Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master in Computing

Master of Science Thesis

ANAVMG: AUTOMATIC GENERATION OF SUBOPTIMAL NAVMESHES FOR 3D VIRTUAL ENVIRONMENTS

Student: Ramon Oliva Martínez

Advisor: Nuria Pelechano Gómez

Barcelona, September 5, 2012

Content

1 INTRODUCTION ... 4

1.1 ORGANIZATION ... 5

2 OBJECTIVES ... 6

3 STATE OF THE ART ... 7

3.1 ROADMAPS ... 7
3.2 CELL DECOMPOSITION (NAVIGATION MESHES) ... 8
3.2.1 Fixed Cell Type NavMeshes ... 8
3.2.2 Arbitrary Cell Type NavMeshes ... 13
3.3 CONCLUSIONS ... 18

4 AUTOMATIC GENERATION OF NEAR-OPTIMAL NAVIGATION MESHES ... 19

4.1 THE GPU BASED APPROACH ... 20
4.2 THE PORTAL VERTEX-PORTAL CASE ... 22
4.3 CONVEXITY RELAXATION ... 23

5 SINGLE-LAYERED ENVIRONMENTS ... 26

5.1 NORMAL-AND-DEPTH MAP EXTRACTION ... 26
5.2 OBSTACLE DETECTION ... 27
5.3 CONTOUR EXPANSION AND REFINEMENT ... 28
5.4 POLYGON RECONSTRUCTION AND SIMPLIFICATION ... 28

6 MULTI-LAYERED ENVIRONMENTS ... 30

6.1 FIRST APPROACH: DYNAMIC CUTTING PLANES ... 30
6.2 SECOND APPROACH: ADAPTATIVE CUTTING SHAPE ... 31
6.2.1 Rough Voxelization ... 32
6.2.2 Layer Extraction and Labeling ... 33
6.2.3 Layer Refinement ... 35
6.2.4 NavMesh Merging ... 39

7 IMPLEMENTATION ... 41

7.1 DESIGN ... 41

8 RESULTS ... 43

8.1 CPU VS GPU VERSION ... 43
8.2 MULTI-LAYERED ENVIRONMENTS ... 45
8.3 ANAVMG VS RECAST ... 50

9 CONCLUSIONS AND FUTURE WORK ... 52

9.1 PERSONAL CONCLUSIONS ... 53

10 BIBLIOGRAPHY ... 54

1 Introduction

A popular solution to solve the problem of navigation in a complex scene, consists of subdividing the scene into convex regions (cells) forming what is commonly known as a *Navigation Mesh* (*NavMesh*). A *Cell-and-Portal Graph* (CPG) is then created where a node of the graph corresponds to a convex region of the *NavMesh* and a portal is an edge shared by two cells. Path-finding can then be solved using an algorithm such as A*.

Although *NavMeshes* are widely used on complex applications such as videogames and virtual simulations, there are not many applications to automatically generate a *NavMesh* appropriate for path planning, so often either the user need to refine those semi-automatic *NavMeshes*, or create them by hand from scratch which is extremely time consuming and a source of errors.

There is therefore a need for automatic methods to generate *Cell-and-Portal Graphs* for navigation, from any given 3D environment with minimum user input required. This thesis focuses on solving such a complex problem. The work presented in this master thesis gets as an input any 3D virtual environment represented by a polygon soup, and provides as an output the complete *CPG.*

The main contribution of this thesis is a novel GPU based method to generate a *NavMesh* for a given 3D scene. 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 several layers of a single simple polygon (floor) that can contain holes (obstacles). Secondly, it automatically generates a suboptimal convex decomposition of this 2D representation which represents the CPG. Our method is robust against degeneracies of the starting 3D model, such as interpenetrating geometry.

In previous work [24], we presented a method to automatically generate near-optimal *NavMeshes* in the CPU from a 2D floor plan represented by a simple polygon with holes. The method, entitled ANavMG, calculates for every notch (concave vertex) of the scene, the closest element that lies on the area formed by the prolongation of the edges incident to the notch, and creates a portal between the notch and the closest element, which can be a vertex, an edge or a previously created portal. The main target of this algorithm is to create a near-optimal convex decomposition of the scene representing a *CPG.*

In this master's thesis we also improved the creation of CPG, with an efficient and robust GPU approach to speed up the step of searching for the closest element, based on performint renders of the scene for each notch with the camera parameters defined by characteristics of the notch. The new method not only has higher performance values which allow the user to have very large environments, but it is also more robust since it overcomes several problems that ANavMG presented. Unlike the work presented by [24] which created *NavMeshes* for 2D floor plans, this new method can also deal with complex 3D multi-layered environments.

Therefore, this master thesis completes and improves the work that I started during my final degree project, solving the complex problem of generating robust and accurate CPGs with as few cells as possible from any 3D virtual environment.

1.1 Organization

The rest of the document is organized as follows. In the next chapter we present the objectives of this master thesis. Chapter 3 contains the study of the state of the art in the field of navigation of autonomous characters in complex virtual environments. In chapter 4, we briefly describe the core algorithm of our system to generate a *Navigation Mesh*, and we introduce a new GPU based version of the same algorithm. Chapter 5 describes the automatic system that, given a 3D scene representing a single floor plan, is able to obtain the 2D polygonal information of the scene required by the *NavMesh* generation step, and Chapter 6 describes our approach to extend this idea to multi-layered environments. Chapter 7 briefly describes the implementation design of the application. The results obtained with our system are explained in chapter 8. Finally, in chapter 9 we can find our conclusions and future work.

2 Objectives

Navigation of autonomous characters in complex virtual environments, avoiding collisions in a natural manner with static and dynamic obstacles, remains still an open problem and it is of main interest in areas involving both Computer Graphics and Artificial Intelligence, such as crowd simulation, videogames, movies or robotics. In some of these applications, often real-time response is required for large numbers of characters. Creating *Navigation Meshes* with as few cells as possible is important because it allows to speed up the path finding process. More importantly, having an automatic method to generate those *Navigation Meshes* from any given 3D geometry can help developers to create new scenarios or modify existing ones without extra time consuming work. Therefore, the main purpose of this thesis has been the introduction of a new technique to solve the problem of navigation in complex virtual environments generating as few cells as possible.

In order to achieve our goal, we first studied the previous work that has been done in the field on navigation of autonomous characters in complex virtual environments. Specifically, we have focused on the technique based in the use of *Navigation Meshes* that is a partition of the walkable space into convex regions.

In our previous work [24], we presented a method to automatically generate a *Navigation Mesh* for a complex environment in 2D, represented as a simple polygon (the walkable area) that contains holes (the static obstacles of the geometry). The main target was to generate a partition of the scene with as few cells as possible. In this thesis, we have firstly introduced a technique to automatically obtain the 2D floor plan of a 3D model, representing a single floor, secondly we developed a GPU version of the algorithm presented in [24], that greatly speeds-up the execution time of creating the convex decomposition, and thirdly we present a novel algorithm to extract the different layers forming a complex 3D environment.

3 State of the Art

The determination and representation of the free space of a virtual environment is a central problem in the fields of robotics, videogames and crowd simulation. Two general approaches exist in order to represent the free space of a scene: roadmaps and cell decomposition. The main objective of both methods is to generate a graph that is used later for a search algorithm (usually the A*) to find a path free of obstacles between two points of the scene.

3.1 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 problem of visibility graph representation is that it is only usable on very specific cases. In addition, the quality of the paths as well as the performance of the search algorithm is greatly related to the number and position of the vertices of the visibility graph: 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 specially in applications that requires a real-time response; if we want to improve the performance of the graph search algorithm, the number of vertices spread must be low, but it produces an unnatural look on the movement of the characters, like if they were walking on rails.

An alternative representation for roadmaps is to compute the generalized Voronoi Diagram [23]. An approximation of the generalized Vornoi Diagram can be computed using the graphics hardware [14]. The property of roadmaps generated by this way is to maximize the clearance with obstacles. In [32], 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 1 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.

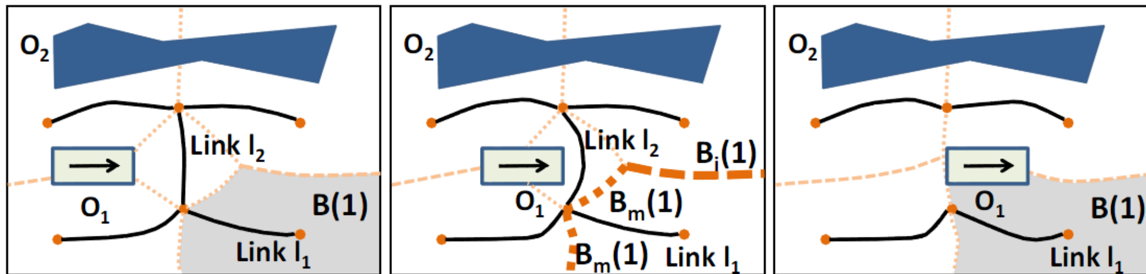

Figure 1: *The obstacle O_1 is moving towards link l_2 (left) and it is deformed (center). When the elasticity of the link is exceeded, the link l_2 is removed (right)*

3.2 Cell Decomposition (Navigation Meshes)

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 [32]. The Cell Decomposition method consists on 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*) [31] 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 [28] 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.

3.2.1 Fixed Cell Type NavMeshes

Uniform grids have been proposed as a cell decomposition of the environment [2][19]. It is an easy and fast solution to obtain a convex decomposition, but the main problem of this technique is that a high-density CPG is generated, increasing the time of the search algorithm.

Valve's Game Engine has an automatic *NavMesh* generator method based on subdividing the virtual map by Axis-Aligned quads of arbitrary size [36]. This method gives only satisfactory results in very specific scenes with Axis-Aligned obstacles, such as some indoor scenes. But in most cases, with complex obstacles randomly distributed and oriented, the partition obtained is useless because it does not adapt well to the contour of the obstacles, as can be seen in figure 2. In addition, if the environment contains very steep stairs, ramps or hills, the generator system makes errors, resulting in a *NavMesh* that does not cover the entire map. So it is necessary to manually complete the *NavMesh*.

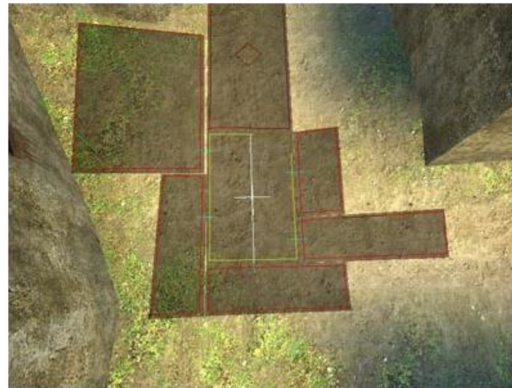

Figure 2: *Axis-Aligned quads do not adapt well to the contour of general obstacles.*

Triangular Meshes are commonly used to represent a *Navigation Mesh*. In [18][15], 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 [18][15] 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 step consists in computing the shortest path joining the starting and the goal points inside the channel. For this, the *funnel algorithm* [3][12][21] is applied. The main feature of the *NavMesh* proposed is the support for dynamic obstacles, but the results show that 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. So in fact, interactive rates can only be obtained if the CDT and the constraints are simple enough and therefore, the method proposed seems not to fit with the requirements of complex applications such as videogames, with many movable obstacles and high frame-rate requirements.

In [5] the CDT technique is compared against grid-based maps of real commercial videogames. 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 [17], 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 [25], a method for computing the CDT using the GPU has been presented. The implementation is done using the CUDA programming model [4] on NVIDIA GPUs and the results show that it runs several times faster than any CPU method.

In [16], 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 3 (left) illustrates this situation.

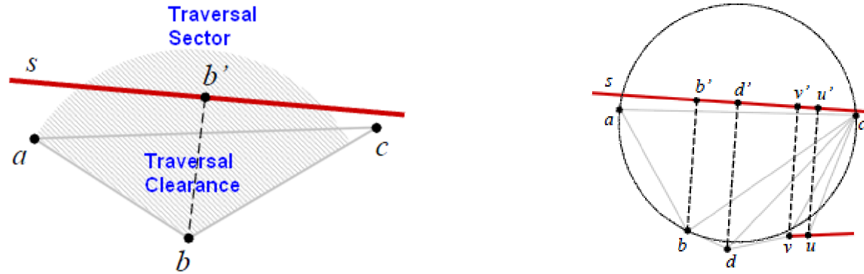

Figure 3: *The traversal sector and the traversal clearance of a triangle traversal* τ_{abc} *(left); the vertex v is a disturbance of the traversal* τ_{abc} *(right).*

Given the situation depicted on the figure 3 (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 path 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 4. 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.

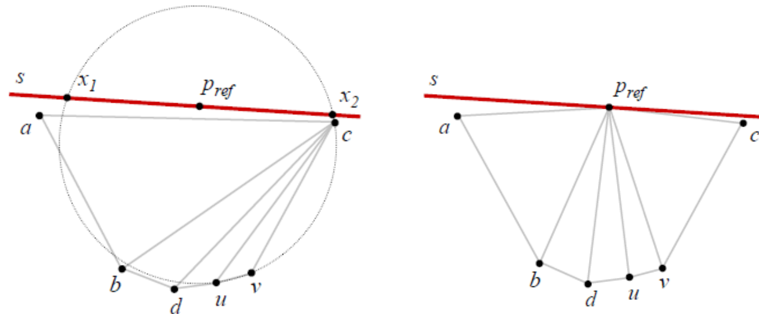

Figure 4: *The point of refinement P_{ref} is computed as the midpoint of the intersection points with the constrained edge s and the circle(u,v,c) (left); once P_{ref} has been computed, all the vertices are joined to this new point (right).*

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 problem 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. Moreover, it is not demonstrated that the iterative process used to compute the LCT from the CDT always 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 [20] 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 environment by a set of vertical 3D prisms dividing the 3D model into layers. The workflow of the algorithm is described in figure 5. 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.

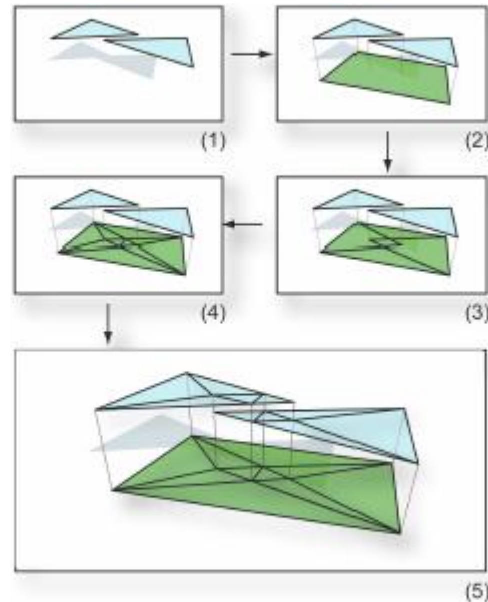

Figure 5: *Different steps of the subdivision into prisms of the 3D model*

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 be wrong. And finally, the method proposed is extremely 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.

3.2.2 Arbitrary Cell Type 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. [22] 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 postprocessing step to convert the resulting cells into convex, for example, using the *Hertel-Mehlhorn* method [13] that is used to decompose a simple polygon without holes into convex regions.

In [34], 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 [13] 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 6. In addition, the subdivision process generates many little sub-cells that cannot be easily merged during the merging process, resulting on an over-segmented partition of the scene.

