Universitat Politènica de Catalunya

Master Thesis

Study and Development of Hierarchical Path Finding to Speed Up Crowd Simulation

Student:
Carlos Fuentes Paredes

Advisor:
Nuria Pelechano Gómez

A thesis submitted in fulfilment of the requirements
for the degree of Master in Innovation and Research in Informatics

in the

Facultat d'Informàtica de Barcelona

November 2014

Contents

Contents i

List of Figures iii

List of Tables vi

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Objectives . . . 2
- 1.3 Contribution . . . 2
- 1.4 Organization . . . 2

2 State of the Art 4
- 2.1 Environment Division . . . 5
  - 2.1.1 Regular Grids . . . 5
  - 2.1.2 Roadmaps . . . 6
  - 2.1.3 Navigation Meshes . . . 8
    - 2.1.3.1 Triangular NavMeshes . . . 9
    - 2.1.3.2 Convex polygon decomposition . . . 9
- 2.2 Path finding Algorithms . . . 10
  - 2.2.1 Introduction . . . 11
  - 2.2.2 Dynamic Search . . . 12
- 2.3 Environment-Aware Navigation . . . 14
  - 2.3.1 Dynamic Search on the GPU . . . 17
  - 2.3.2 Planning in Complex Domains . . . 19
- 2.4 Hierarchical Path finding . . . 22
- 2.5 Conclusion . . . 25

3 Our Approach 27
- 3.1 Introduction . . . 27
  - 3.1.1 Voxelize the polygons. . . . 28
  - 3.1.2 Build navigable space from solid voxels. . . . 28
  - 3.1.3 Build watershed partitioning and filter out unwanted regions. . . . 29
  - 3.1.4 Trace and simplify region contours. . . . 29
  - 3.1.5 Triangulate the region polygons and build triangle connectivity. . . . 29
- 3.2 Hierarchical Subdivision . . . 32
- 3.3 Path finding Computation . . . 36
  - 3.3.1 Find S and G al certain level . . . 37

3.3.2 Connect S and G to the graph . . . 37
3.3.3 Search for a path between S and G at the highest level . . . 38
3.3.4 Obtain optimal subpaths . . . 38
3.3.5 Delete temporal nodes . . . 38

4 Results and Discussion **41**
4.1 Performance test . . . 41
4.1.1 Number of Nodes . . . 41
4.1.2 Time execution . . . 43
4.1.2.1 Find S and G al certain level . . . 47
4.1.2.2 Connect S and G to the graph . . . 48
4.1.2.3 Search for a path between S and G at the highest level . 49
4.1.2.4 Obtain optimal subpaths . . . 49
4.1.2.5 Delete S and G . . . 51
4.2 Discussion . . . 51

5 Conclusions and Future Work **54**
5.1 Conclusions . . . 54
5.2 Future Work . . . 55

A Performance Test **56**

Bibliography **71**

List of Figures

2.1 Regular Grid Extraction with two states: 3D environment (a). Extracted graph (b). . . . 5
2.2 Limitations of Grid Methods: Walkable and obstacle area inside the same cell (a). Restricted movement (fixed angles)(b). . . . 6
2.3 Probabilistic Roadmap Planner: Connect Start and Goal node to the roadmap. . . . 7
2.4 Voronoi-Based Roadmaps: Retract PRM (nodes and edges) to the medial axis. . . . 7
2.5 The Constrained Delaunay Triangulation: *O(n)* conformal cell decomposition. . . . 9
2.6 NEOGEN subdivision. From left to right we can see the original scene, the result of the layer extraction step after the coarse voxelization,the 2D floor plan of each layer, and finally the near optimal navigation mesh. . . . 10
2.7 Representations of the environment division. Empty map a). Regular Grid map b). Roadmap c). NavMesh d) . . . 11
2.8 Minimal memory abstract graph. . . . 13
2.9 Movement behaviour: Set of possible actions. . . . 14
2.10 Hybrid discretization: Blue edges represent the grid portion of the graph while red edges represent the highways . . . 15
2.11 Static and Dynamic Constraints) . . . 16
2.12 Density Based Path Planning . . . 17
2.13 Algorithm: Propagate state inconsistency . . . 18
2.14 Different scenarios: (a) Agents crossing a highway with fast moving vehicles in both directions. (b) 4 agents solving a deadlock situation at a 4-way intersection. (c) 20 agents distributing themselves evenly in a narrow passage, to form lanes both in directions. (d) A complex environment requiring careful foot placement to obtain a solution. . . . 19
2.15 Domain Representations: (a) Problem definition with initial configuration of agent and environment. (b) Global plan in static navigation mesh domain Σ_1 accounting for only static geometry. (c) Global plan in dynamic navigation mesh domain Σ_2 accounting for cumulative effect of dynamic objects. (d) Grid plan in Σ_3. (e) Space-time plan in Σ_4 that avoids dynamic threats and other agents. . . . 20
2.16 Characters exhibiting two cooperative behaviours: Two actors hide while a third lures a guard away (a), then sneak past once the guard is distracted (b). An actor lures the guards towards him (c), while a second presses a button to activate a trap (d). . . . 22
2.17 Upper graph level with different subdivision: 100 clusters (a). 25 clusters (b). 16 clusters (c). 4 clusters (d). (Model: map (1000 nodes) . . . 23

2.18 Abstract Graph: Black cells represent obstacles and walls. Entrances are blue cells. Green cell is the start and the red one is the goal. Path between them is with orange color. 23
2.19 HPA* in Recast tool: Cluster division (purple). Entrances (red). Inter and intra edges (blue). Optimal path (black). 24
2.20 Clearance Values: (a)-(d) Computing clearance; the square is expanded until a hard obstacle is encountered. (e)-(g) Clearance values for different capabilities 25

