudocode of our method is described in Algorithms 11 and 12. For the sake of clarity, we have used the well known $A*$ as a base for our *path finding* algorithm. However, notice that this technique can be applied to any existing *path finding* algorithm such as the ones described in the state of the art chapter.

Algorithm 11 Path finding algorithm based on A*

```
1: function computePath(Cell start, Cell goal, Real clearance)
2:                                   ▷ Create the open and closed sets of Nodes.
3:     Let closedSet be a new Set.
4:     Let openSet be a new Set.
5:
6:         ▷ Create the auxiliary sets to store the g and f value of each Node.
7:     Let gValue be a new Set.
8:     Let fValue be a new Set.
9:
10:                ▷ Initialize the data structures with the initial set of Nodes.
11:    initialNodes ← generateNodes(start, null)
12:    for all CrossNode n_i ∈ initialNodes do
13:        gValue[n_i] ← 0
14:        fValue[n_i] ← computeHeuristic(n_i, goal)
15:    openSet.add(initialNodes)
16:
17:    while openSet.hasValues do
18:                            ▷ Get the node in openSet with the lowest f value.
19:        current ← getNextNode(openSet)
20:        if current.cell = goal then
21:            return reconstructPath(cameFrom, current)
22:
23:        openSet.remove(current)
24:        closedSet.add(current)
25:        cellNext ← current.cell.getAdjacentCell(current.portalExit)
26:        neighbors ← generateNodes(cellNext, current.portalExit)
27:        for all CrossNode n_i ∈ neighbors do
28:            if (n_i.clearance ≥ clearance) & !closedSet.contains(n_i) then
29:                g ← gValue[current] + distance(current, n_i)
30:                if !openSet.contains(n_i) then
31:                    openSet.add(n_i)
32:                else if g < gValue[n_i] then
33:                                    ▷ This path is the best until now. Record it.
34:                    cameFrom[n_i] ← current
35:                    gValue[n_i] ← g
36:                    fValue[n_i] ← g + computeHeuristic(n_i, goal)
37:    return null
```

Algorithm 12 Algorithm to generate the A* nodes, given a Cell c and an entry portal.

```
1:  function generateNodes(Cell c, Portal pEntry)
2:      Let S be a new Set.
3:      for all Portal p_i ∈ c.portals do
4:          if p_i ≠ pEntry then
5:              n ← new CrossNode
6:              n.portalEntry ← pEntry
7:              n.portalExit ← p_i
8:              n.cell ← c
9:              n.clearance ← computeClearanceCrossCell(c, pEntry, p_i)
10:             S.add(n)
11:     return S
```

7.8 Conclusions

We have presented a general technique to compute paths free of obstacles with an arbitrary value of clearance that can be easily integrated in any existing *navigation mesh* system.

Our method can be divided into the following three steps. Firstly, during the construction of the *NavMesh*, the clearance value of each cell is computed in order to obtain paths that guarantee clearance when applying the $A*$ algorithm. Secondly, the portals of the path are refined by shrinking them depending on the clearance required for each character and the surrounding geometry. Finally, way points over the shrunk portals are computed based on the character position and hence, it mostly avoids two characters sharing the same attractor point.

Bullet Physics Engine [8] has been integrated in order to improve the overall quality of the simulation. Although its main purpose is to solve the collisions against moving and static geometry, we have used Bullet to efficiently detect when a portal crossing has been produced and avoided artifacts that arise in traditional methods as characters approach their target position.

Results show that our method is fast enough compared to simplest implementations, but produces paths of higher quality as it takes into account clearance for both path planning and way point calculations, and its dynamic assignation of way points along portals avoids characters lining up when crossing portals or causing bottlenecks.

We have tested our algorithm with *NavMeshes* of a variety of scenarios created by *NEOGEN-ML* [65] which is a *NavMesh* generator that provides an almost near-optimal number of cells with very few ill-conditioned cells. To show the potential of our method even for other kinds of *NavMeshes*, we have also manually generated navigation meshes with ill-conditioned cells.