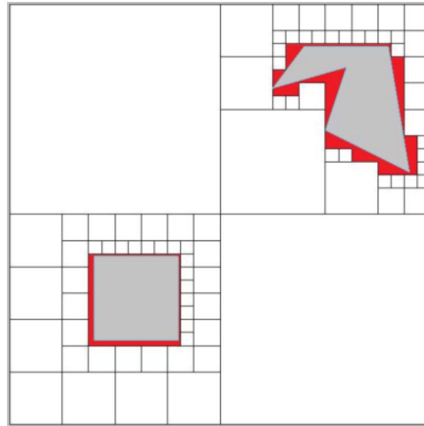

Figure 6: *The subdivision process applied with two different obstacles. In red it is marked the walkable space that it is discarded due to the subdivision method does not adapts well to the shape of the obstacles.*

In [10], 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-Alligned 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 [9], but it has the same limitations than the 2D version.

Toll et al. [33] 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. Also, this technique creates many degenerated cells, which can

introduce artifacts on the movement of the virtual characters. An approximation of the medial axis set can be computed using the GPU, as described in [14]. The implementation of this *NavMesh* generation method, restricted to one single layer, can be found in [8]. It requires to manually creating a file that describes the contour of the obstacles, so the process is not fully automatic.

Unreal Engine [35] 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 7 (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 7 (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.

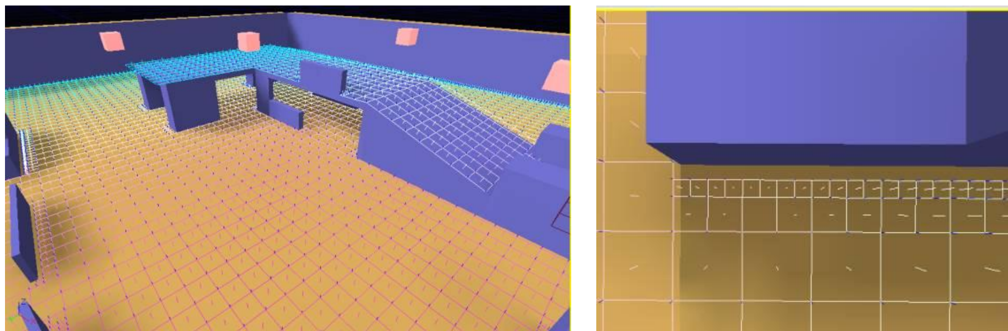

Figure 7: *The resulting high density grid (Left). The cell size is adapted to fit with the shape of the obstacles (Right).*

Recast [27] is an automatic open-source *NavMesh* generator broadly used in popular videogames and other complex virtual applications. The method used by Recast is inspired by the work by Haumont et al. [11]. Recast computes a partition of the scene by applying the *Watershed Transform* (WST) [29] on the *Distance Map Field* [30] of the scene. Figure 8 illustrates this idea.

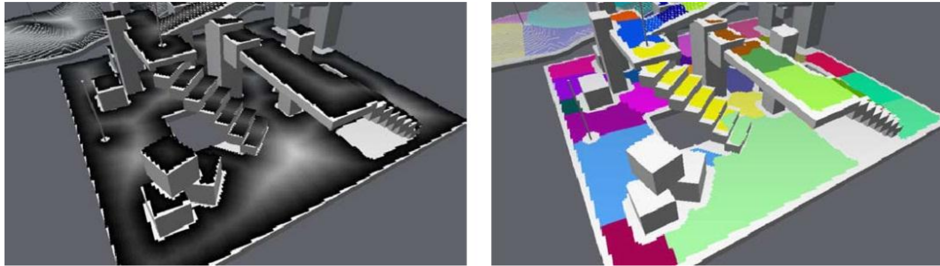

Figure 8: *The Distance Map Field of a scene (left) and the resulting partition after applying Watershed (right).*

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 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). On the other hand, the main drawback is that it generates walkable regions on zones where a character is not allowed to walk, because it only takes into account the characteristics of the geometry enclosed by the voxels, but not the connectivity between potentially walkable regions. It is a problem because increases unnecessarily the size of the resulting CPG with cells that never will be used. Some of those regions can be removed if they are small enough tuning the parameters of the application but it is not always possible to remove all of them. Another problem is that the partition obtained is over-segmented, as illustrated on figure 9. 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.

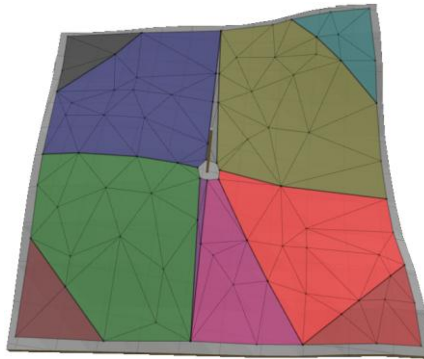

Figure 9: *A partition (colored cells) obtained using Recast on a very simple scene in which however, suffers from over-segmentation.*

3.3 Conclusions

As mentioned in the previous chapter, our main objective is to solve the problem of navigation of autonomous characters in complex virtual environments, that is a central problem in the fields of robotics, videogames and crowd simulation. The limitations of the previous exposed methods that address this problem, have pushed us to develop a new approach.

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, the avoidance of collisions with dynamic obstacles is usually a hard task, and not always possible, so the character can get stuck in local minima of the geometry.

The cell decomposition technique (*Navigation Mesh*) fits better with our requirements as it provides a more accurate description of the scene, but the methods studied suffers mainly of over-segmentation, ill-conditioned cells or does not adjust well to the contour of the obstacles, missing some portions of the walkable space. In this work, we will further extend the functionalities of our previous work [24], adding support for multi-layered 3D virtual environments. Currently, our method is able to automatically generate the partition of a 3D scene representing several floors, and the partition obtained is near-optimal as well as it avoids in most cases the creation of ill-conditioned cells.

4 Automatic Generation of Near-Optimal Navigation Meshes

In [24], we proposed a new algorithm to create a *NavMesh* by calculating a convex partition of a 2D description of the scene represented by the floor as a simple polygon without intersections, and the static obstacles as being holes of this polygon. We demonstrated that the convex partition obtained is near-optimal. Before getting to the details of the work presented in this thesis, we will give a brief summary of the algorithm to calculate the convex decomposition since it will also be used in this work, although with an important number of improvements that will be explained in this section. We refer the reader to [24] for a more in depth explanation of the basic algorithm. The method, entitled *ANavMG*, calculates for every notch (concave vertex) its *Area of Interest* that is the area of the polygon delimited by the prolongation of the edges incident to the notch (figure 9). The closest element to the notch inside its *Area of Interest* is then computed and a portal is created to convert the notch into a convex vertex. The closest element to the notch can be another vertex, an edge of the geometry or a previously created portal. Each of these cases needs to be treated differently.

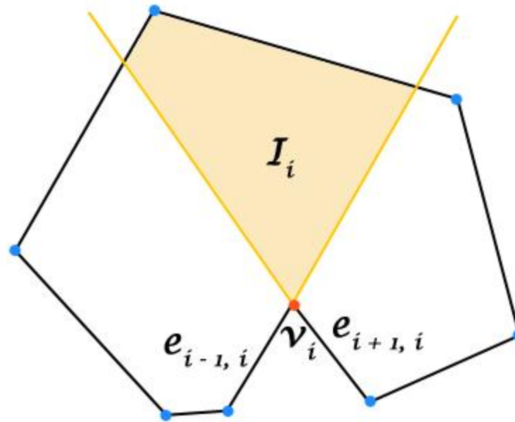

Figure 9: *The Area of Interest (I_i) of a notch v_i.*

If the closest element to a notch *n* is a vertex *v*, a *vertex-vertex* portal is created by creating a portal edge *p* joining *n* and *v*. In addition, if *v* is also a notch, then the algorithm also checks whether by creating portal *p*, the notch *v* gets split into two convex angles. This will happen exclusively when *n* falls within the *Area of Interest* of *v*.

If the closest element to a notch *n* is an edge of the geometry *e*, a *vertex-edge* portal is created. In this case, a portal joining the notch with a point over the edge is created. Five point candidates over the edge are computed: The projection of *n* over *e*, the intersection points between the lines supporting the edges defining the *Area of Interest* of *n* and the edge *e*, and the endpoints of the edge. The best candidate is the closest to the notch *n* that lies inside the *Area of Interest* of *n* and between the endpoints of the edge *e*. Once the best candidate is determined, it is used to subdivide the edge (if the best candidate is not one of its endpoints) and it proceeds as in the case of a *vertex-vertex* portal.

Finally, if the closest element to a notch *n* is a previously created portal *p*, a *vertex-portal* portal is created. The treatment differs from the previous case since we do not want to have intersecting portals (or *T*-shapes). Therefore, a portal is created with the closest endpoint of the portal that lies inside the *Area of Interest* of the notch. If neither of the endpoints of the portal

lies inside the *Area of Interest*, it is necessary to create two portals to convert the notch into a convex vertex. In this case, we create two portals by joining the notch with both endpoints of the closest portal.

In most cases, the method described above creates a single portal to partition each notch into a convex vertex, and in some situations, a portal is enough to split two notches at the same time, as described in the *vertex-vertex* case. Only in the case that the closest element to the notch is a portal and if none of its endpoints is inside the *Area of Interest* of the notch, two portals need to be created, as we want to avoid *T-shapes*. Therefore, the obtained partition is near-optimal (proof explained in [24]). The main limitation is that it only works for 2D environments.

4.1 The GPU based approach

The previously described algorithm 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 videogames.
During the first stage of the development of the thesis, we realized that we could easily adapt the algorithm to exploit the efficiency of the GPU. The new method based on GPU, starts by assigning a unique color identifier to each edge of the scene, which will be used for rendering and identification purposes. 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 10.

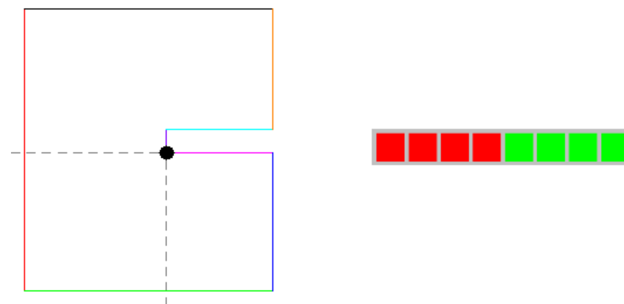

Figure 10: *A simple scene with all edges drawn with a unique color (Left) and the texture generated from the point of view of the notch (Right).*

To recover the edges that are 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. A variable *zFarScene* contains the average of the distances to the closest element of the already visited notches. Initially, *zFarScene* is set to 1/10th of the diagonal of the bounding box of the scene. Then for each notch, a render is performed with *zFar* set to *zFarScene*. If no element has been rendered with such *zFar*, the *zNear* is set to the current *zFar* and the *zFar* is doubled in order to carry out a new render. This process continues until at least one element has been found that lies in the Area of Interest of the notch. Once the closest element to the notch is found, *zFarScene* is updated accordingly. Notice that this process implies several renders for some notches, but we have found empirically that the *zFarScene* 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 11.

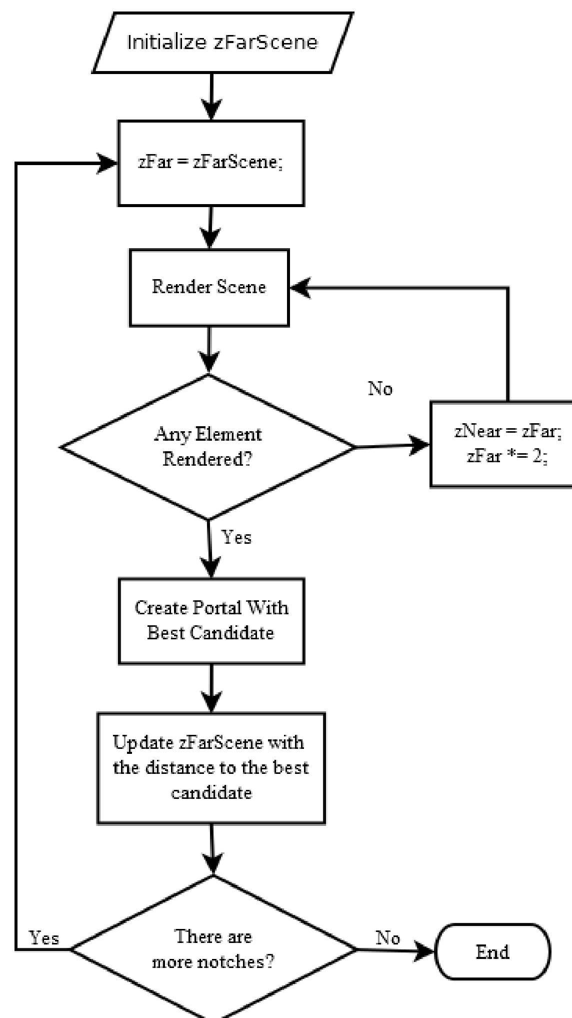

Figure 11: *Diagram describing the dynamic update of the zFar.*

4.2 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 the introduction of *T-Shapes*, the algorithm presented in [24] proposes creating 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. However, we realized that this approach can cause intersection with the existing geometry, when the endpoints are not actually visible from the notch (see figure 12). Therefore, we have modified the previous algorithm so when a portal is created with another portal, we check for intersections between the segment formed by the notch and the endpoint of the portal, against 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 new found closest edge.
In the GPU version of the algorithm, 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 12.

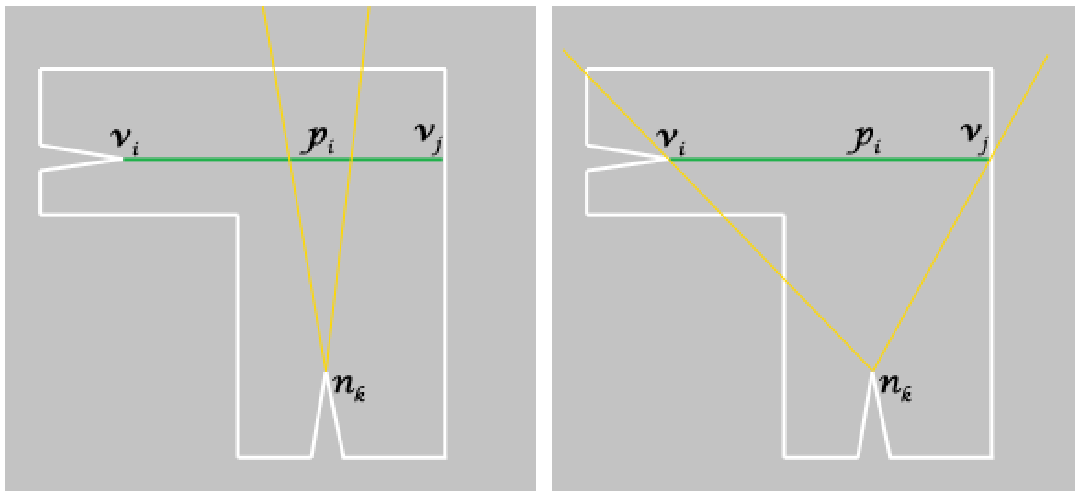

Figure 12: *The original Area of Interest of a notch n_k (Left). When the closest element is a portal p_i, the new Area of Interest is defined by the notch and the endpoints of the portal (Right).*

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 Convexity Relaxation

By definition, a *Navigation Mesh* is a partition of the space into convex cells, i.e., polygons with all its internal angles smaller or equal than π. This is the mathematical definition of convexity, but in the case of *NavMeshes*, it is not necessary to deal with such a strict definition. The movement within a convex cell and between cells is driven by some local movement algorithm, which can usually deal relatively easily with small concavities in the cell through obstacle avoidance behavior. Therefore, depending on the characteristics of the local movement method being used, the definition of convexity can be relaxed. The main advantage of introducing the notion of convexity relaxation is that we can reduce the number of cells.