3.1 Recast Tool software. (Model: Tropical Islands (12666 polygons)). 28
3.2 Build navigable space from solid voxels: Voxelization with walkable cells marked (a) and walkable cells overlayed on top of input geometry (b). . . 28
3.3 Build watershed partitioning and filter out unwanted regions: The catchment basins become the centers of the regions. (White areas represent lower region). 29
3.4 Ramer-Douglas-Peucker algorithm: Initial vertices at region edges (a); Find vertex with maximum error, and subdivide (b); Iterate untill certain error criteria is met (c). 30
3.5 Contours: Traced contours (a); Simplified contours(b). 30
3.6 Triangulation 31
3.7 Recast steps (from left to right): Input mesh and the five steps of Recast process. (Model: Scifi City (2090 polygons)). 31
3.8 Recast tool: Unreachable walkable areas. (Model: Scifi City (2090 polygons)). 32
3.9 Hierarchical Subdivision: The NavMesh triangulation of the model (a). The graph of the lowest level (level 0) (b). Nodes are painted in different colors. Edges connects a node with its neighbours (Model: Dungeon (120 polygons)). 33
3.10 The three phases of multilevel k-way graph partitioning. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph 34
3.11 Hierarchical Subdivision: (Simple map, $numMergedNodes = 5, levels = 5$). Portals are presented with red dots. IntraEdges are painted with yellow lines. Partitions are exposed with black, blue and red separation lines respectively. Level 0 = 76 nodes (a), Level 1 = 12 nodes (b), Level 2 = 3 nodes (c). (Model: Simple Map (76 polygons)). 36
3.12 Hierarchical Graphs: (City Islands, $numMergedNodes = 3, levels = 10$). Level 0 = 5151 nodes (a), Level 1 = 1469 nodes (b), Level 2 = 316 nodes (c), Level 3 = 72 nodes (d), Level 4 = 17 nodes(e), Level 5 = 4 nodes (f). (Model: City Islands (5515 polygons)). 36
3.13 Connect Start node to the graph: Red circles are portals of the orange partition. White lines are the computed intraedges. Gray polygons are obstacles or no walkable areas. 37
3.14 Hierarchical Subdivision: The graphs in the hierarchy from the lowest level (From (a) to (e)). The start and goal nodes are written in white. . . 38
3.15 Path finding Computation: S and G are linked to their partitions at level 2 (a). Sub paths are calculated until level 0 is reached. (Level 1 (b) and Level 0 (c)). The final result of the shortest path between S and G at level 2 (d). 39

3.16 Path finding Computation: (Serpentine Islands, $numMergedNodes = 4, levels = 10$) Path finding at level 0 (3908 nodes) (a). Path finding at level 0 where each node has a different color (b). Path finding at level 3 (28 nodes) (c). 40

4.1 Sample maps: (Dungeon (a). City Colony (b). Serpentine City (c). City Islands (d). Tropical Islands (e). Sirus City (f). 42
4.2 Level vs Number of Nodes (Sirus City). 43
4.3 Level vs Execution Time (Map). 44
4.4 Map model (a). Shortest path ($numMergedNodes = 2, level = 2$) (b) . . 44
4.5 Level vs Execution Time (City Colony). 45
4.6 City Colony model (a). Shortest path ($numMergedNodes = 5, level = 2$) (b) 45
4.7 Level vs Execution Time (Tropical Islands). 46
4.8 Tropical Islands model (a). Shortest path ($numMergedNodes = 16, level = 2$) (b) 46
4.9 Get S and G Time vs Level 48
4.10 Link S and G Time vs Level 49
4.11 A* Time vs Level 50
4.12 Obtain optimal subpaths. The map model has two nodes at level 3. No intermediate paths are calculated 50
4.13 Time vs Level 51
4.14 Special case: Start partition has 18 portals (red points) and end partition has 10 (blue points) 52

List of Tables

4.1 Sample maps 42
4.2 Number of portals (Sirus city) 48

A.3 City Colony (2615 Nodes) 57
A.4 Serpentine City (3908 Nodes) 59
A.5 City Islands (5515 Nodes) 61
A.6 Tropical Islands (12666 Nodes) 63
A.7 Sirus City (18738 Nodes) 66
A.1 Dungeon (120 Nodes) 69
A.2 Map (208 Nodes) 70

Chapter 1

Introduction

1.1 Introduction

Path finding is a common problem in computer games. Most videogames require to simulate thousands or millions of agents who interact and navigate in a 3D world showing capabilities such as chasing, seeking or intercepting other agents. This behaviour is solved using path finding. A* is the most commonly used method to determine the shortest path between two points. It expands the nodes in the graph representation of the environment with the smallest estimated solution cost first; however the cost of this search can grow exponentially with the size of the terrain and the allocated memory is very limited. Therefore, a hierarchical subdivision of the world environment is necessary under those restraints.

Hierarchical path finding has been studied in the last decade, it allows to compute the shortest and optimal route between two locations in large terrains based on a hierarchical graph. This significantly decreases the execution time and memory footprint in crowd simulation environments. Many of these approaches only have one abstract graph to work on. It means, the searches are just done on one abstract level of the environment subdivision. These approaches have been only applied in 3D environments based on regular grids.

The work presented in this master thesis proposes a new hierarchical path finding solution for big environments. We use a navigation mesh as abstract data structure to partition the 3D world. Then, we extract the graph representation and consider it as the level 0 in a hierarchical tree. After, many subdivisions are created by recursively partitioning a lower level graph into a specific number of nodes. The number of nodes is a parameter. The partition is performed until the graph of the latest level can't be

divided. Thus, a particular path planning search can be executed in any level of this hierarchical tree. The higher the level of the hierarchy, the fewer the number of nodes to search in. This approach allows faster path finding calculations than a common A* without any hierarchy.

1.2 Objectives

The execution time of planning the shortest path from one point to another becomes slower in big scenarios. The graph that represents the environment division could be oversize due to the number of nodes and edges. That size makes calculating traditional shortest-path algorithms too slow for a real application. Therefore, the main purpose of this thesis is to speed up the shortest path time execution applying a hierarchical subdivision of the environment with many levels.

In order to achieve our goal, we first studied the state of the art not only in path finding algorithms but also in spatial division and hierarchical approaches. We focused more specifically on the work based on hierarchical methods and partitioning.

1.3 Contribution

The main contribution of this thesis is a new hierarchical path finding subdivision method to speed up crowd simulation that works on navigation meshes as opposed to previous work based on regular grids. Our approach divides the environment in a hierarchy of graphs in order to minimize the complexity of the problem. It basically has two steps: firstly, it creates the hierarchy tree based on a recursively partition (preprocessing step). This partition is made by taking into account the number of connector edges. Secondly, we insert the start and goal nodes into the current graph at certain level and search the optimal path with common A* between them. Once the high level path is calculated, we compute the low path level within the higher level start and goal nodes (online step).