For the qualitative evaluation of this work we have considered that higher quality paths are those that tend to use most of the available space, avoid artificial line formation, reduce bottlenecks and collisions. In this chapter we have also provided a quantitative evaluation of the improvements achieved with our exact clearance method by counting collisions against static and dynamic geometry. Results show how our method provides not only smoother paths with better usage of space, but also reduces the average number of collisions that are caused by way points not being correctly assigned. Compared to our previous work [64], we have made significant improvements in terms of generality as our new algorithm can handle a larger variety of navigation meshes, while improving performance with the introduction of the critical radius and a revised version of the code.

Finally, we address the problem of having cycles when computing paths with *clearance* on *NavMeshes*. Our solution consists in a new encoding of the classic $A*$ algorithm, which uses a special data structure to encode the *Nodes* as the

different ways of crossing a cell. This way the problem is naturally solved, as a specific cell cross appears once and only once on a given path. Although we have used $A*$ as a base, any other *path finding* algorithm can be extended using the same idea.

Chapter 8

Conclusions & Future Work

8.1 Conclusions

In this thesis we have presented a complete framework for the navigation of autonomous characters in complex virtual environments. As most approaches in the literature, our solution splits this problem into two sub-problems. First, we have introduced solutions for *global movement* algorithms by developing a novel technique for the generation of *navigation meshes* for a given virtual scene. Then, a novel *local movement* technique has been introduced, that exploits the information of the free space of the scene provided by the previous data structure in order to guide the agents through the scene in a natural manner. Our system introduces several improvements in both fields with respect to the current state of the art.

In the case of the *global movement* technique, first we have presented *NEOGEN-2D*, a novel automatic *NavMesh* generator that takes as an input any simple 2D polygon (which may contain holes) and outputs a near-optimal partition consisting of convex polygons. The polygons can represent the floor plan of a given environment, with holes representing static objects such as walls. Our algorithm focuses on the idea of sequentially splitting notches into convex areas by creating a portal with the notch and the closest element in its *Area of Interest*. Since our approach is based on subdividing the original polygons with segments instead of diagonals, we achieve on average a smaller number of convex cells in the environment than previous work in the literature based only on diagonals.

We have also introduced the concept of *convexity relaxation*, based on the fact that small concavities in the environment can be easily overcome by most *local movement* algorithms, and thus we state that for *navigation meshes* we can relax the notch condition by ignoring small concavities. The ultimate goal of this step is to further reduce the number of generated cells, especially ill-conditioned ones. Results show that convexity relaxation is a powerful tool to

reduce the final number of cells, especially when the scenario contains many rounded objects. To the best of our knowledge, this is the first time that a *NavMesh* generator applies this concept successfully to reduce the size of the *CPG*. In fact our results show that the convexity relaxation concept can reduce the final number of cells even below the smallest value of the optimality bound.

Then, we have presented our system *NEOGEN-ML*, a method for automatically computing a near-optimal convex decomposition for any multi-layered 3D environment. In this approach we start by doing a coarse voxelization of the scene in order to get an approximation of the walkable area. This walkable area is subdivided into several layers by using a flooding process that guarantees each layer is a 2.5D floor plant representation. Then, for each layer, we obtain a high resolution depth map and a contour detection algorithm is applied in order to obtain a 2D simple polygon with holes. The individual *NavMesh* of each layer is computed by applying the algorithm described in Chapter 3. Finally, all the individual *NavMeshes* are joined into a single *NavMesh* representing the navigable space of the whole scene. The main properties of *NEOGEN-ML* are given by the voxelization structure used to split the layers and filter each 2D projection. This offers the advantage of providing and accurate reconstruction of the 2D contour of the geometry, which simplifies the process of dealing with non-perfect geometry (holes, intersections, non-manifolds, etc). However the need of a voxelization brings some limitations such as: maximum map size, resolution of the depth map for the filtering step, and also resolution on the Y dimension. For this reason we evaluated for some time how to overcome those limitations while still keeping the core of the method, but in the end we realized that we could develop a new method that will keep the core ideas behind the partitioning but would free us from the voxelization.