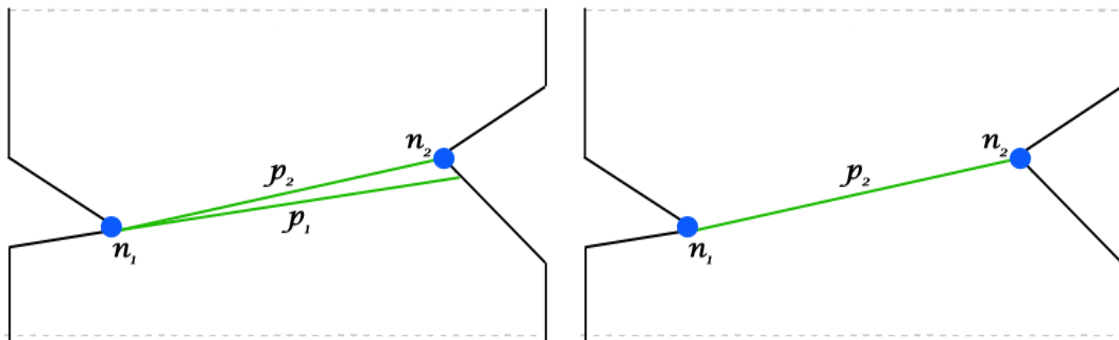

Figure 13: *With a strict definition of convexity, a degenerated cell appear on the partition. This is because the notch n_2 is outside the area of interest of the notch n_2 (Left). The old convexity relaxation definition addressed this problem by increasing the Area of Interest of the notches by a certain threshold τ. By this way, n_2 is inside the Area of Interest of n_1 and a portal vert-vert is created (Right).*

We introduced this idea in our previous paper [24], where we proposed a solution that consisted in increasing the *Area of Interest* of the notch by a certain threshold τ. This 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. Figure 13 describes this situation. Later we realized that this statement is not always certain, and the most important drawback is that due to the increase of the *Area of Interest,* some non ill-conditioned cells can become ill-conditioned cells, as can be seen in figure 14, and it is precisely the situation that we want to avoid.

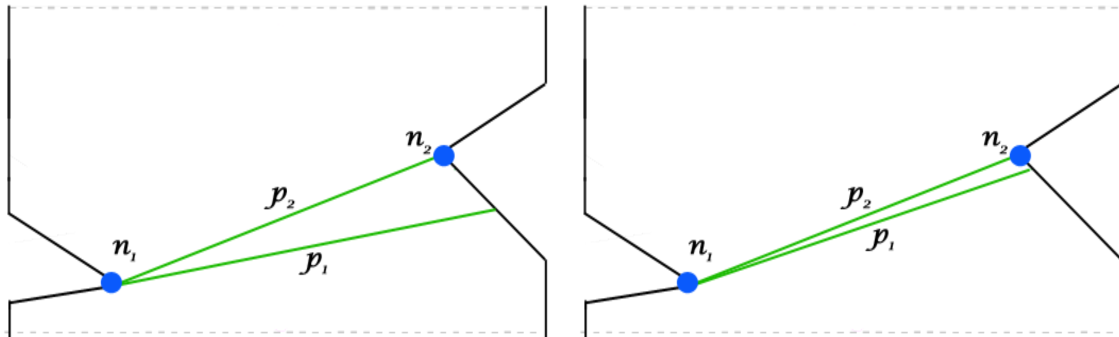

Figure 14: *With a strict definition of convexity, the resulting cells have an area non-close to zero (Left). If we increase the Area of Interest by τ, we obtain an ill-conditioned as it has an area close to zero (Right).*

For the present work, we maintain the concept of convexity relaxation in the sense that we allow creating near-convex cells, but we follow a different approach. Firstly, we realized that the *Ramer-Douglas-Peucker* algorithm [26][6] (that is a tool often used for polygon simplification), could be easily adapted in order to obtain near-convex cells. In a preprocess step, we look for strings of notches on the geometry, i.e., a set of consecutive concave vertices, and we apply the *Ramer-Douglas-Peucker* algorithm to simplify the string of notches. The algorithm requires a threshold distance that the user can modify through the application interface (parameter *convexDistance*). Only the notches that survive to the simplification are considered *real notches;* the rest are ignored as if they were convex vertices. This is extremely useful specially to reduce the number of redundant portals in rounded objects, as can be seen in figure 15.

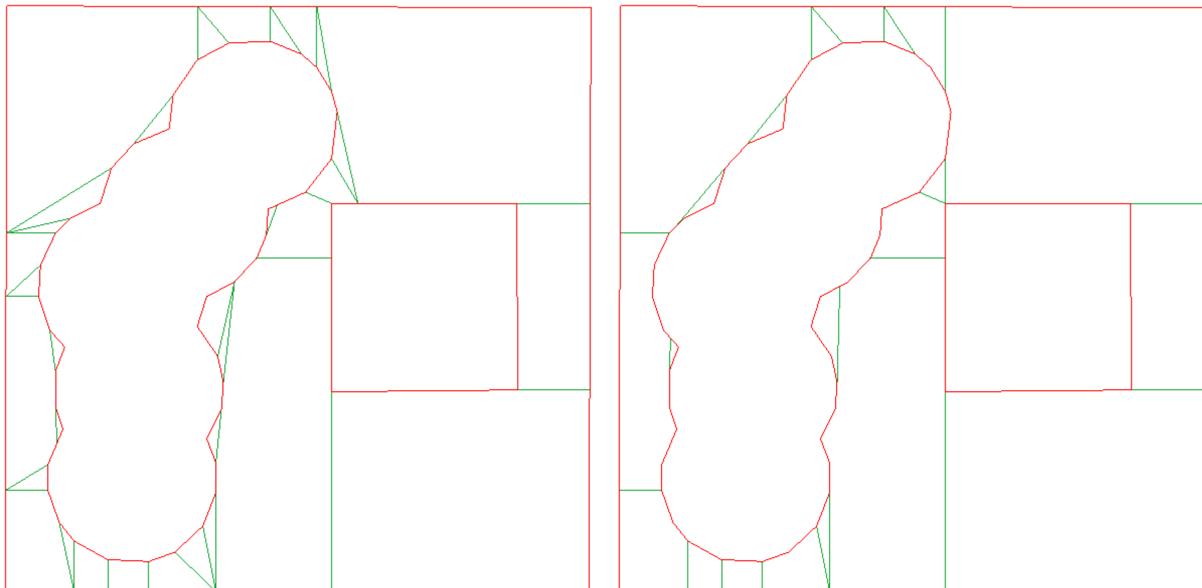

Figure 15: *The CPG of a scene with many rounded objects, applying the strict definition of convexity.It generates a total of 33 cells (Left); for the same scene, with the convexity relaxation definition (setting the convexDistance to 0.4), it generates 21 cells.*

The ill-conditioned cells that can appear during the generation of the *NavMesh,* are treated in post-process. When all the portals have been generated, and just before the reconstruction of the cells, we check for *dispensable* portals. A portal is *dispensable* if removing this portal, the next and the previous edge incident to both endpoints of the portal, forms a near-convex angle.. For example, in figure 13 (Left), the portal p_1 is *dispensable*, and hence, it can be discarded, dealing to the situation described in figure 13 (Right).

5 Single-Layered Environments

As most *NavMesh* generation methods, the *Navigation Mesh* is constructed in 2D. However, especially in the case of videogames, the virtual world is typically generated using a 3D software modeler. Since we want the method to be fully automatic, the second stage of the work of our thesis mainly consisted in the development of a method to transform the 3D input data into a 2D representation. In particular, the input required by the *Navigation Mesh* Generator consists on 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 14.

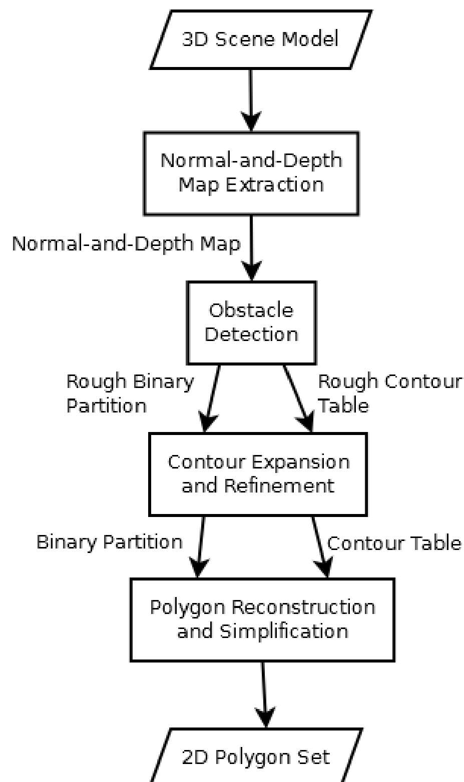

Figure 14: *This figure describes the data flow of the pipeline of the 2D Abstraction step to convert from the 3D world to the 2D representation.*

5.1 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 15 shows the resulting Normal-and-Depth Map for a given scene.

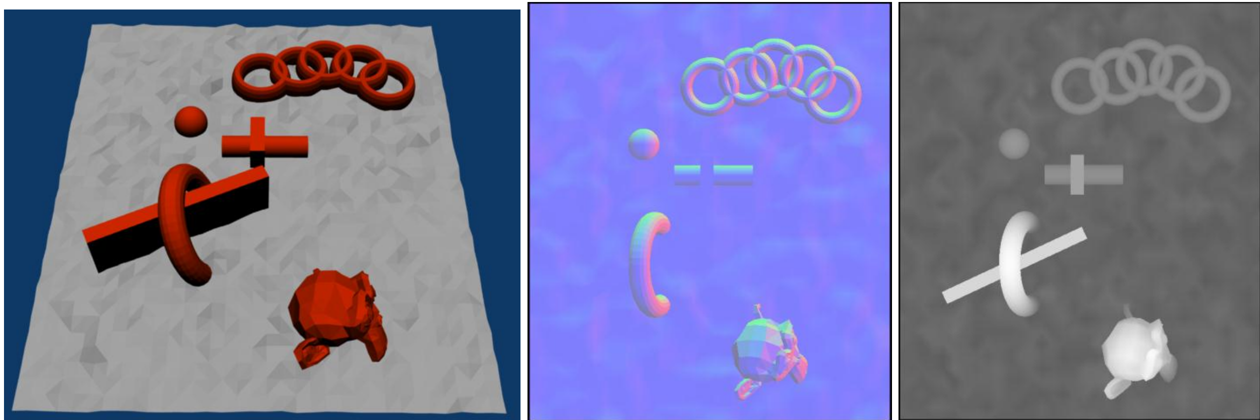

Figure 15: *The initial 3D scene (Left) and the Normal-and-Depth Map generated with the shader (Center & Right).The figure of the center represents the information of the normals, that is stored in the RGB channels of the texture; the right most figure shows the depth information of the scene, stored in the A channel of the texture.*

5.2 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 16 we can see the binary partition with the walkable areas and the obstacles. Notice that the torus is treated as a solid obstacle seen from above and thus the floor underneath it will not be treated as walkable.

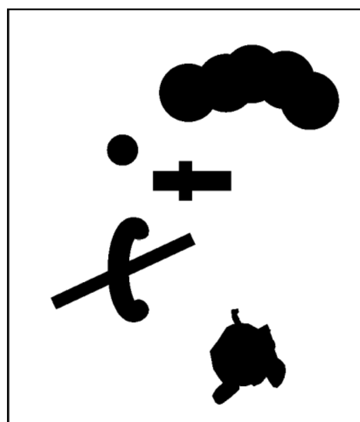

Figure 16: *The Rough Binary Partition resulting of the Obstacle Detection stage.*

5.3 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 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 17 shows these two steps over a given obstacle.

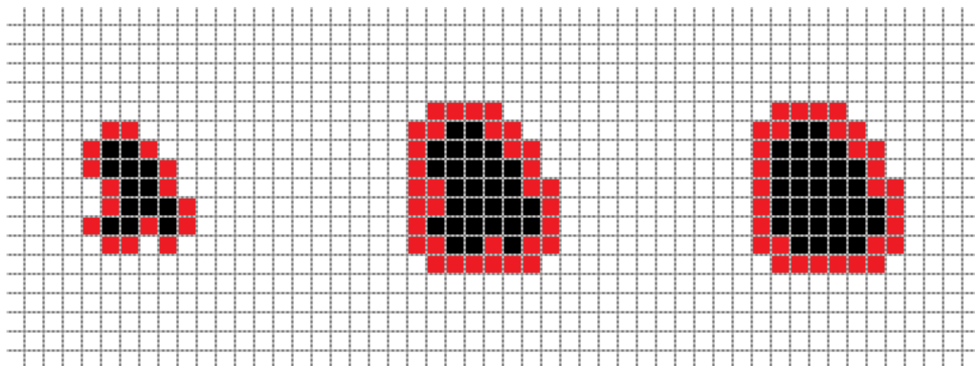

Figure 17: *The initial contour of an obstacle (Left); the expanded contour (Center); the refined contour (Right).*

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

5.4 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 18 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.

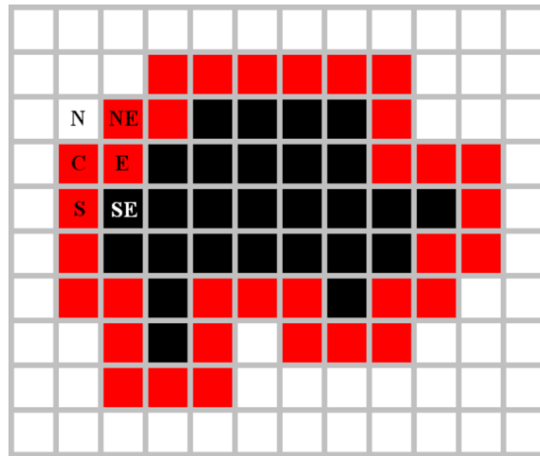

Figure 18: *The most left contour pixel C of an obstacle and the potential neighbors that can be chosen as next.*

To reduce the total number of vertices per polygon, the first straight forward simplification consists on eliminating all vertices that belong to segments aligned horizontally, vertically or with 45º angle and are not end points. This pre-simplification step is done during the reconstruction process. Next, the Ramer-Douglas-Peucker Algorithm [26][6] is applied to further simplify the polygon (figure 19).

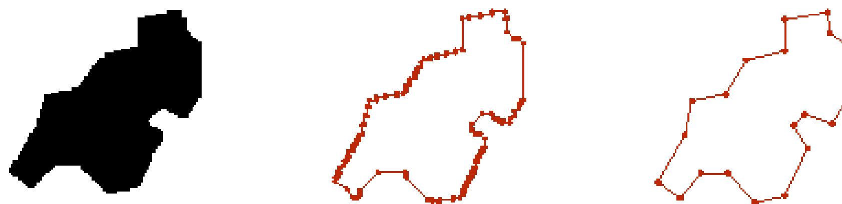

Figure 19: *A polygon on the Binary Partition (Left); A high density pre-simplified polygon (Center); the final simplified polygon (Right).*

6 Multi-Layered Environments

The main limitation of the system proposed above is that it cannot deal with multi-layered environments nor arch-type obstacles. On the other hand, the shape of the obstacles obtained during the generation of the 2D representation, adjust very well to the shape of the original obstacles, contrary to other proposed automatic methods, such as [34][27][36]. The third main stage of the development of the thesis, consisted in extending the proposed method from 3D scenes representing one single layer, to 3D general scenes representing any number of layers.

The main idea is to subdivide the scene into layers. Each individual layer will not have any overlapping geometry, and thus can be "flatten" to obtain its 2D floor plan. Afterwards, the previously described method is applied in order to extract the 2D representation of each layer, that is used to compute its *Navigation Mesh,* using the CPU or GPU version of our algorithm. Once the *NavMesh* of each layer has been computed, the final step is to join these separated meshes into one single *Navigation Mesh* that represents the walkable space of the entire scene.