1.4 Organization

The rest of this work is organized as follows. In the next chapter, we discuss about the State of the Art on path finding algorithms, spatial environment subdivisions and hierarchical approaches. In chapter 3, we do a brief description of the tool we use for

partitioning as well as the explanation of our approach. Here we will describe how the graph partition is done as well as a deep description of the preprocessing and online steps required by our method. The comparison and results are explained in the chapter 4 together with a discussion. Finally, we present conclusions and future work.

Chapter 2

State of the Art

Many path finding algorithms have been studied for more than a decade, all of which attempt to balance the inherent tradeoff between two criteria, namely the path planning runtime computation and the resulting path quality.

The computational cost depends on how the scenario is divided (based on grids, waypoints or navigation meshes) and whether the navigation environment is fully static or dynamic, where a complex replanning may be required on the fly in order to get the correct path. This cost can be significantly reduced by using GPU calculation.

At present, a hierarchy of graphs is applied to reduce the computational calculation known as hierarchical path finding, where the idea is to decompose the search problem into multiple searches on smaller graphs and to cache information about path segments that are shared by many routes. The higher the level the graph belongs to, the smaller the number of nodes the graph has.

In order to understand the current state of the art on path finding techniques, this chapter starts with a short summary about static and dynamic path planning algorithms and spatial environment subdivision techniques in computer games. These techniques allow robust and efficient computation of paths regardless of the kind of surrounding area in the interactive virtual world.

Then, we proceed with a compilation of some of the most important approaches on path finding, based on GPU, multi-domain planning and hierarchical subdivisions. Finally, the report is completed with some conclusions.

2.1 Environment Division

In videogames and artificial intelligence fields, the 3D world environment could be characterized as a weighted direct graph. There are well known structures suitable for fast path planning calculation. These spatial divisions allow fast searches for collision-free paths, but which are not necessarily globally the shortest ones. The most common environment representations are regular grids and navigation meshes (NavMesh).

2.1.1 Regular Grids

Regular grids are the simplest way to represent the 3D world. This regular cell grid is commonly used both global and local paths of agents. The scenario is divided into a two dimensional square cells with the same size. Each cell can have two states: opened or blocked. An opened cell is the space where the character can stand or move in, a blocked one is not accessible for the character because there are obstacles or walls. However, more than two states can be stored in a cell such as density, percentage of occupancy, etc. In the method proposed by Loscos [1], they store more than two states in each cell of a regular grid. Each cell has local information for collision of the environment and agents in order to improve the realism of the simulation. It also can be extended for creating more complex agent-environment behaviours using different layers [2], [3].

A navigation graph can be extracted from the regular grid by using the connectivity information between cells. Each node in the graph represents a cell and an edge the link between two cells.(See Fig. 2.1)

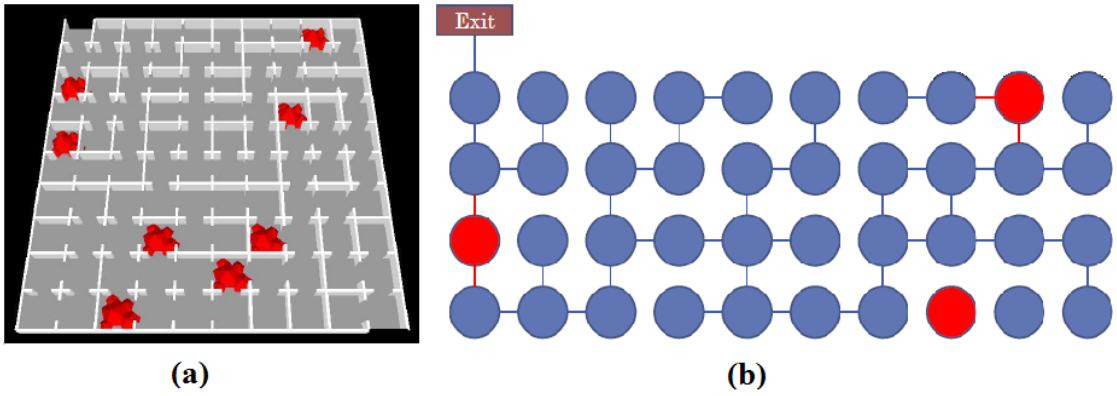


FIGURE 2.1: Regular Grid Extraction with two states: 3D environment (a). Extracted graph (b).

If the virtual world is small and grid-like, this technique is useful. Also, this approach is ideal for 2D terrains where the height of the character is not relevant. Moreover, the resolution of the grid determines how accurate it represents the walkable space. However,

the memory cost becomes unacceptable when the number of cells is extremely high for a regular grid with enough resolution. The idea of partitioning the world representation in cells with big sizes in order to mitigate this problem could affect the path quality.

Other limitation of working on grids is the coarse coverage of underlying terrain, it means that one big cell could cover both walkable and obstacle area. It should classified as either walkable or obstacle under some criterion. Also, the paths don't look realistic because of the agent movement is only restricted to do fixed angle turns. Motions created by grid search tend to be unnatural because, in general, for grid searches a smooth path needs to be created in the post-processing phase resulting in expensive queries. See Fig. 2.2

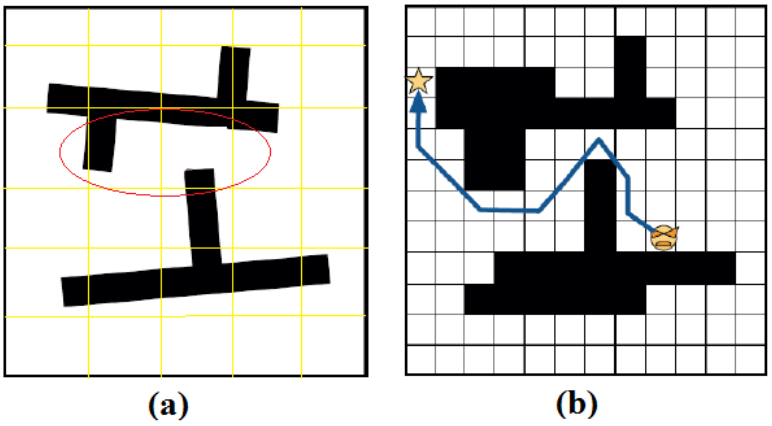


FIGURE 2.2: Limitations of Grid Methods: Walkable and obstacle area inside the same cell (a). Restricted movement (fixed angles)(b).