The new system *NEOGEN-3D*, fully supports any 3D input geometry and overcomes all the limitations detected on the previous *NEOGEN-ML*. The algorithm can be described as follows. First, we classify all the faces of the 3D model into walkable or obstacle depending on the angle formed by the normal of the face and a user defined world-up vector. Then, the walkable area is further refined by applying a ceil-constraint algorithm that detects the exact portion of the walkable area that is not accessible due to the geometry above it. Finally, the *NavMesh* of the scene is constructed by applying the 3D version of our core algorithm that is able to detect the closest element to a notch by following the face connectivity described by the input 3D model itself. Notice that our approach completely respects the input geometry at any time, so the resulting *NavMesh* perfectly fits the virtual environment. This is of main importance when applying the *local movement* algorithm as it avoids problems on character location and other artifacts such as foot-floating. Initial results from the latest version of *NEOGEN-3D* are promising, as it handles a larger number of 3D scenarios. This method not only provides an exact adjustment to the geometry but also it is not limited in any way by the resolution of the voxelization described in Chapter 5.

Regarding the *local movement* part of this thesis, we have presented *ExACT*, a general technique to compute paths free of obstacles with an arbitrary value of clearance that can be easily integrated in any existing Navigation Mesh system. Our method can be divided into the following steps: Firstly, during the construction of the *NavMesh*, the clearance value of each cell is computed in order to obtain paths that guarantee clearance when applying the $A*$ algorithm. Secondly, the portals of the path are refined by shrinking them depending on the clearance required for each character and the surrounding geometry. Finally, an attractor point over the shrunk portal is computed depending on the character's position and hence, avoids two characters sharing the same attractor point.

Results show that our method is fast enough compared to much simpler implementations, although produces paths of higher quality as it takes into account clearance for both path planning and way point calculations. Assigning dynamically way points along portals avoids characters forming lines when crossing portals, since the avoidance behavior of the local movement algorithm will steer their trajectories correctly and thus modify the projection of their positions over portals.

Finally, we have addressed a problem that arises due to the fact that path planning algorithms such as $A*$ do not admit loops, so a character can pass on a given node of the graph once and only once. However, due to the underlying *NavMesh* and agent characteristics, this is not always possible. We solve this problem by using a special data structure to encode the *Nodes* as the different ways of crossing a cell. This way the problem is naturally solved, as a specific cell cross appears once and only once on a given path. To the best of our knowledge, this is the first time that a general solution is proposed for this specific problem.

8.2 Future Work

Regarding the *global navigation* algorithm, we need to integrate support for dynamic scenes. The current implementation of our *NavMesh* generator, only takes into account the static geometry which is enough for most applications, as collisions against dynamic obstacles such as other agents are driven by the *local movement* algorithm. However, it is common in applications such as video games to have dynamic worlds that are constantly changing (for example, an explosion that creates a crack on the floor, a tree that falls and blocks a path, a door that blocks or makes accessible a region of the scene, etc.). In those situations, the *NavMesh* needs to be modified. We would like to further improve our *global movement* algorithm to also handle such dynamic events in real time and modify the *NavMesh* in consequence. We believe that our latest method *NEOGEN-3D*, could easily deal with dynamic environments, as it is now free from recalculating the voxelization, and it also has a much smaller cost when searching for closest elements.

Another key improvement would be to include support for a wider range of character skills. Currently, our system assumes that a character can only reach its goal point by walking or running. However, a real character can do other richer actions, such as crouching, jumping or climbing, just to mention a few. Such actions give access to parts of the scenario that are not accessible by walking animations. So it is necessary to develop techniques to automatically identify the main features of the scene and associate such information to the portals of the *NavMesh*, so the agent knows what animations can be played at each moment, and also plan paths across the *NavMesh* accordingly.