6.1 First approach: Dynamic Cutting Planes

Since we had a method to compute the 2D floor plan from a 3D model by carrying out a render of the scene, the first idea towards dealing with multi-layered environments was simply to take several renders, one for each layer and later on connecting them through portals. The straight forward method to do this, consisted on adjusting the *zNear* and *zFar* of the camera to make "slides" of the scene. Each one of these slides represents a layer.

However, the main problem of extracting the layers by using the cutting planes of the camera, is that is very hard to determine where to place the zNear and zFar planes to cut the scene properly. In fact, this method would give only satisfactory results in a very determined kind of scenes, such as a building, but it is not really extensible to arbitrarily complex scenes, as can be seen in figure 20. As our main objective was to obtain a method that could deal with any kind of scene, we discarded this idea.

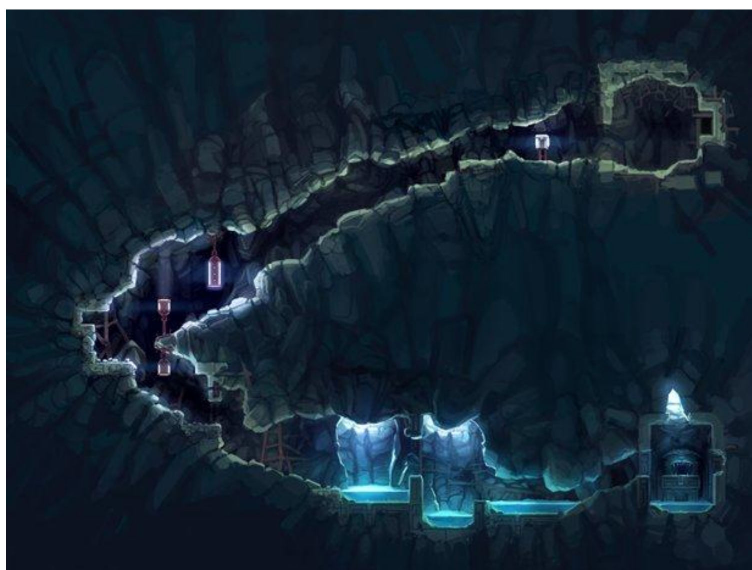

Figure 20: *In a cave-like environment, it is not clear where to set the cutting planes, nor how many cuts are necessary in order to represent the entire scene.*

6.2 Second approach: Adaptative Cutting Shape

The main idea is to construct a shape that adapts to the form of the environment, instead of using planes. The characteristic of this shape is that it can be projected into a 2D image without having any overlapping walkable geometry. This will allow us to implement a new method in the fragment shader which will render only te geometry within some dynamic *Zmin* and *Zmax* depending on the underlying geometry, to generate the 2D floor plan of each layer. The method proposed can be subdivided into these steps: Firstly, a low resolution voxelization is used in order to obtain a first approximation of the geometry that is potentially walkable and the geometry that it is not. Secondly, an ordered flooding process is applied to classify the potentially walkable regions into layers. Then, for each layer we compute a refined version with higher resolution, and the corresponding *Navigation Mesh* is computed. Finally, those individual *NavMeshes* are joined into a single one that represents the entire walkable space of the scene. Figure 21 depicts the pipeline of this process.

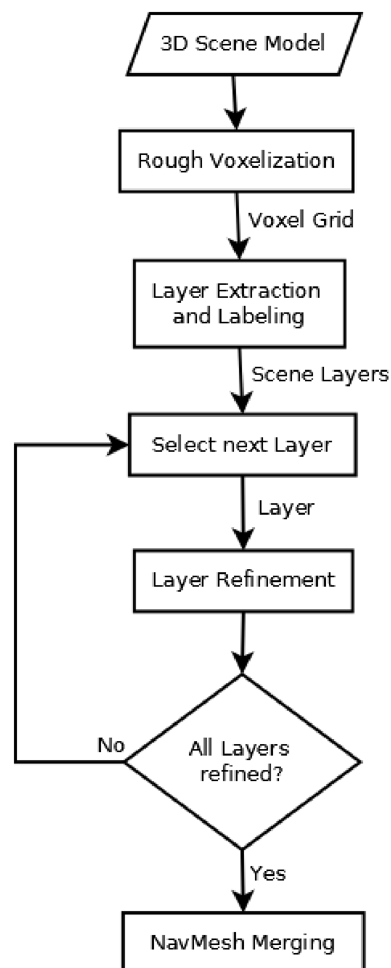

Figure 21: *The data flow of the pipeline of our approach to generate a NavMesh for MultiLayered environments.*

6.2.1 Rough Voxelization

The aim of this step is to obtain a first approximation of the walkable area. The voxelization method used is based on a GPU method described in [7]. The main idea of this algorithm is to obtain a voxelization based on a slicing method of a scene with one rendering pass. The algorithm takes as input a 3D scene. A grid is defined by placing a camera in the scene and adjusting its view frustum to enclose the area to be voxelized. This camera has an associated viewport with (w, h) dimensions. Then, the scene is rendered, constructing the voxelization in the framebuffer. A pixel (x, y) represents a column in the grid and each cell within this column is encoded using the RGBA value of the pixel. Hence, the corresponding image represents a w x h x 32 grid with one bit of information per cell. This bit indicates whether a primitive passes through a cell or not. The union for all columns of voxels corresponding to a given bit defines a slice. Consequently, the image/texture encoding the grid is called a slicemap. When a primitive is rasterized, a set of fragments is obtained. A fragment shader is used in order to determine the position of the fragment in the column, according to its depth. Notice that the column is implicitly defined by the position of the fragment. The result is then OR-ed with the current value of the frame buffer.

We apply the same idea, but three rendering passes are done in order to avoid rasterization problems, as faces near-parallel to the viewing direction do not produce any fragment. Instead, we create 3 separated slicemaps, one for each viewing direction (along the X, Y and Z axis respectively). A final slicemap is then created by merging those separated slicemaps into a single one. In addition, the fragment shader that indicates the position of the fragment on the column, also classifies this fragment, i.e., a voxel, into *positive*, *negative* and *empty* voxels. A *positive voxel* is a voxel that encloses geometry with a normal low enough to be traversed by a character. Otherwise, it is considered a *negative voxel*. An *empty voxel* is defined as a voxel that does not enclose geometry of any type. If a voxel can be *positive* or *negative* at the same time (for example, on the intersection of the floor with an obstacle), it is considered *positive*. The subsequent Layer Refinement step will determine if the geometry enclosed by the voxel is entirely walkable or not. This classification of the voxels is done in the same rendering step, using the Multi Render Target technique (MRT), one texture storing the positive voxels and the other texture storing the negative ones. Figure 22 shows the decomposition into positive and negative voxels of a simple scene.

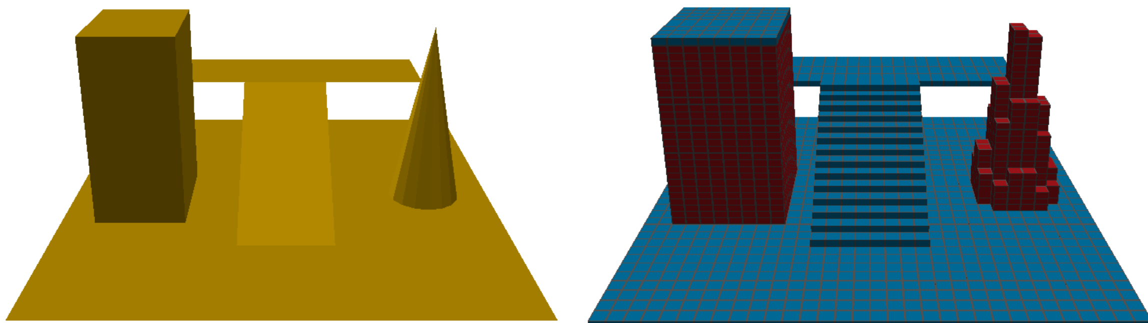

Figure 22: *A simple multi-layered scene (Left) and its voxelization (Right). Voxels are classified into positive (blue voxels) or negative (red voxels) depending on the characteristics of the geometry enclosed.*

6.2.2 Layer Extraction and Labeling

The set of all *positive* voxels obtained during the voxelization process, is known as the *potentially walkable area*, and it is a first approximation of the *real walkable area,* that is the set of voxels that are really accessible by the character. The aim of this step is to subdivide this area into layers. A layer is a set of connected *accessible voxels* that do not overlap. An *accessible voxel* is a *positive voxel* such that has enough empty voxels above it in order to contain an agent on top, without colliding with an obstruction above him. We will explain the two methods that we approached to solve this step:

6.2.2.1 First approach: Flooding & Depth Peeling

The first method that we developed to extract the layers of the scene was implemented using the CPU and the GPU. Firstly, the voxels that conforms the *real walkable area* are detected by using a flood fill algorithm, where the seed is introduced by the user, clicking on a voxel that he knows for sure that it belongs to the *real walkable area*. The *accessible voxels* connected to the seed are the voxels that forms the *real walkable area.*

Once the *real walkable area* has been determined, we proceed to subdivide this area into layers. The idea is to successively render the voxels that forms the current *real walkable area* to select the voxels that conform each layer. On each render, we let pass only the farthest fragments, and we store the depth map of the scene on a texture. The resolution used in the render is the same that the used to construct the voxelization of the scene, so the (x, y) coordinates of each pixel, determines de column of the voxel, and the depth stored on the texture determines the cell on such column. The voxels represented in the texture belong to a layer and hence, those voxels are discarded from the *real walkable area* before doing the next render. The process continues until the *real walkable area* is empty.

The main problem of this method is that it is very slow: It requires a flooding CPU step to determine the *real walkable area* and then *n* GPU steps in order to determine the layers. In addition, it does not guarantees that the layers are formed by connected voxels, and it can be problematic for the refinement layer process.

6.2.2.2 Second approach: Ordered Flooding

We realized that we could do the determination of the *real walkable area* and Layer extraction in a single flooding pass. Firstly, the voxels that conforms the *potentially walkable area* are sorted from bottom to top by its y coordinate (if two voxels have the same y coordinate, then are sorted by x, and if it is also the same, then they are sorted by z). The idea consists on assigning and propagating IDs for the different layers. So, for each voxel of this list, we check if it is an *accessible voxel*, and hence, it can transmit its layer identifier to the neighboring *accessible voxels*. If the voxel does not have yet a layer identifier, means that it is the first voxel of a new layer and therefore, a new identifier is created and assigned to it. When the layer identifier is transmitted to the neighbors, two main cases may occur: If the neighbor does not have a valid layer identifier, then the current voxel can transmit its layer identifier if there is not any voxel on the column of the neighbor that contains the same layer identifier that the current voxel is trying to transmit (this is done to avoid overlapping regions inside the same layer). On the other hand, if the neighbor had a valid different layer identifier, it means that the neighbor belongs to a different layer than the current voxel. A merging process is then applied in order to determine if

both layers can be merged into a single bigger one. It occurs if both layers do not overlap each other.

At this moment, all the *accessible voxels* have been grouped into layers, but some of those layers may not be accessible for the character. To discard those layers, the user is required to select a voxel that he knows belongs to the *real walkable area*. This voxels belongs to a layer and hence, we recursively determine the layers that are connected with this one. The union of all the voxels of those layers conforms the *real walkable area*, and the rest of layers are discarded. Figure 23 shows the resulting layers of the sample scene. Notice that as only *accessible voxels* can transmit its layer identifier, the region under the stairs where a character cannot access, as well as the part of the floor intersecting the obstacles, have been discarded.

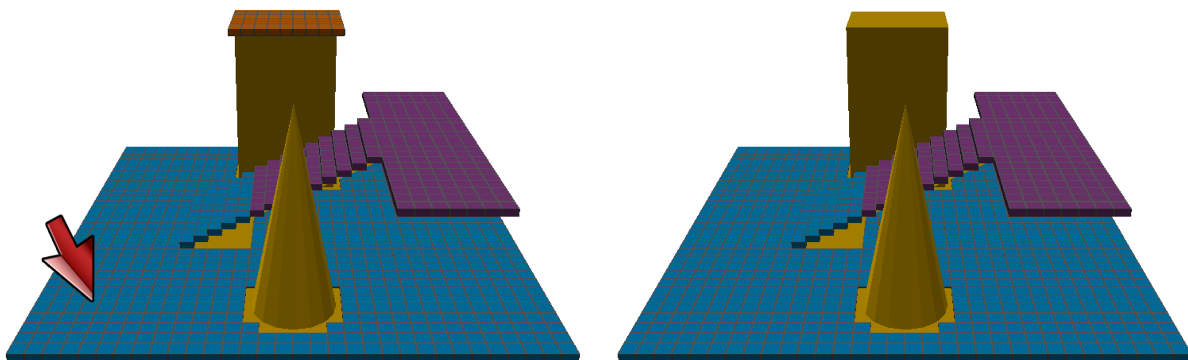

Figure 23: *The resulting layers with the ordered flooding (Left). Each color represents a different layer. The cursor indicates the position that the user has marked as walkable. The recursive process discards the regions that are not connected with the one selected by the user (Right).*

6.2.3 Layer Refinement

At this time, we have subdivided the *real walkable space* into Layers. The *Navigation Mesh* could be computed from this representation, but as we want to obtain a fine adjustment to the obstacles, we will further increase the resolution. To refine the layer, we apply an improved version of the method described in chapter 5 for the computation of the *NavMesh* for single-layered environments. The main idea remains the same, but we have simplified some stages and we have solved the problems that we have encountered during the development of this phase. Figure 24 describes the main steps of the refinement process.

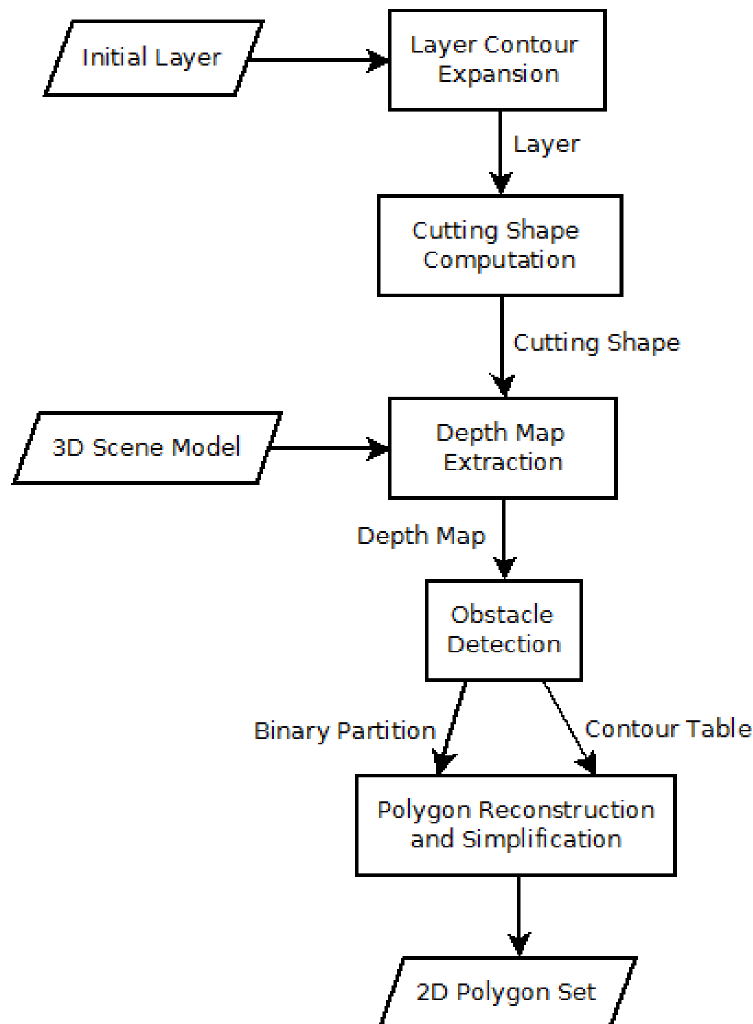

Figure 24: *Diagram illustrating the process used to refine a layer.*

Firstly we expand the *accessible voxels* of the layer that are in contact to an *obstacle voxel*, i.e., a *negative voxel* or a *positive voxel* that is not accessible. This step is necessary because some of those voxels can contain small parts of walkable geometry. During the contour expansion step, we also detect the *portal layer voxels*, that are voxels that have at least one accessible neighboring voxel that belongs to a different layer.