Therefore, using large grids that are composed for thousand of cells could easily exceed the memory requirements of the current high-end hardware provides. Moreover the preprocessing time of calculating the extracted graph becomes impractically large with the growing number of cells.

All of these limitation make a grid representation not suitable for big environments.

2.1.2 Roadmaps

The Probabilistic Roadmap Planner (PRM) [4], [5] is a planner that can compute collision-free paths. The PRM consists of two phases: a construction phase (off-line) and a query phase (on-line).In the construction phase, a roadmap is built, it consists of computing a very simplified representation of the free space by sampling configurations at random. Then the sampled configurations are tested for collision and each collision-free configuration is retained as a "milestone". Each milestone is linked by straight paths to its k-nearest neighbours. Finally the collision-free links will form the PRM.

Sampled configurations and connections are added to the roadmap until the roadmap is dense enough. A roadmap is normally represented as a graph in which the nodes correspond to placements of the entity and the edges represent collision-free paths between these placements.

In the query phase, the start and goal configurations are connected to the roadmap. Then, the path can be obtained by a Dijkstra's shortest path query. See Fig. 2.3

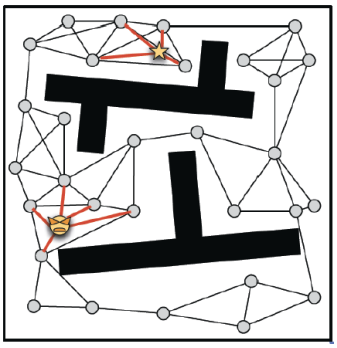


FIGURE 2.3: Probabilistic Roadmap Planner: Connect Start and Goal node to the roadmap.

A roadmap can also use a Voronoi diagram classifier [6]. The Voronoi diagram is one of the most popular structure for spatial partitioning. Given seed points, it will partition the plane in cells such that for each seed there will be a corresponding cell consisting of all points closer to that seed than to any other.

In order to calculate the seeds, a random sample is picked of the entity (placement) in each iteration. Then, the placement is checked whether is collision free from the entity. If so, it is retracted to the Voronoi diagram using binary interpolation. Finally, the edges are also retracted until every part of the edge is at least some pre-specified distance away from the obstacles. See Fig. 2.4.

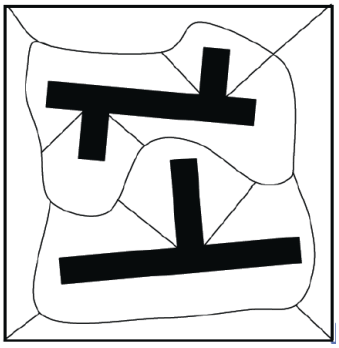


FIGURE 2.4: Voronoi-Based Roadmaps: Retract PRM (nodes and edges) to the medial axis.

Unfortunately, the PRM method leads to low quality roadmaps, consisting of straight line segments, that require a lot of time-consuming smoothing in order to be useful for virtual world applications. This is due to the random nature of the PRM method. Also, it lead to larger graphs when many milestones are needed.

2.1.3 Navigation Meshes

Navigation Meshes (NavMesh) is a data structure that is specifically designed for supporting path planning and navigation computations. It encodes a convex decomposition of the scene where each convex polygon (nodes) is a walkable area and they are connected using links (edges) that provide the connection between cells for agents to walk through. This representation has fewer nodes which contain more accurate information about the 3D environment.

The main function of a navigation mesh is to represent the free environment efficiently in order to allow path queries to be computed in optimal times and to support other spatial queries useful for navigation. NavMesh has some properties that are listed below:

- **Linear number of cells.** A navigation mesh should represent the environment with $O(n)$ number of cells or nodes (n) for efficient path calculations. This is critical for path search to run in optimal times.
- **Quality paths.** A navigation mesh should facilitate the computation of quality paths. At least, locally shortest paths should be provided.
- **Arbitrary clearance.** A navigation mesh should provide an efficient mechanism for computing paths with arbitrary clearance from obstacles. No pre-computed clearance valued should be known.
- **Representation robustness.** A navigation mesh should be robust to degeneracies in the description of the environment. Each description of obstacles should be handled such as intersections, overlaps, etc.
- **Dynamic updates.** A navigation mesh should efficiently update itself to accommodate dynamic changes in the environment.

An example of the world representations are shown the Fig. 2.7 [1].

[1]The 3D model representations were performed using the Recast navigation Tool. [7]

2.1.3.1 Triangular NavMeshes

Most of the path finding approaches use the Constrained Delaunay Triangulation (CDT) [8], [9] in a planar surface. It can be defined as follows. Given a set P of n points in the plane, the Delaunay triangulation of P, is the rectilinear dual graph of the Voronoi Diagram (See Fig. 2.5). CDT has the following characteristics:

- Two points $p_i, p_j \epsilon P$ form a Delaunay edge if and only if there exists a circle through p_i and p_j which does not contain any point of P in its interior.
- Three points p_i, p_j, p_k form a Delaunay triangle (in general, are vertices of a face) if and only if the circle through them does not contain any point of P in its interior.

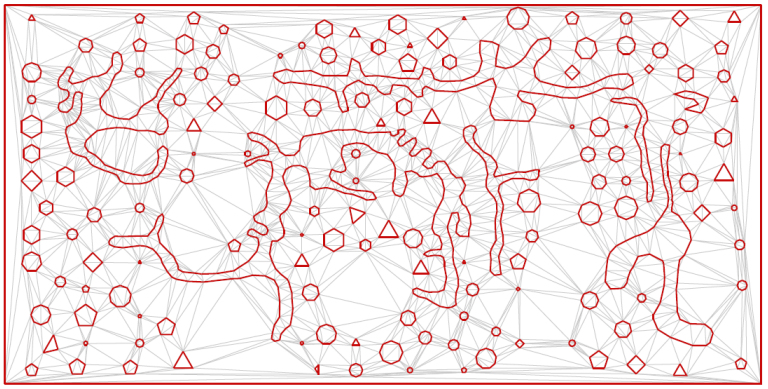


FIGURE 2.5: The Constrained Delaunay Triangulation: *O(n)* conformal cell decomposition.

CDT maximizes the minimum internal angle of its triangles and is the dual of Voronoi diagram. It is widely used to represent the free environment, for navigation and path planning.