Additionally, we would need our system to be able to identify in a semi-automatic form, which actions the characters can perform depending on the zone they are. This information is not only needed for the aforementioned goal, but also we could use it to modify the behavior of the characters depending on which zone they are. For example, if we think on a university campus, the system should be able to determine what could be a library, an office or a classroom. People behave differently in each of these scenarios, so the agents could also exhibit different behaviors based on their role and on the type of cell being visited.

Regarding to the *local movement* part of our framework, we would like to enhance the agent behavior against dynamic obstacles since our current implementation is based on a simple rule based model which does not predict trajectories of other moving obstacles. Also, more rules needs to be implemented. Currently, agents just move from one starting point to a goal point on the scene while avoiding collisions against static geometry and other agents, but they could exhibit more complex behaviors such as grouping and collaborating to realize a group task.

In addition, for the simulation of truly realistic crowds, we need to introduce psychological rules for the agents. The skills of a real character depends not only on its physical attributes (strength, size, speed, etc) but also on its psychological attributes (leadership, fear, doubt, etc). A really strong character can be totally paralyzed if he is in a panic state. This would allow us to simulate complex situations such as a violent protest or a terrorist attack as well as it can be used to study how the emotions are propagated from individual to individual.

Finally, we would like to do research on parallel programming and computing techniques, both CPU (multi-threading) and GPU accelerated techniques (CUDA and OpenCL), in order to improve the general performance and scalability of the whole framework presented in this thesis. In the case of the *global movement* part, it not only would reduce the time needed to generate the *NavMesh*, but also it would be of main importance when introducing the dynamic changes support, as it requires to modify the *NavMesh* online in real-time. Regarding to the *local movement* part, we can exploit parallelism in order to be able to simulate large crowds exhibiting really complex behaviors, both in the per-agent level as well as at the crowd level.

Bibliography

[1] O. Arikan, S. Chenney, and D. A. Forsyth. Efficient multi-agent path planning. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 151–162, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[2] S. Bandi and D. Thalmann. Space discretization for efficient human navigation. *Comput. Graph. Forum*, 17(3):195–206, 1998.

[3] O. B. Bayazit, J.-M. Lien, and N. M. Amato. Roadmap-based flocking for complex environments. In *Pacific Conference on Computer Graphics and Applications*, pages 104–115. IEEE Computer Society, 2002.

[4] G. Berseth, M. Kapadia, and P. Faloutsos. Acclmesh: Curvature-based navigation mesh generation. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, MIG '15, pages 97–102, New York, NY, USA, 2015. ACM.

[5] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):7–28, 2004.

[6] B. Chazelle. A theorem on polygon cutting with applications. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 339–349, Washington, DC, USA, 1982. IEEE Computer Society.

[7] S. Chenney. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, pages 233–242, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[8] E. Coumans. Bullet physics library. `http://bulletphysics.org/`, 2013.

[9] S. Curtis, J. Snape, and D. Manocha. Way portals: efficient multi-agent navigation with line-segment goals. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 15–22, New York, NY, USA, 2012. ACM.

[10] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 942–947. AAAI Press, 2006.

[11] D. H. Douglas and T. K. . Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[12] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH, 2006.

[13] J. Fernandez, L. Canovas, and B. Pelegrin. Algorithms for the decomposition of a polygon into convex polygons. *European Journal of Operational Research*, 121(2):330–342, March 2000.

[14] J. Fernández, B. Tóth, L. Cánovas, and B. Pelegrín. A practical algorithm for decomposing polygonal domains into convex polygons by diagonals. *TOP*, 16(2):367–387, 2008.

[15] R. C. Franco Tecchia, Céline Loscos and Y. Chrysanthou. Agent behaviour simulator (abs): A platform for urban behaviour development. In *In GTEC'2001*, pages 17–21, 2001.

[16] F. Garcia, M. Kapadia, and N. Badler. Gpu-based dynamic search on adaptive resolution grids. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 1631–1638, May 2014.

[17] R. Geraerts. Explicit corridor map. `http://www.staff.science.uu.nl/~gerae101/motion_planning/ecm.html`, 2010.