Once the contour has been expanded, we proceed to compute the *cutting shape* of the layer. It is a shape that adapts to the characteristics of the layer and describes the geometry that we will let pass on the hig-resolution render. The *cutting shape* is computed as follows: for every

voxel of the layer (including the already expanded voxels), we look for the voxel above it, that is at a distance equal to the height of the character in voxels. The plane defined by the top cap of this voxel, describes the cutting plane on that column (figure 25). The *cutting shape* is stored in a texture with the same width and height than the voxel grid, each pixel representing a cutting plane. The *red channel* of the pixel is used in order to identify the type of the voxel that has generated such plane. If an *accessible voxel* of the layer exists on the column represented by the pixel, the *red channel* is set to 1.0; if this voxel is a *portal layer voxel*, the red channel contains the value 0.5; finally, if none *accessible voxel* lies in the column described by the pixel, its *red channel* is set to 0.0. The *green, blue* and *alpha* channels are used in order to store the depth of the cutting plane.

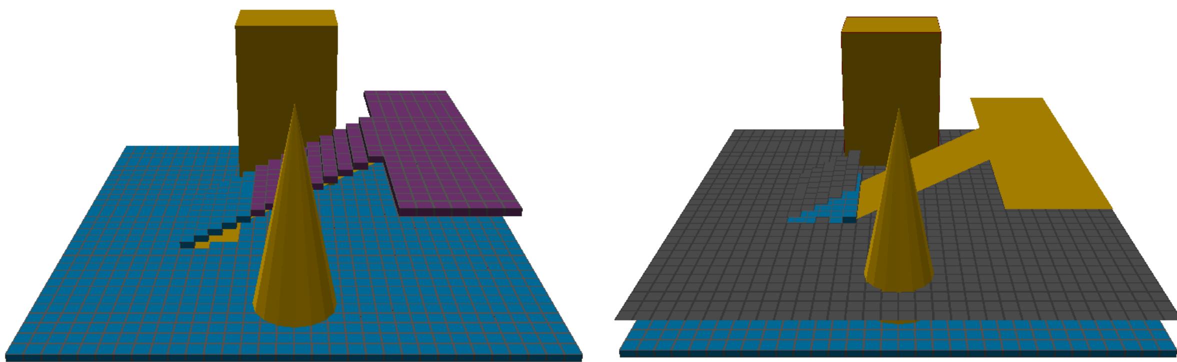

Figure 25: *The first layer of the sample scene after the expansion process (Left) and its cutting shape (Right).*

The next stage of the procedure performs a high resolution render of the scene in top view, using an orthographic camera in order to obtain its *Depth Map*. A texture is created using an improved version of the fragment shader presented in section 5.1. It takes as an input the texture representing the *cutting shape* of the layer. The screen coordinates of the fragment are used in order to do a lookup texture. If the current depth value of the fragment is greater than the one stored in the texture (GBA channels), the fragment passes the filter and can update the frame and depth buffers. Otherwise, the fragment is discarded. By doing this we guarantee that only the geometry corresponding to the current layer is rendered. The *red channel* of the fragment stores the type of the voxel that has generated such fragment. The *green, blue* and *alpha* channels are used to store the depth of the fragment. In addition, in this version of the fragment shader, the fragments containing a normal too high to be traversed by the character are discarded on this stage. This allows us to represent the depth of the fragment with more precision (24 bits instead of the 8 bits of the previous version of the shader), and consequently, it reduces the cost of the subsequent flooding stage that was done in CPU. Therefore, the result of this stage is a refined *Depth Map* of the layer, instead of the old *Normal-and-Depth Map*.

As we do a single render on top-view, if the cutting shape cuts a perpendicular face in respect to the view direction (like the cube-like obstacle in the sample scene), we may miss part of the obstacle (figure 26) or even we can miss completely the obstacle if it is small enough. This is because near-perpendicular faces do not produce any fragment on the rasterization process. This problem is solved in a pre-process step where we iterate over all the faces of the model and we keep in a list all the faces that have a normal near-perpendicular to the vector (0, 1, 0). Then, we compute which of those faces are intersected by the current layer, i.e., for every *accessible*

voxel of the layer, we check if it intersects with any face in the list. We do an additional render in top-view in wire-frame mode of all the faces that presents an intersection with an *accessible voxel*. The wire-frame mode allows us to see the frontiers of the perpendicular faces. The final *Depth Map* is the result of the fusion of both renders. Figure 26 illustrates the creation of the *Depth Map* for the first layer of the sample scene.

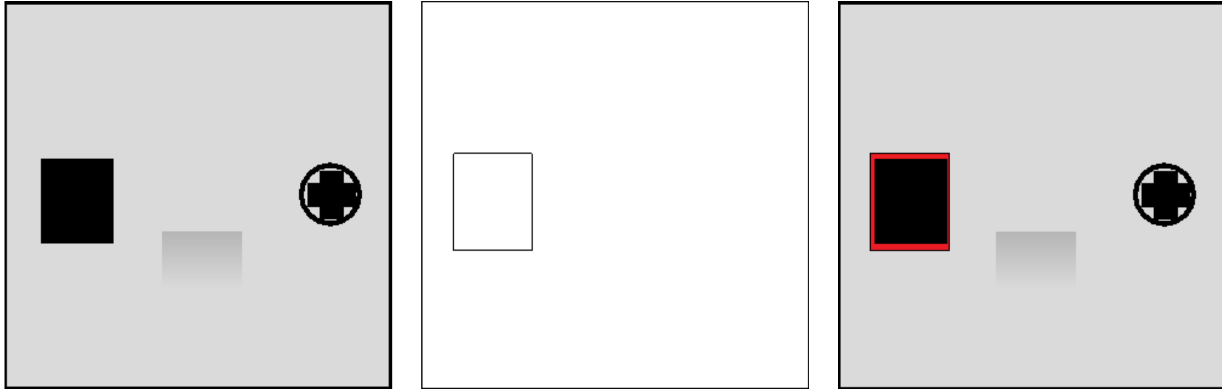

Figure 26: *The original depth map of the first layer of the sample scene, filtered with its corresponding cutting-shape (Left); the depth map of the perpendicular faces of the geometry that are intersected by the voxels of the layer (Center); the final Depth Map is the fusion of both maps (Right). In red, we have marked the zone that is inside the cube and we would detect as walkable if we do not do the fusion of both renders.*

We proceed by identifying the walkable space (floor) vs. non-walkable space (obstacles) by using a flood fill algorithm over the *Depth Map* computed in the previous stage. Notice that in this case, only the depth value of the neighboring pixels is taken into account in order to determine if two pixels are similar. However, the most important difference with the previous version of the system is that the definition of *contour pixel* has been completely reversed. A *contour pixel* is now defined as a *walkable pixel* that has at least one neighboring pixel that is *obstacle* (instead of an *obstacle pixel* that has at least one neighboring *walkable pixel*). This new definition of *contour pixel* allows us to directly suppress the subsequent stage of *Contour Expansion and Refinement*, as the new definition guarantees that we will not have degeneracies such as obstacles mapped to a single pixel or line obstacles, and hence, the resulting *Binary Partition* and *Contour Table* of this stage are the final ones.

The last step consists in reconstructing the polygons from the list of *contour pixels*. While increasing the complexity of the scenarios being treated, we realized that our previously proposed method does not always reconstruct the polygon correctly and the polygon reconstruction method from the list of contour pixels has also been modified, in order to avoid such problems. Firstly, the *contour pixels* are sorted from left to right and from top to bottom. Then, the first not-visited contour pixel is chosen. The order of the *contour pixels* guarantees that this pixel is the top-left one of the polygon that we want to reconstruct. Then, the neighboring *contour pixels* are ordered by its turn to the left (if we are reconstructing the floor) or to the right (if we are reconstructing an obstacle), with respect to the vector formed by joining the current pixel with the last visited pixel. Figure 27 illustrates this explanation. The *contour pixel* neighbor with the lesser turn to the left/right is chosen as the next contour pixel to process, and the current pixel is now set as the previous one.

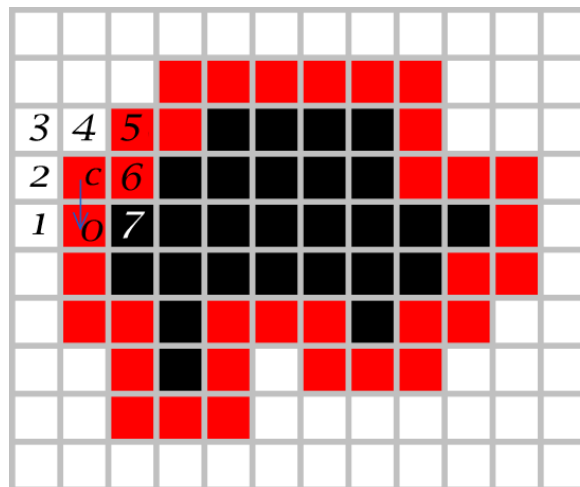

Figure 27: *The ordering of the pixels neighboring to pixel C. As we are reconstructing an obstacle, pixels are sorted to the right with respect to the vector CO.*

The algorithm of reconstructing the polygon ends when we reach again the initial pixel that we started with. As every pixel is considered a vertex, the last step consists into simplifying the generated polygon by applying the *Ramer-Douglas-Peucker* algorithm [26][6]. Finally, the *NavMesh* of the resulting set of polygons is computed by either using the CPU or the GPU versions of our method. Figure 28 shows the *NavMesh* obtained of the layers conforming the sample scene.

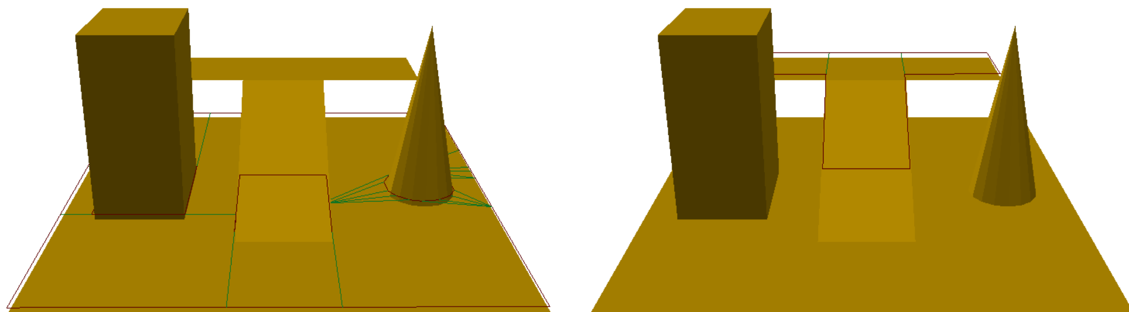

Figure 28: *The NavMesh of the first layer of the sample scene. Notice that in this case, adjusting the parameter convexDistance, we could eliminate many of the cells generated by the conic obstacle (Left); the NavMesh of the second layer of the sample scene (Right).*

6.2.4 NavMesh Merging

Once the *NavMesh* for each layer has been computed, the last stage consists into merging all those *NavMeshes* into a single one, that represents the walkable space of the entire scene. To create the final *NavMesh*, we merge each *NavMesh* of a layer with the *NavMeshes* of the adjacent layers. Let be *navMesh1* and *navMesh2* two *NavMeshes* of a pair of neighboring layers. We iterate over all the *portal layer vertices* of *navMesh1*, and we look for the closest *portal layer edge* in *navMesh2*. A *portal layer edge* is an edge of the geometry that has at least, one endpoint marked as *portal layer vertex*. Then, the Euclidean distance between the vertex and the endpoints of the edges is computed. If the vertex is close enough to one of the endpoints of the edge, the vertex is merged with such endpoint. Otherwise, the projection of the vertex over the edge is computed. If its projection is close enough to the vertex, the edge is subdivided into two new edges by the projection point, and the vertex is merged with its projection point. During the fusion of vertices, it occurs that some *geometry edges* become *portal edges*. It happens when a *geometry edge* is shared by a cell of *NavMesh1* and a cell of *navMesh2*. Some of the newly introduced portals may be *non-essential.* A portal is *non-essential* if for both endpoints of the portal, we can remove the portal and the endpoint is still convex. For example, the portal introduced during the fusion of both *NavMeshes* in the sample scene is a *non-essential* portal, and hence, it can be removed from the final partition (figure 29).

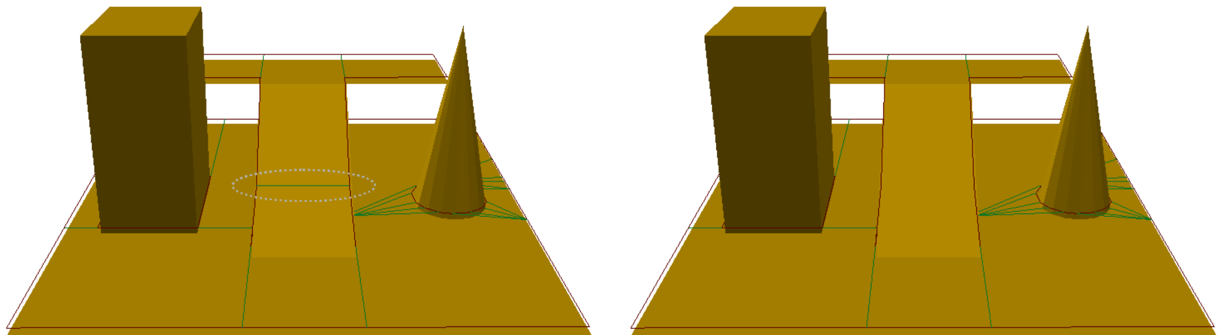

Figure 29: *The initial Scene NavMesh resulting from the fusion of the NavMeshes of the two layers (Left); the newly created portal is non-essential, and hence, it can be eliminated on the final Scene Navmesh (Right).*

The merging process of two *NavMeshes* may also introduce *T-Joints*, that are vertices such that all the incident edges are portals. This is something that needs to be avoided, since it can introduce errors in the movement of the character. We can eliminate those portals by directly removing the vertices that form those *T-Joints.* This step may introduce new local concavities, and thus the cells affected are locally partitioned again into convex regions.