Other approach based on NavMeshes is explained by Kallmann [9]. He proposes a Local Clearance Triangulations (LCT) which is a refinement strategy for CDTs allowing clearance information to be stored in the triangulation and well address dynamic updates and robustness. It degenerates input such as when obstacles intersect or overlap during dynamic movements. This reduce the run-time clearance determination to a simple value comparison tests during path search.

2.1.3.2 Convex polygon decomposition

Most of the spatial structures try to reduce the number of nodes in the virtual representation. NEOGEN structure [10] provides a near optimal NavMesh from a 3D world. It

is based on large almost-convex cells partitioning the free region of a given terrain. This GPU based method generates a Cell-and-PortalGraph(CPG) [11] for a given 3D scenes (with slopes, steps and other obstacles).

This approach computes a GPU voxilization of the geometry in order to obtain a first approximation of the walkable area. Then, the potentially walkable area is subdivided into layers, using an ordered flooding where each layer is refined by using the fragment shader at higher resolution and the NavMesh is computed using [12]. Finally, all those individual NavMeshes are merged into a single one, that represents the walkable space of the entire scene. See Fig. 2.6 .The coarser cell decomposition allows to reduce the number of nodes and makes the path finding faster.

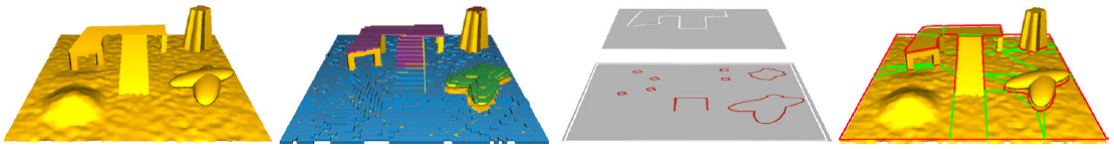


FIGURE 2.6: NEOGEN subdivision. From left to right we can see the original scene, the result of the layer extraction step after the coarse voxelization,the 2D floor plan of each layer, and finally the near optimal navigation mesh.

Other method which automatically generate navigation meshes from a 3D scene is proposed by [7]. It is accompanied with Detour, path-finding and spatial reasoning toolkit. Recast tool offers simple static navigation mesh which is suitable for many simple cases, as well as tiled navigation mesh which allows us to plug in and out pieces of the mesh. Recast is the program we use to test our method and it will be explained in detail in the next chapter.

Other approaches for navigation meshes have also been proposed for example to allow the interconnection of floor plans in multi-layer and non-planar environments, in order to address 3D scenes. [13], [14], [15].

2.2 Path finding Algorithms

Current state-of-the-art real time path finding algorithms try to improve the performance measures described in (Russell and Norvig, 2003, page 71) in order to guarantee a constant bound on response time. These measures are:

- Completeness: Whether or not a route is found, if one exists.
- Optimality. Whether or not the best path is found.

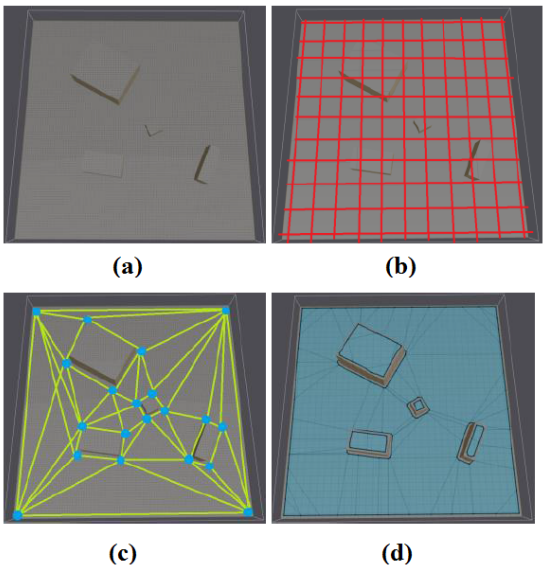


FIGURE 2.7: Representations of the environment division. Empty map a). Regular Grid map b). Roadmap c). NavMesh d)

- Time complexity: Number of iterations to reach the goal.
- Space complexity: Maximum number of nodes stored in memory at each iteration.

2.2.1 Introduction

In order to do path planning, the representation of the environment need to be discretized to facilitate efficient path finding queries. Then, efficient planning algorithms need to be developed in order to be able to generate solutions with strict time constraints for extremely large and complex problem domains.

The most known and popular dynamic search algorithm is A* search [16], It is robust and simple to implement, with strict guarantees on optimality and completeness of solution. Hence, it represents a popular and widely used method for path planning in virtual environments. The A* algorithm uses a heuristic to restrict the number of states that must be evaluated before finding the true optimal path. It guarantees to expand an equal number or fewer states than any other algorithm using the same heuristic. A* may be too slow. Its memory use is also variable and may be high depending on the size of its opened and closed lists and the heuristic function used.

Also, Anytime Planning algorithms find the best suboptimal plan and iteratively improve this plan while reusing previous plan efforts. One of the most popular A$*$ is called Anytime Repairing A* (ARA*) [17]. It performs a series of repeated weighted A* searches while iteratively decreasing a loose bound (ε). Then, it iteratively improves

the solution by reducing ε and reusing previous plan efforts to accelerate subsequent searches. The key to reusing previous plan efforts is keeping track of over-consistent states. ARA* solution no longer guaranteed to be optimal.

Furthermore, Incremental planning algorithms try to reuse the results of the previous plan calculation in order to reduce the planning effort. It also helps to compute the new plan when there is a small change in the environment. One common replanning method is presented in [18]. D* Lite performs A* to generate an initial solution, and repairs its previous solution to accommodate world changes by reusing as much of its previous search efforts as possible. D* can correct "mistakes" without re-planning from scracth but requires more memory.

Finally, Anytime Dynamic A* (AD*) [19] combines the properties of D* and ARA* to provide a planning solution that meets strict time constraints. It efficiently updates its solutions to accommodate dynamic changes in the environment. These updates are performed by series of repeated searches by iteratively decreasing the inflation factor. AD* cannot handle dynamic changes in goal.

2.2.2 Dynamic Search

DBA* algorithm [20] combines the memory-efficient sector abstraction developed for [21] and the path database used by [22] in order to improve space complexity and optimality.

Basically, it performs an off-line pre-processing step where computes a minimal memory abstract map [21] with regions, sectors and the database path between adjacent regions. Sectors are further divided into regions.