[18] R. Geraerts. Planning short paths with clearance using explicit corridors. In *IEEE International Conference on Robotics and Automation, (ICRA)*, pages 1997–2004. IEEE, 2010.

[19] R. Geraerts and M. Overmars. A comparative study of probabilistic roadmap planners. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 43–57, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[20] R. Geraerts and M. Overmars. The corridor map method: A general framework for real-time high-quality path planning: Research articles. *Comput. Animat. Virtual Worlds*, 18(2):107–119, May 2007.

[21] R. Geraerts and M. Overmars. Enhancing corridor maps for real-time path planning in virtual environments. pages 64–71, 2008.

[22] M. Haciomeroglu, R. G. Laycock, and A. M. Day. Distributing pedestrians in a virtual environment. In *Proceedings of the 2007 International Conference on Cyberworlds*, CW '07, pages 152–159, Washington, DC, USA, 2007. IEEE Computer Society.

[23] D. H. Hale and G. M. Youngblood. Full 3d spatial decomposition for the generation of navigation meshes. In C. Darken and G. M. Youngblood, editors, *AIIDE*. The AAAI Press, 2009.

[24] H. D. Hale, M. G. Youngblood, and P. N. Dixit. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. *Artificial Intelligence and Interactive Digital Entertainment AIIDE*, pages 173–178, 2008.

[25] D. Harabor and A. Botea. Hierarchical path planning for multi-size agents in heterogeneous environments. In P. Hingston and L. Barone, editors, *CIG*, pages 258–265. IEEE, 2008.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[27] D. Haumont, O. Debeir, and F. X. Sillion. Volumetric cell-and-portal generation. *Comput. Graph. Forum*, 22(3):303–312, 2003.

[28] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl.*, 4(2):63–97, June 1994.

[29] S. Hert. Cgal 4.8.1 - 2d polygon partitioning. `http://doc.cgal.org/latest/Partition_2/index.html`, 2016.

[30] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 207–218, London, UK, UK, 1983. Springer-Verlag.

[31] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[32] R. Holte, T. Mkadmi, R. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence (AIJ)*, 85:321–361, 1996.

[33] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical a: Searching abstraction hierarchies efficiently. In W. J. Clancey and D. S. Weld, editors, *AAAI/IAAI, Vol. 1*, pages 530–535. AAAI Press / The MIT Press, 1996.

[34] C.-J. Jorgensen and F. Lamarche. From geometry to spatial reasoning: automatic structuring of 3d virtual environments. In *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 353–364, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] M. Kallmann. Path planning in triangulations. In *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 49–54, Edinburgh, Scotland, July 31 2005.

[36] M. Kallmann. Navigation queries from triangular meshes. In *Proceedings of the Third international conference on Motion in games*, MIG'10, pages 230–241, Berlin, Heidelberg, 2010. Springer-Verlag.

[37] M. Kallmann. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 159–168, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[38] M. Kallmann. Dynamic and robust local clearance triangulations. *ACM Trans. Graph.*, 33(5):161:1–161:17, Sept. 2014.

[39] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained delaunay triangulations. In G. Brunnett, B. Hamann, H. Mueller, and L. Linsen, editors, *Geometric Modeling for Scientific Visualization*, pages 241–257. Springer-Verlag, Heidelberg, Germany, 2003. ISBN 3-540-40116-4.

[40] A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. In R. Boulic and D. K. Pai, editors, *Symposium on Computer Animation*. The Eurographics Association, 2004.

[41] M. Kapadia, F. M. Garcia, C. D. Boatright, and N. I. Badler. Dynamic search on the GPU. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, pages 3332–3337, 2013.

[42] M. Kapadia, K. Ninomiya, A. Shoulson, F. Garcia, and N. Badler. Constraint-aware navigation in dynamic environments. In *Proceedings of Motion on Games*, MIG '13, pages 89:111–89:120, New York, NY, USA, 2013. ACM.

[43] I. Karamouzas, R. Geraerts, and M. Overmars. Indicative routes for path planning and crowd simulation. In *Proceedings of the 4th International Conference on the Foundations of Digital Games*, pages 113–120, 2009.