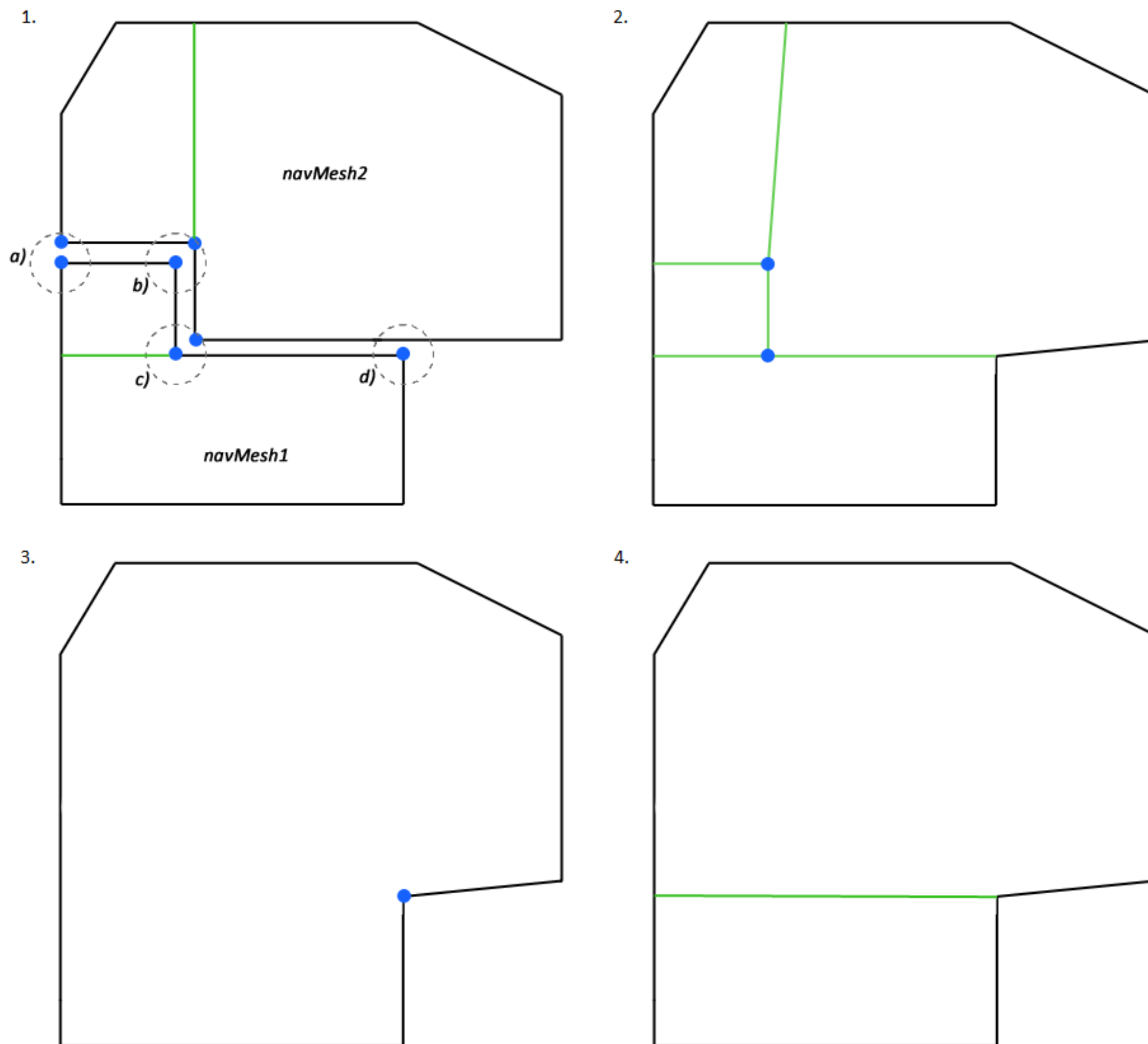

Figure 30: *Different stages of the merging process of two NavMeshes. For each portal layer vertex of navMesh1, it looks for the closest portal layer edge in navMesh2 and fusions the vertex with the best candidate over the edge, if it exists (1.). The fusion of vertices may introduce T-Joints (2.) that are eliminated as they can introduce errors on the movement of the character (3.). The elimination of T-Joints, may introduce local concavities that are solved using our method (4.).*

7 Implementation

The project has been implemented using OGRE3D, an open-source rendering engine written in C++, used for the professional development of graphic applications. The main advantage of OGRE3D is that it offers an Object-Oriented interface that abstracts the complexity of the subjacent graphics API (Direct3D or OpenGL) and hence, the programmer can focus on the development of the application, instead of on the rendering process. However, for this version of the system, we needed to add a modification on the OGRE core. In particular, for the voxelization on the GPU, it is required to blend the information that is currently on the frame buffer with the result of the shader using a logical OR, but OGRE does not natively offer this functionality, so we had to add it. The IDE used for the development of the project has been the Microsoft Visual Studio 2010.

7.1 Design

Figure 31 shows the approximated design of our system. For clarity, the name of the attributes of each class, as well as its operations has been omitted. The class *MainApp* is the starting point of the application, and it basically creates an instance of *ANavMG,* that is the main controller of all the process, and it controls the data-flow from one specific controller to another. It has an *InputManager*, that takes the input of the user and executes the corresponding action. The *SceneModel* represents the model of the scene, for which we want to compute its *NavMesh*. The *Voxelizer* is the class that voxelizes the scene model. For the development of the application, a CPU version has been implemented (*VoxelizerCPU*), as well as two GPU versions, one that produces an exact voxelization (*VoxelizerGPUExact*) and the other that produces an approximate result (*VoxelizationGPUApproximate*). The *Voxelizer* is composed by a *VoxelGrid* and it subdivides the scene into a set of *Layers.* The *VoxelGrid* is the set of all voxels of the scene. A *Layer* is defined as a set of connected *accessible voxels* that do not overlap. It also has an associated *NavMesh*, that represents its decomposition into cells and portals. The *ObstacleSolver* generates a high-resolution representation of a layer. The result consists of a set of polygons (*Poly* class), one representing the floor and the others representing the static obstacles. A *Poly* is just a set of points in the space (*Vert*). The *NavMeshGenerator* receives as an input such set of polygons, and generates its corresponding *NavMesh*. As *NavMeshGenerator* uses geometric operations intensively, such operations have been grouped into the helper class *GeometrySolver*.

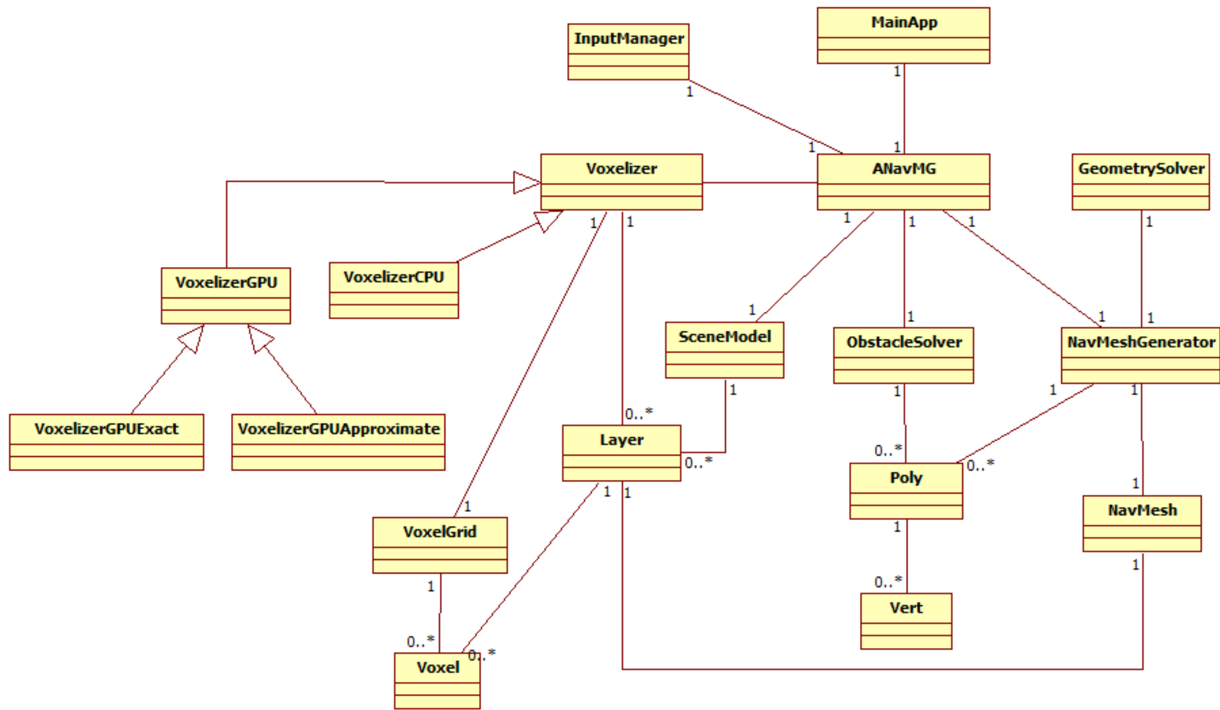

Figure 31: *The design of ANavMG*

8 Results

8.1 CPU Vs GPU Version

On this section, we compare the GPU based algorithm presented in this thesis to automatically generate *NavMeshes,* against an optimized version of the work presented in [24] with the extension of checking for visibility in the case of creating portals between a notch and a previous portal. The experimental results have been obtained on a NVIDIA GeForce 8800 GTX and an Intel Core 2Quad Q6700 at 2.66GHz with 8 GB of RAM.
To test the overall performance of the algorithm, we created 10 2D scenarios of increasing complexity ranging from 23 vertices to 965.The algorithm applied dynamic *zFar* calculation. Figure 32 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 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.

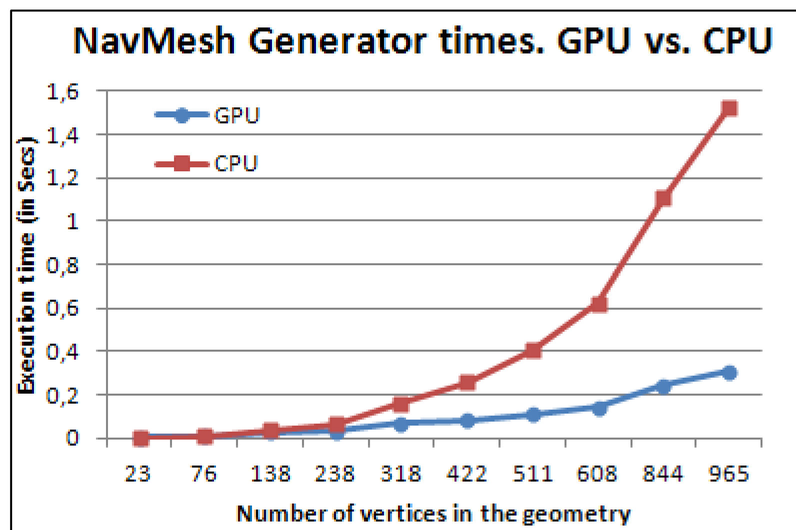

Figure 32: *Time comparison between the CPU and GPU versions of 10 scenarios with different complexity.*

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 33.

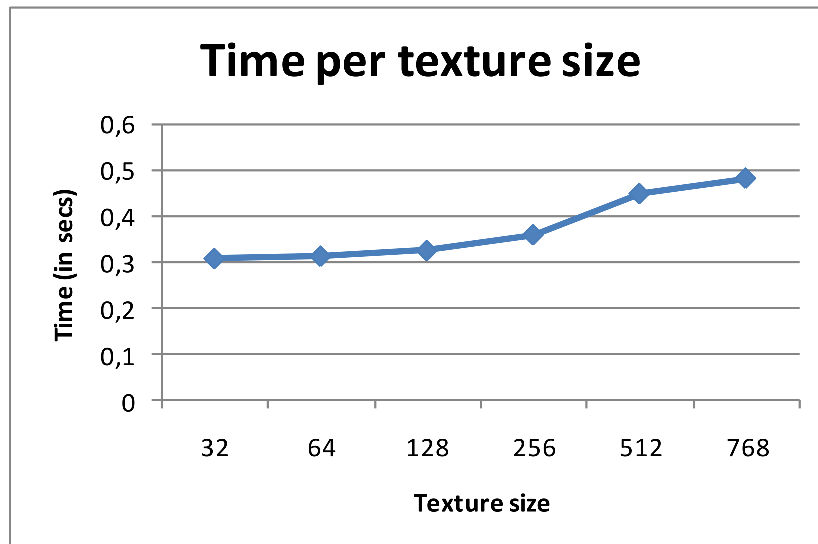

Figure 33: *Time spend to solve the most complex test environment, for different sizes of texture.*

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 34 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).

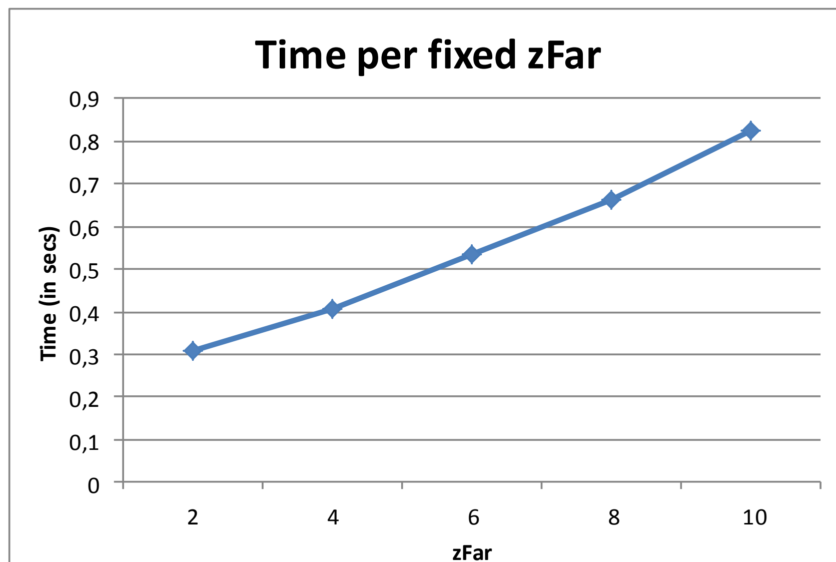

Figure 34: *Time spend to solve the most complex test environment, for different values of fixed zFar.*

Our goal with this experiment was to test whether the automatic method for calculating the *zFar* dynamically would solve the problem in similar times. Therefore we then tested that same scenario with the method presented in section 4.1 for the dynamic computation of the *zFar*. 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.

The presented GPU based method not only is more efficient than the previous CPU version, but also is more robust, since by carrying out renders, it automatically solves any visibility issues. Even though, the visibility problem when creating new portals of the type notch-portal could be treated with the CPU, we have shown an efficient and straight forward approach to solve the problem simply by performing a second render.

8.2 Multi-Layered Environments

In this section, we show the results of our application in some multi-layered environments of increasing complexity. Figures from 35 to 38 shows the test environments.