Each region has walkable and reachable nodes within a sector. These nodes are identified by performing a BFS (Breadth-first search). Edges between sectors are added by comparing each pair of values between adjacent sectors and regions as shown in the Fig. 2.8. Sectors (0, 1, 2, 3), regions (0a, 0b; 1a, 1b; 2a, 2b, 2c; 3a, 3b).

The next step is to construct a database of optimal paths for each edge using a simple A* [16]. Many sequences of subgoals are stored in a compressed format. Each subgoal can be reached using an optimal local search called hill-climbing agent [23]. These paths are used to populate a R2 path matrix where R is the number of regions. Dynamic programming is performed on the matrix to compute the cost and next hop region for all pairs of regions.

The final step performs an online search. The start and goal regions are identified performing a BSF for each region. The complete path is found by iterating several steps

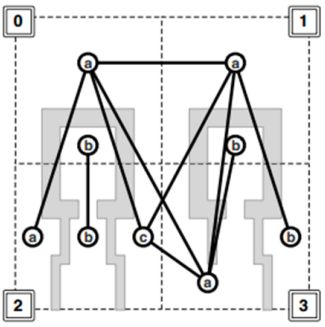


FIGURE 2.8: Minimal memory abstract graph.

where each sub path is obtained by looking into the path matrix to find the next hop to navigate from the start region to the goal region.

Finally, a path refinement is performed by path trimming and increasing the neighbourhood depth. The approach does not handle dynamic updates to the environment and it is only limited for grid based 3D worlds.

Path finding is not only focused on agents with individual targets, but also on groups of characters who have a common goal and where the lateral and longitudinal dispersion between them is important. A new approach is introduced by Huang [24] that presents a path planning method for coherent and persistent groups in arbitrarily complex navigation mesh environments. The group is modeled as a deformable and splittable area preserving shape. The efficiency of the group search is determined by three factors: path length, deformation minimization, and spitting minimization.

The problem is defined as:

$$\Sigma = \langle S, A, c(s, s'), h(s) \rangle \tag{2.1}$$

Where,

- S is the state space of the group. It means the collection of triangles which contain one head and the tail.
- A is the set of possible movement behaviour of each subgroup: LEFT, RIGHT, SPLIT, PAUSE, and MERGE (See Fig. 2.9)
- *c(s,s')* is the cost of transition from s to s'. It is composed by the distance cost, the deformation cost and the split penalty.

- *h(s)* is the heuristic estimate of reaching the goal. It computes the straight line distance to the goal and deformation cost from the current state of the group to the goal.

Paths to given goals are computed with a three-tuple cost vector of distance, deformation, and split costs. The distance cost is calculated by the Euclidean distance between the centers of the traversal exit edges of the heads of the subgroup. The deformation cost is only intended to consider the head of each group. It measures the group dispersion while passing through the environment corridors.

Finally the split penalty is the cost for the group to split and it only happens when the deformation cost is too high. For more details, please refer to [24].

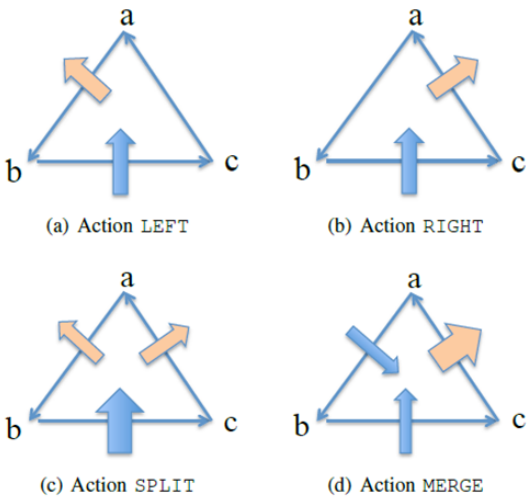


FIGURE 2.9: Movement behaviour: Set of possible actions.

These two techniques are focused on finding the shortest and optimal path regardless of how dynamic the environment is.

2.3 Environment-Aware Navigation

Most of the research regarding path finding algorithms are motivated by reaching an optimal solution and improving the measures describes in the section 2.2. However, in order to achieve a real time planning, the agents should be aware in their surrounding area while they are advancing towards their goal.

In a dynamic environment, there are many things to consider such as moving obstacles, the size of the corridors or even the crowd density information of the 3D world in order to avoid congested routes that could lead to traffic jams.

There is little work that explores the satisfaction of spatial constraints between objects and agents at the global navigation layer. The planning framework [25] enables agents to be more aware of the environment. It satisfies multiple special constraints imposed on the path. These constraints could be: Stay behind a building, walking along walls or avoiding the line of sight of other agents.

The obstacles and agents in the world are annotated with additional objects to allow for more detail in behaviour specifications.

A hybrid discretization of the environment is used to represent the 3D world. It balances computational efficiency while still ensuring that it has enough expressive power to accommodate both static and dynamic constraints. It is represented as a grid-like graph with extra "highway" edges added in, allowing the planner to find better suboptimal paths at "any-time".(See Fig. 2.10). It benefits from the computational speed of triangulations, and the uniform grid ensures enough resolution for dynamic constraints.

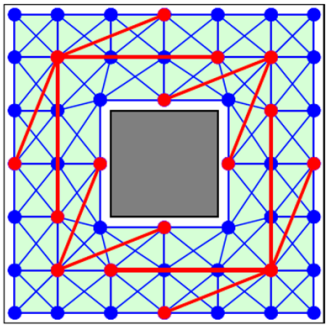


FIGURE 2.10: Hybrid discretization: Blue edges represent the grid portion of the graph while red edges represent the highways

The dynamic constraints can be represented as a continuous potential field [26] which can be easily superimposed to calculate the total effect in the same region and can be easily found during search exploration. These constraints can be:

- Hard constraints: Use to avoid invalid transitions in the search graph.
- Soft constraints: Have an effect on the cost of choosing a transition.