[44] J. M. Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985.

[45] S. Koenig and M. Likhachev. D*lite. In *Eighteenth National Conference on Artificial Intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[46] F. Lamarche. Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. *Comput. Graph. Forum*, 28(2):649–658, 2009.

[47] R. Lawrence and V. Bulitko. Database-driven real-time heuristic search in video-game pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games*, 5(3):227–241, 2013.

[48] D.-T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[49] W. Lee and R. Lawrence. Trading space for time in grid-based path finding. In M. desJardins and M. L. Littman, editors, *AAAI*. AAAI Press, 2013.

[50] A. Lerner, Y. Chrysanthou, and D. Cohen-Or. Efficient cells-and-portals partitioning: Research articles. *Comput. Animat. Virtual Worlds*, 17(1):21–40, Feb. 2006.

[51] J.-M. Lien, S. Rodríguez, J.-P. Malric, and N. M. Amato. Shepherding behaviors with multiple shepherds. In *ICRA*, pages 3402–3407. IEEE, 2005.

[52] M. Likhachev, D. Ferguson , G. Gordon, A. T. Stentz, and S. Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 262–271, June 2005.

[53] M. Likhachev, G. J. Gordon, and S. Thrun. Ara: Anytime a with provable bounds on sub-optimality. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *NIPS*, pages 767–774. MIT Press, 2003.

[54] Y.-H. Liu and S. Arimoto. Finding the shortest path of a disc among polygonal obstacles using a radius-independent graph. *IEEE Transactions on Robotics and Automation*, 11(5):682–691, 1995.

[55] R. Lopez-Padilla, R. Murrieta-Cid, and S. M. LaValle. Optimal gap navigation for a disc robot. In *Algorithmic Foundations of Robotics X*, pages 123–138. Springer, 2013.

[56] C. Loscos, D. Marchal, and A. Meyer. Intuitive crowd behaviour in dense urban environments using local laws. In *TPCG*, pages 122–129. IEEE Computer Society, 2003.

[57] J.-c. L. Lydia Kavraki, Petr Svestka and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE International Conference on robotics and automation*, pages 566–580, 1996.

[58] M. Maria, S. Horna, and L. Aveneau. Constrained convex space partition for ray tracing in architectural environments. In *Computer Graphics Forum*. Wiley Online Library, 2016.

[59] M. Mononen. Recast navigation toolkit. `http://code.google.com/p/recastnavigation/`, 2009.

[60] A. C. I. Norman Jaklin and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds*, 5(24):285–295, 2013.

[61] NVIDIA. Cuda. a parallel computing architecture developed by nvidia for graphics processing. `http://www.nvidia.com/object/cuda_home_new.html`, 2007.

[62] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[63] R. Oliva and N. Pelechano. Automatic generation of suboptimal navmeshes. In *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 328–339, Berlin, Heidelberg, 2011. Springer-Verlag.

[64] R. Oliva and N. Pelechano. A generalized exact arbitrary clearance technique for navigation meshes. In *Proceedings of Motion on Games*, MIG '13, pages 103–110, New York, NY, USA, 2013. ACM.

[65] R. Oliva and N. Pelechano. Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments. *Computers & Graphics*, 37(5):403–412, 2013.

[66] R. Oliva and N. Pelechano. Clearance for diversity of agents' sizes in navigation meshes. *Computers & Graphics*, 47:48–58, 2015.

[67] N. Pelechano, J. M. Allbeck, and N. I. Badler. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '07, pages 99–108, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[68] J. Pettre, J.-P. Laumond, and D. Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. *First International Workshop on Crowd Simulation*, 43(44):194, 2005.

[69] J. Pettre, J. P. Laumond, and D. Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *Proceedings of the 1st International Workshop on Crowd Simulation*, pages 81–90, 2005.

[70] J. Pettre and D. Thalmann. Path planning for crowds: From shared goals to individual behaviors. pages 45–48, 2005.