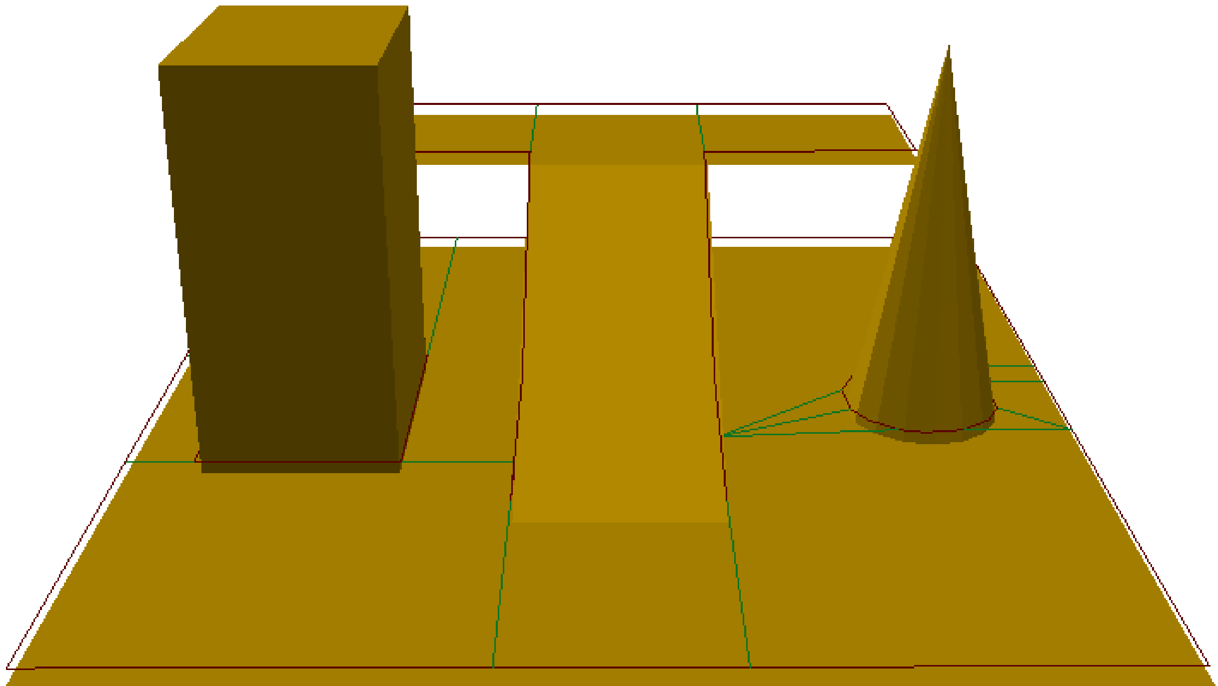

Figure 35: *The scene testMap0*

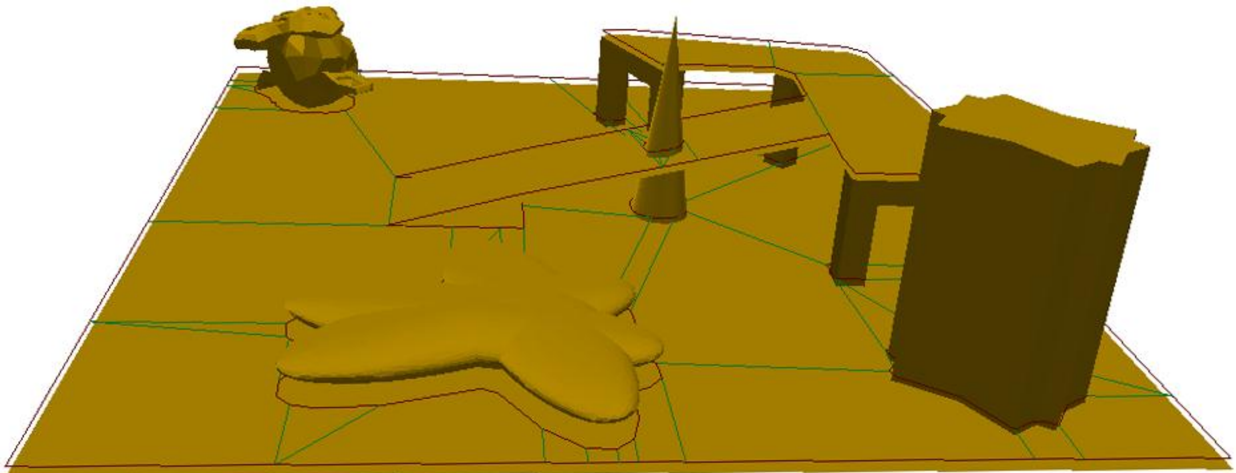

Figure 36: *The scene testMap1*

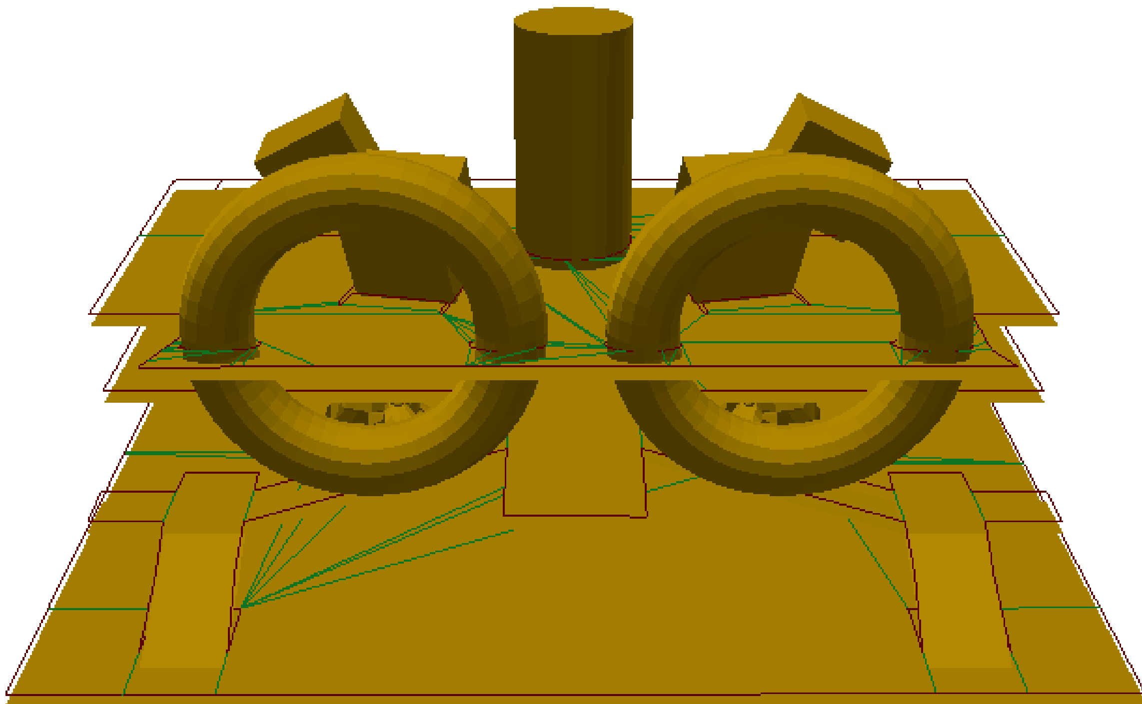

Figure 37: *The scene testMap2*

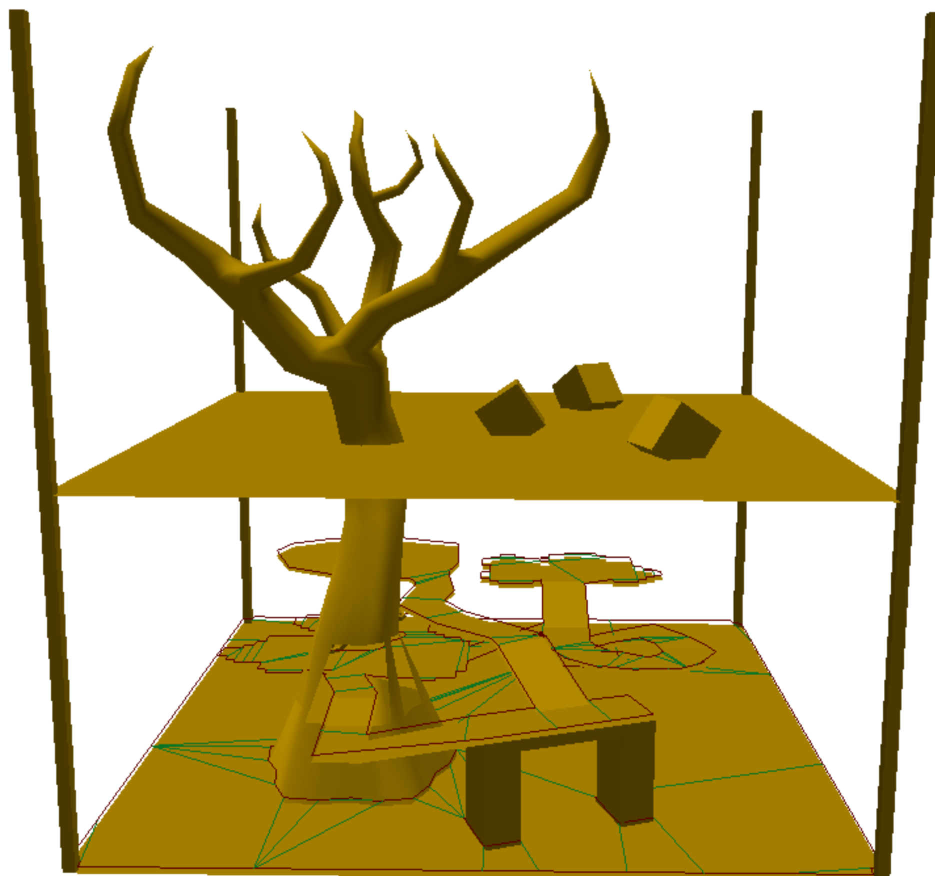

Figure 38: *The scene testMap3*

Figures from 39 to 42 show the effect in the number of resulting cells, applying different values of *convexity relaxation.*

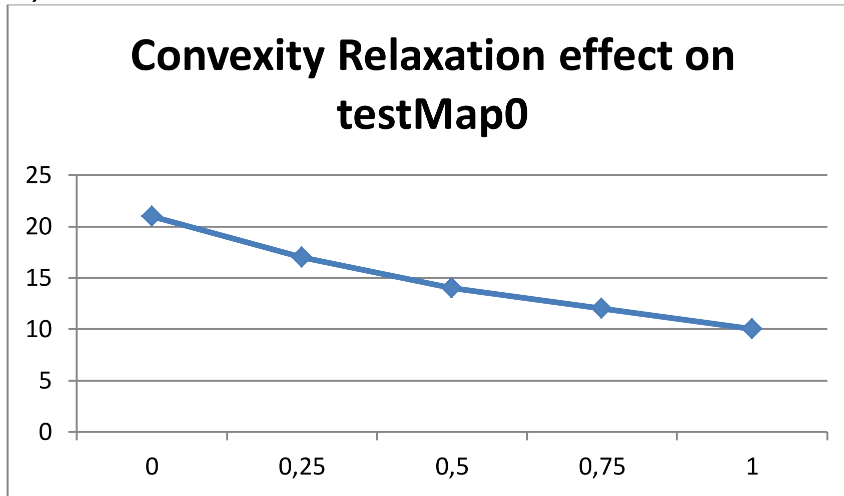

Figure 39: *The evolution of the final number of cells according to different values of the convexity relaxation, on testMap0.*

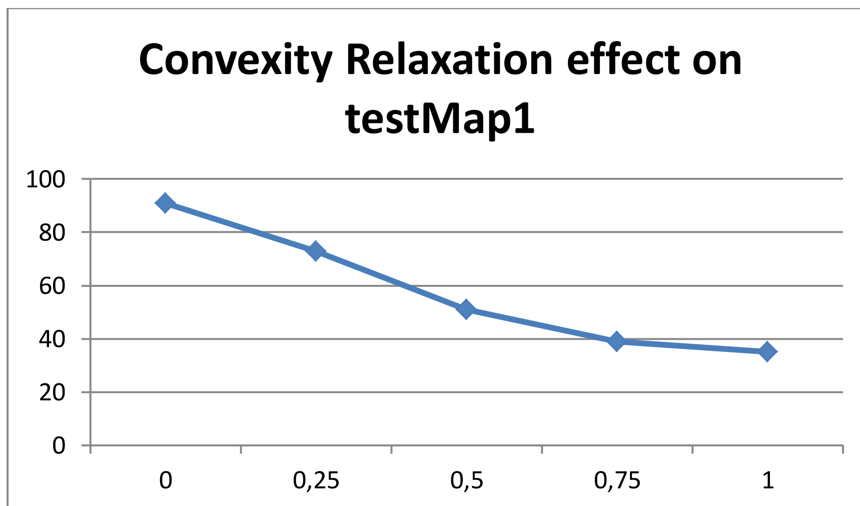

Figure 40: *The evolution of the final number of cells according to different values of the convexity relaxation, on testMap1.*

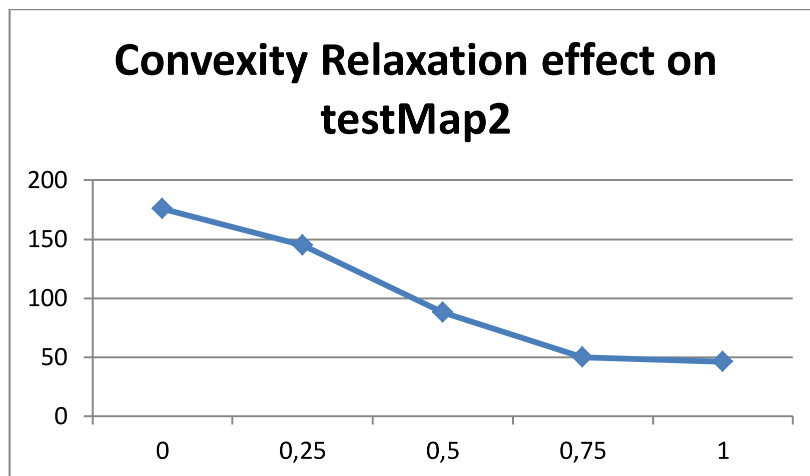

Figure 41: *The evolution of the final number of cells according to different values of the convexity relaxation, on testMap2.*

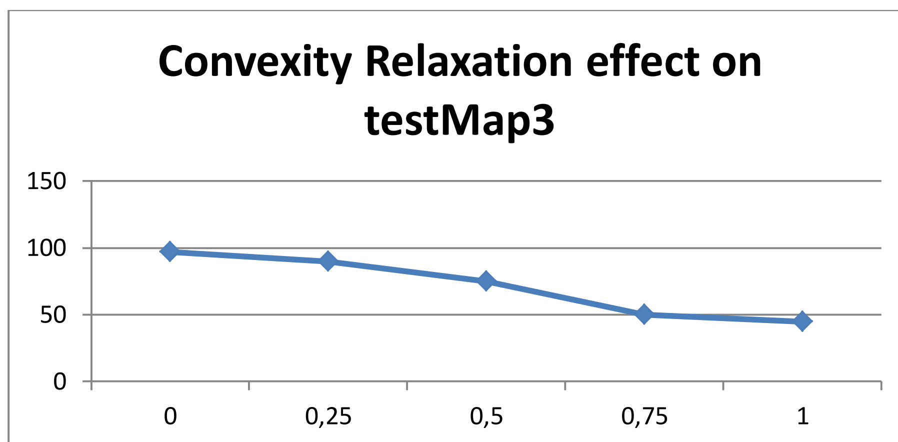

Figure 42: *The evolution of the final number of cells according to different values of the convexity relaxation, on testMap3.*

Figure 43 shows the time spend in order to solve each of the test scenarios.

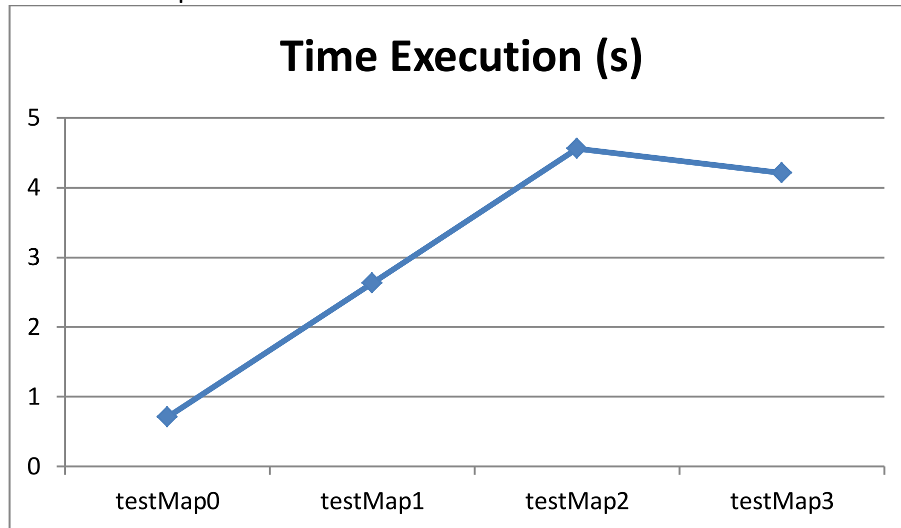

Figure 43: *Time execution for every test scenario*

Notice that *testMap2* has a time execution slightly higher than *testMap3*. This is because during the refinement process of each layer, we do a flooding step in high resolution in order to obtain the corresponding 2D representation into floor and obstacles. Hence, the cost of this step is proportional to the area of the real walkable area of the layer.