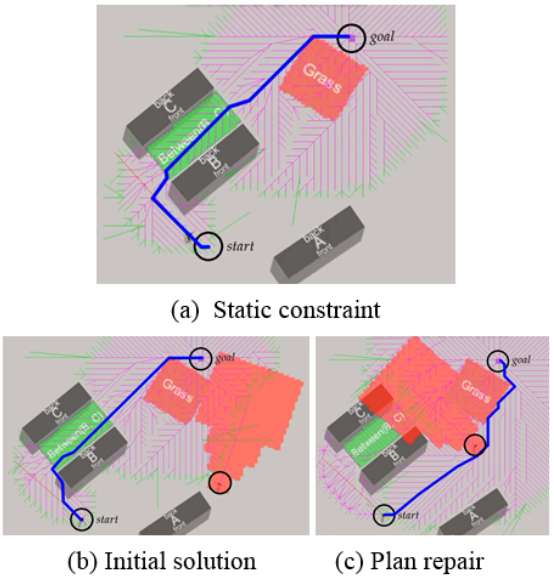
Hard constraints have two fields that allow defining an area of influence of the constraint where all the transitions are forbidden, while the soft constraints have three fields:

- Prepositions: (NEAR, IN): Define the boundaries of the region of influence.
- Annotations: (BACK, FRONT, LEFT, RIGHT, LINEOFSIGHTOF, BETWEEN). Define the area of influence of the constraint, relative to the position of the object.

- Weight: Define the influence of a constraint to another one (+/-). For instance. W = 1 is a weak attractor whilst W = -5 would be a strong repeller. It allows to set priorities between them and having multiple constraint in the same area.

For instance, a solution with static constrains could be "In Between (B, C), Not In Grass" shown in the Fig. 2.11a. But if a dynamic constraint is added to the problem such as an agent, the preposition would be "In Between (B, C), Not In Grass, Not In LineOfSightOf(Agent)" (Fig. 2.11b and 2.11c).

In this case, a plan repair would be released to satisfy all the constraints.



(a) Static constraint

(b) Initial solution (c) Plan repair

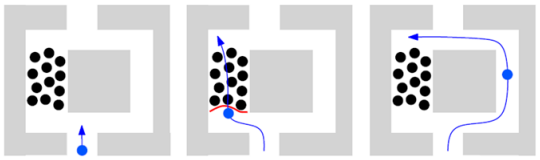
FIGURE 2.11: Static and Dynamic Constraints)

An Anytime Dynamic A* (AD*) [19] is used as a planner. This algorithm not only finds a suboptimal plan but also can quickly repair its plan in the case of environment changes.

This approach is a goal-directed navigation system that satisfies multiple spatial constraints imposed on the path. It's suitable for static and dynamic objects.

Other method proposed in [27] that is aware of crowd density information in a dynamic environment at certain time. Not always the shortest path is the fastest one.

This approach takes into account the number of characters in a specific region. It allows the agents to naturally spread out among the available routes inside a navigation mesh. (See Fig. 2.12).



(a) Scenario (b) Short path (c) Density path

FIGURE 2.12: Density Based Path Planning

The density value is equal to the total area occupied by the characters inside the walkable region divided by the total area of the walkable region. Each agent has a disk with a constant radius which is independent of their height.

The navigation mesh is based on a medial axis of the polygonal world and the Explicit Corridor Map (ECM)[27] which allows to have no overlapping regions and each of them share just one edge with the medial axis.

The density value of each walkable region will be used to weigh the medial axis edge that touches this region. The resulting medial axis serves as a weighted graph. Each density value is constantly updated so that the characters can periodically re-plan their routes when it is necessary.

A modified A* is adapted to compute the path through the weighted graph of the medial axis.

2.3.1 Dynamic Search on the GPU

Kapadia et al. [25] in his method provides a wave-front based search technique that can efficiently handle world changes and agent movement, while reusing previous efforts, and is amenable to massive parallelization. This approach can also be extended to handle any number of moving agents, at no additional computational cost.

The method sets up a grid-based discretization of the environment on the GPU where each cell has a cost value = -1 (needs to be updated) and obstacles = ∞. Two copies of the map are created in device memory for only read and write operations. They are updated and swapped during the computation of the plan.

The planner kernel is independently executed for each thread which maps to every grid cell in the environment. Each thread reads values from its neighbour cells in order to compute the cost value of its grid cell using the appropriate read or write map. This ensures that the map we are reading from will not change we are executing the kernel.

Each thread is responsible for computing and maintaining its least cost predecessor and its cost as the sum of predecessors cost and the cost of the transition as follows:

$$g(s) = min_{(s' \in succ(s) \bigwedge g(s') \geq 0)}(c(s, s') + g(s')), 0 \leq c(s, s') \leq \infty \tag{2.2}$$

Each computation of the plan is performed by a loop until the cost values of all the states converges to the minimum cost of reaching the goal where the g value of the goal was initially set to 0. Once the planner has finished executing, an agent can simply generate its plan by following the path of least cost from the goal to its position.

There is an extension of this algorithm to handle dynamic environments where obstacle changes may invalidate plans that are currently being executed. Invalidate plans mean inconsistent states. A state is inconsistent if its predecessor is not the neighbour with lowest cost or if any of its successors are inconsistent. The algorithm (See Fig. 2.13) shows the steps to fix the invalid plans. It can be appended to the end of getting the optimal solution to guarantee that node inconsistency is propagated and resolved in the entire map.

$s \leftarrow threadState$;
if $(pred(s) \neq NULL)$ **then**
  if $(g(s) == obstacle \vee pred(s) == obstacle \vee g(s) \neq g(pred(s)) + c(s, s'))$ **then**
    $pred(s) = NULL$;
    $g(s) = -1$;
    $incons =$ `true`;

FIGURE 2.13: Algorithm: Propagate state inconsistency

The technique explained above has an extension to handle multiple agents. The kernel is executed until all agent states have been reached, and the maximum g-value of a state that is occupied by an agent is less than the g-value of any other state. The number of iterations for convergence depend on the distance from the goal to the farthest agent.

This framework is currently limited to a small number of target locations, since a separate map needs to be maintained for each target, resulting in a significant memory overhead.

The size of grid-based discretization of the environment is really important in the performance of the whole path planning because the main limitation is memory overhead. This discretization can be adapted to different resolutions quads, using finer resolution only where necessary. This significant produces memory savings and time benefits as state space is reduced.

In an adaptive environment representation on the GPU, it is necessary to use a quadcode scheme for performing efficient indexing, update, and neighbour finding operations and efficiently handling dynamic environment changes by performing local repair operations on the quad tree. For more details, please refer to [28].