[71] M. Qi, T.-T. Cao, and T.-S. Tan. Computing 2d constrained delaunay triangulation using the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 39–46, New York, NY, USA, 2012. ACM.

[72] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *j-CGIP*, 1(3):244–256, nov 1972.

[73] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, Aug. 1987.

[74] C. W. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference 1999*, GDC '99, pages 763–782, San Francisco, California, 1999. Miller Freeman Game Group.

[75] S. Rodriguez and N. M. Amato. Roadmap-based level clearing of buildings. In *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 340–352, Berlin, Heidelberg, 2011. Springer-Verlag.

[76] J. B. Roerdink and A. Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundam. Inf.*, 41(1,2):187–228, Apr. 2000.

[77] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, Oct. 1966.

[78] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In N. J. Nilsson, editor, *IJCAI*, pages 412–422. William Kaufmann, 1973.

[79] J. Snape, J. van den Berg, S. J. Guy, and D. Manocha. The hybrid reciprocal velocity obstacle. *Trans. Rob.*, 27(4):696–706, Aug. 2011.

[80] G. Snook. Simplified 3d movement and pathfinding using navigation meshes. In *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[81] A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2:127–145, 1995.

[82] N. Sturtevant. Memory-efficient pathfinding abstractions. In *AI Programming Wisdom 4*. Charles River Media, 2008.

[83] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pages 1392–1397. AAAI Press, 2005.

[84] N. R. Sturtevant and R. Geisberger. A comparison of high-level approaches for speeding up pathfinding. In G. M. Youngblood and V. Bulitko, editors, *AIIDE*. The AAAI Press, 2010.

[85] N. R. Sturtevant and M. R. Jansen. An analysis of map-based abstraction and refinement. In I. Miguel and W. Ruml, editors, *SARA*, volume 4612 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2007.

[86] A. Sud, E. Andersen, S. Curtis, M. Lin, and D. Manocha. Real-time path planning for virtual agents in dynamic environments. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 55:1–55:9, New York, NY, USA, 2008. ACM.

[87] A. Sud, R. Gayle, E. Andersen, S. Guy, M. Lin, and D. Manocha. Real-time navigation of independent agents using adaptive roadmaps. In *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, VRST '07, pages 99–106, New York, NY, USA, 2007. ACM.

[88] N. I. B. Tianyu Huang, Mubbasir Kapadia and M. Kallmann. Path planning for coherent and persistent groups. In *Proceedings of the IEEE International Conference on Robtics and Automation*, ICRA '14. IEEE, 2014.

[89] P. Tozour. Building a near-optimal navigation mesh. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[90] UDK. Unreal navmesh generator. `http://udn.epicgames.com/Three/NavigationMeshReference.html`, 2004.

[91] Valve. Valve navmesh generator. `http://developer.valvesoftware.com/wiki/Navigation_Meshes`, 2005.

[92] J. van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *2008 IEEE International Conference on Robotics and Automation*, pages 1928–1935. IEEE, May 2008.

[93] J. van den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin. Interactive navigation of multiple agents in crowded environments. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 139–147, New York, NY, USA, 2008. ACM.

[94] W. van Toll, A. F. C. IV, and R. Geraerts. Navigation meshes for realistic multi-layered environments. In *IROS*, pages 3526–3532. IEEE, 2011.

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

[96] W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts. A comparative study of navigation meshes. In *Submitted to Motion in Games*, pages 1–12. ACM, 2016.

[97] N. M. Wardhana, H. Johan, and H. S. Seah. Subregion graph: A path planning acceleration structure for characters with various motion types in very large environments. *Computational Visual Media*, 1(2):105–118, 2015.

[98] R. Wein, J. P. Van Den Berg, and D. Halperin. The visibility–voronoi complex and its applications. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 63–72. ACM, 2005.

[99] S. Zlatanova, L. Liu, and G. Sithole. A conceptual framework of space subdivision for indoor navigation. In *Proceedings of the Fifth ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*, ISA '13, pages 37–41, New York, NY, USA, 2013. ACM.