8.3 ANavMG vs Recast

Figure 44 compares the time execution of our proposed method, against *Recast* For the given example scenarios. As we can observe, our method takes considerably less time to compute the *Navigation Mesh*, and this difference seems to increase as the size and complexity of the environment increases.

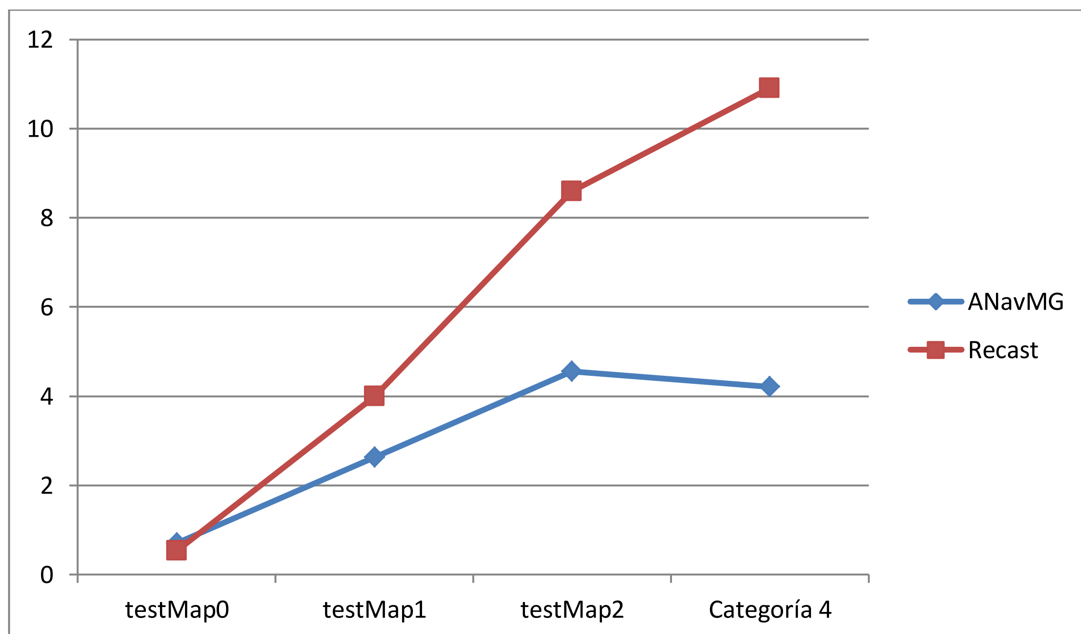

Figure 44: *Time comparison between ANavMG and Recast*

The next figure shows the main problem of *Recast* which is that it creates walkable regions where a character cannot access. This is due to *Recast* only takes into account if the geometry enclosed by a voxel is traversable by the character or not, but it does not takes into account any type of connectivity. *S*ome of those regions can be discarded if they are small enough compared to the real walkable region, but in the contrary, they cannot be discarded and therefore, the resulting *CPG* has many unnecessary cells.

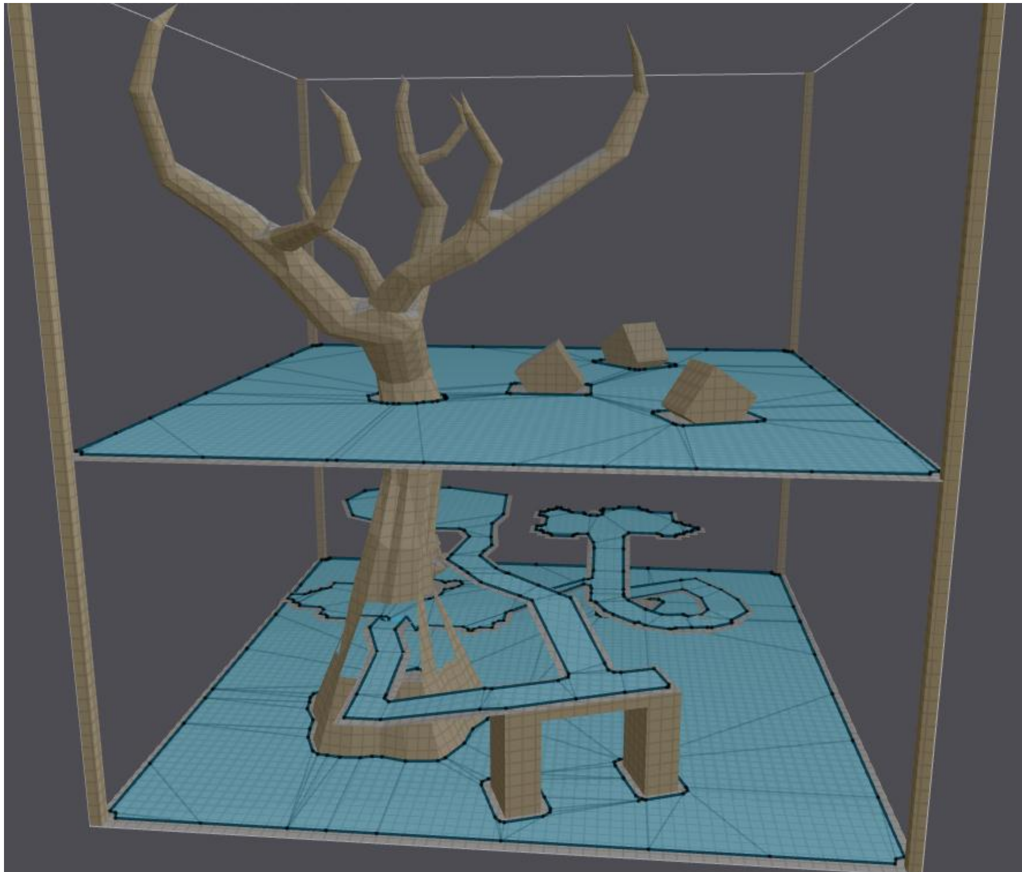

Figure 45: *Using Recast, is not possible to discard the upper-most layer, that it is not accessible by the character.*

Finally, if we compare figure 38 against 45 we can observe how our method offers a better adjustment to the contour of the obstacles, and hence, the description of the walkable space is more accurate in our case.

9 Conclusions and Future Work

In this master thesis, we have presented a novel method that uses the GPU intensively, in order to automatically compute a *NavMesh* for a given 3D scene. This work has had three main stages. Firstly, we developed a GPU based version of the algorithm that we presented in [24]. During the first phase of this thesis we have also improved the concept of *Convexity Relaxation* that we presented in our previous work. The results show that it is a powerful tool in order to reduce the final number of nodes on the *CPG* and hence, the performance of searching a path between two given nodes is much faster. On the second phase of this thesis, we focused on the development of a GPU based system that, given a 3D complex scene representing a single floor, is able to extract the 2D polygonal information that is needed by the *NavMesh* generation step. Finally, we have extended the previous system to multi-layered environments.
The results show that the GPU based version is more efficient and scalable than the CPU version method, but it is also more robust than the CPU version since it solves efficiently visibility issues that could lead to intersecting geometry.

About the *convexity relaxation*, the results show that it 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.

Compared to Recast, the results show that our system is faster, using the same resolution in both cases. In addition, *ANavMG* offers a better adjustment to the contour of the obstacles, and hence, our representation of the walkable space of the scene is more precise. Another problem that Recast suffers from is that it creates regions where the character cannot access, increasing unnecessarily the size of the final *CPG.* Moreover, the user must delete the regions that are not accessible, by manually tuning the parameters of the application. The criterion used for inaccessible region deletion is simply the size of the region, and it works only if the inaccessible regions are small enough compared to the real walkable area. We think that our criterion is more robust, as we first check which layers are connected to the region that the user has indicated as walkable, and we discard the rest. Hence, we do not create unnecessary regions and our *CPG* contains only the partition into near-convex cells of the walkable and accessible space.

As future work, we would like to add support for dynamic events. Our current method only takes into account the static geometry, which is enough for most applications, as the collisions against dynamic obstacles such as other characters, is normally solved by applying a local movement algorithm. However, it is common in applications such as videogames to have 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 on the fly. We would like to further improve our application to also handle such dynamic events in real time and modify the *NavMesh* in consequence.

Finally, we would like also to add support for more *character skills.* A part from walking, a character can do more sophisticated actions such as jumping, crouching, climbing walls, etc. Such abilities allow the character to gain access to parts of the scene that he cannot reach by simply walking, and this information should be represented on the *NavMesh*.

9.1 Personal Conclusions

Personally, although it has been a hard task, I have really enjoyed doing this master thesis, as I have had the chance to improve the work that I started during the development of my *Final Degree Project.*

During the development of this master thesis, we have written a paper for the *CEIG 2012* conference, that has been accepted for publication. It is focused on the GPU version of the algorithm to generate the *Navigation Mesh*, and the GPU system to obtain the 2D polygon representation for a 3D scene, representing a single floor-plan. We are preparing a new article with the final version of our system.

10 Bibliography

[1] ARIKAN O., CHENNEY S., FORSYTH D. A.: Efficient multi-agent path planning. In *Computer Animation and Simulation '01* (2001).

[2] BANDI S., THALMANN D.: Space discretization for efficient human navigation. *Computer Graphics Forum 17,* 3 (1998), 195-206.

[3] CHAZELLE B.: A theorem on polygon cutting with applications. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* (1982), IEEE Computer Society, pp. 339-349.

[4] CUDA. A parallel computing architecture developed by NVIDIA for graphics processing. http://nvidia.com/object/cuda_home_new.html

[5] DEMYEN D, BURO, M.: Efficient triangulation-based pathfinding. In *Proceedings of the 21st national conference on Artificial Intelligence*, The AAAI Press (2006), pp. 942-947.

[6] Douglas, D. H., Peucker, 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*. vol. 10, pp. 112-122, Dec, 1973.

[7] EISEMANN E, DECORET, X.: Single-Pass GPU Solid Voxelization for Real-Time Applications. In *Proceedings of the 2006 Symposyum on Interactive 3D Graphics and Games*.

[8] Explicit Corridor Map generator http://www.staff.science.uu.nl/~gerae101/motion_planning/cm/index.html

[9] HALE D. H., YOUNGBLOOD G, M.: Full 3D Decomposition for the Generation of Navigation Meshes. In *Proceedings of the 5th AAAI Artificial Intelligence and Interactive Digital Entertainment Conference* (Standford, California, USA, 2009), the AAAI Press, pp. 142-147.

[10] HALE D. H., YOUNGBLOOD G, M., DIXIT P, N.: Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds. In *Proceedings of the 4th AAAI Artificial Intelligence and Interactive Digital Entertainment Conference* (Standford, California, USA, 2008), the AAAI Press, pp. 173-178.

[11] HAUMONT D., DEBEIR O., SILLION B.: Volumetric cell-and-portal generation. *Computer Graphics Forum 3*, 22 (2003), 303-312

[12] HERSHBERGER J., SNOEYINK J.: Computing minimum length paths of a given homotopy class. *Computational Geometry Theory and Application 4,* 2 (1994), 63-97.

[13] HERTEL, S., MEHLHORN, K.: Fast triangulation of simple polygons. In *Proceedings of the 4th International conference on Foundations of Computation Theory.* Lecture LNCS, vol 158. Springer, New York, pp. 207-218 (1983).

[14] HOFF III K. E., KEYSER J., LIN M., MANOCHA D., CULVER T.: Fast computation of generalized voronoi diagrams using graphics hardware. *Computer Graphics 33* (1999), 277-286.

[15] KALLMANN M.: Path Planning in Triangulations. In *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games*, pp. 49-54.

[16] KALLMANN M.: Shortest Paths with Arbitrary Clearance from Navigation Meshes. In *Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation* (2010), pp. 159-168.

[17] KALLMANN M.: Navigation Queries from Triangular Meshes. In *Proceedings of the 3th International Conference on Motion in Games (MIG 2010)*, Springer, pp. 230-241.

[18] KALLMANN M., BIERI H., THALMANN D.: Fully Dynamic Constrained Delaunay Triangulations. *Geometric Modeling for Scientific Visualization*, Springer-Verlag (2003), pp. 241-257.

[19] KUFFNER J. J.: Goal-directed navigation for animated characters using real-time path planning and control. *Lecture Notes in Computer Science 1537* (1998), 171-179.

[20] LAMARCHE, F.: TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints. *Computer Graphics Forum 28,* 2 (2009), 649-658.

[21] LEE D. T., PREPARATA F. P.: Euclidean shortest paths in the presence of rectilinear barriers. *Networks 3,* 14 (1984), 393-410.

[22] LERNER A., CHRYSANTHOU Y., COHEN-OR D.: Efficient Cells-and-portals Partitioning. *Computer Animation & Virtual Worlds 17,* 1 (February 2006), 21-40.

[23] OKABE A., BOOTS B., SUGIHARA K.: Spatial Tessellations: Concepts and applications of Voronoi Diagrams. *John Wiley & Sons, Inc. New York, NY, USA* 1992.

[24] OLIVA R., PELECHANO N.: Automatic Generation of Suboptimal NavMeshes. In *Proceedings of the 4th International Conference on Motion in Games (MIG 2011)*, Springer, pp. 328-339.

[25] QI M, CAO, T-T., TAN, T-S.: Computing 2D constrained Delaunay triangulation using the GPU. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, (New York, NY, USA, 2012), ACM, pp. 39-46.

[26] Ramer, U. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing.* vol. 1, pp. 244-256, Nov, 1972.

[27] Recast Navigation Toolkit

http://code.google.com/p/recastnavigation/

[28] REYNOLDS, C. W.: Steering Behaviors For Autonomous Characters, in *proceedings of Game Developers Conference (GDC'99)* pp. 763-782. 1999

[29] ROERDINK J. B. T. M., MEIJSTER A.: The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae, 41* (1-2), 187-228 (2000).

[30] ROSENFELD A., PFALTZ J. L.: Sequential operations in digital picture processing. *Journal of the ACM 13,* 4 (1966), 471-494.

[31] SNOOK G.: Simplified 3D movement and pathfinding using navigation meshes. *Game Programming Gems* (February 2000), 288-304.

[32] SUD A., ANDERSEN E., CURTIS S., LIN M., MANOCHA D.: Real-time path planning for virtual agents in dynamic environments. In *IEEE Virtual Reality conference* (2007).

[33] TOLL W., COOK IVA. F., GERAERTS R.: Navigation Meshes for Realistic Multi-Layered Environments. In *Proceedings of the 24th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, Springer, pp. 3526-2532.

[34] TOZOUR P.: Building a Near-Optimal Navigation Mesh. *AI Game Programming Wisdom* (March 2002), 171-185.

[35] Unreal Engine's *NavMesh* Generation Method:

http://udn.epicgames.com/Three/NavigationMeshReference.html

[36] Valve's *NavMesh* Generation Method: http://developer.valvesoftware.com/wiki/Navigation_Meshes