2.3.2 Planning in Complex Domains

In dynamic scenarios, where the autonomous agents interact with objects and other agents could be extremely high-dimensional and continuous, a trade-off between action fidelity and scalability is needed. Kapadia presents a multi-domain anytime dynamic planning framework [29] which can efficiently work across multiple domains, by using plans in one domain to accelerate and focus searches in more complex domains. It explores different domain relationships including the use of waypoints and tunnels. Fig. 2.14 illustrates some of the challenging scenarios and their solutions.

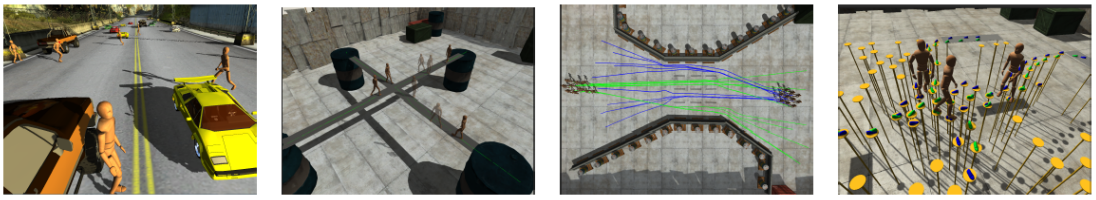


FIGURE 2.14: Different scenarios: (a) Agents crossing a highway with fast moving vehicles in both directions. (b) 4 agents solving a deadlock situation at a 4-way intersection. (c) 20 agents distributing themselves evenly in a narrow passage, to form lanes both in directions. (d) A complex environment requiring careful foot placement to obtain a solution.

In order to showcase the ability of this framework to efficiently work across heterogeneous domains, 4 domains are described to provide a nice balance between global static navigation and fine-grained space-time control of agents in dynamic environments.

- Static Navigation Mesh Domain (Σ_1). Uses a triangulated representation of free space and only considers static immovable geometry. The agent is modelled as a point mass, and valid transitions are between connected free spaces, represented as polygons. The cost function is the straight line distance between the center points of two free spaces and the heuristic function is the Euclidean distance between a state and the goal.

- Dynamic Navigation Mesh Domain (Σ_2). Uses a triangulated representation of free space and coarsely accounts for dynamic properties of the environment by storing a time-varying density field, which contributes to the cost of choosing a triangle for

navigation. The time-varying density field stores the density of moveable objects (agents and obstacles) for each polygon in the triangulation at some point of time t. The presence of objects and agents in polygons at future timesteps can be estimated by querying their plans (if available). The space-time positions of deterministic objects can be accurately queried while the future positions of agents can be approximated based on their current computed paths, assuming that they travel with constant speed along the path without deviation. The resolution of the triangulation may be kept finer than 1 to increase the resolution of the dynamic information in this domain. Hence, a set of global waypoints are chosen in this domain which avoids crowded areas or other high cost regions.

- Uniform Grid Domain (Σ_3). Discretizes the environment into grid cells where a valid transition is considered between adjacent cells that are free. This domain only accounts for the current position of dynamic obstacles and agents, and cannot predict collisions in space-time. An agent is modelled as a point with a radius (orientation and agent speed is not considered in this domain). The cost and heuristic are distance functions that measure the Eucledian distance between grid cells.

- Space Time Domain (Σ_4). Accounts for all obstacles (static and dynamic) and other agents. Transition validity queried in space-time by checking to see if moveable obstacles and agents occupy that position at that particular point of time, by using their published paths.

The different domain representations for a given environment can be illustrated in Fig. 2.15

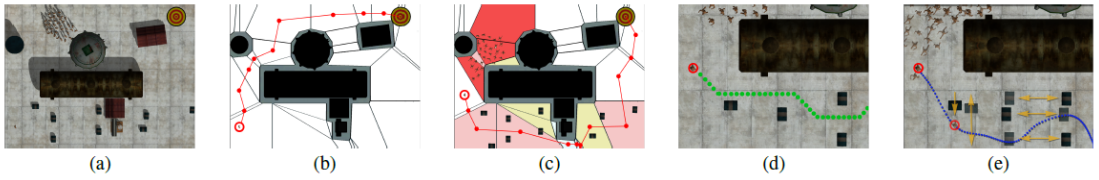


FIGURE 2.15: Domain Representations: (a) Problem definition with initial configuration of agent and environment. (b) Global plan in static navigation mesh domain Σ_1 accounting for only static geometry. (c) Global plan in dynamic navigation mesh domain Σ_2 accounting for cumulative effect of dynamic objects. (d) Grid plan in Σ_3. (e) Space-time plan in Σ_4 that avoids dynamic threats and other agents.

Tunnels [30] are used to connect each of the 4 domains mentioned before, ensuring that a complete path from the agents initial position to its global target is computed at all levels. A tunnel is described as follows

Planning Subpace:

$$\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w) \tag{2.3a}$$

Heuristic:

$$h_t(S, S_{start}) = h(S, S_{start}) + |d(s, \Pi(\Sigma))| \tag{2.3b}$$

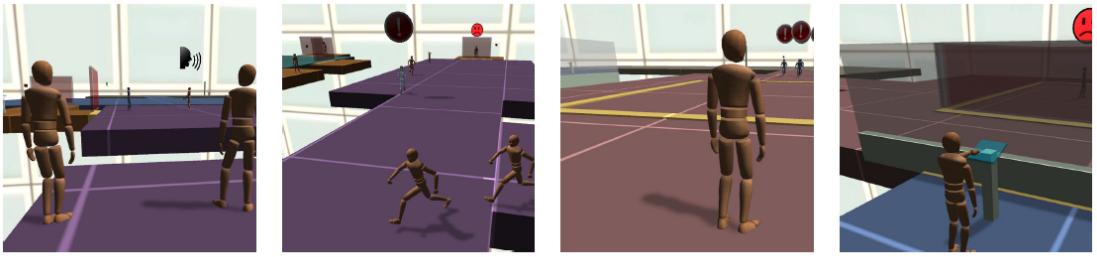
Tunnels are a sub graph in the high dimensional space such that the distance of all states in the tunnel from the low dimensional plan is less than the tunnel width t_w. Furthermore, node expansion can be prioritized to states that are closer to the path by modifying the heuristic. Also, d is the perpendicular distance between s and the line segment connecting the two nearest states. d will return a two-tuple value for spatial distance as well as temporal distance. Finally, node expansion can be prioritized to states that are closer to the path by modifying the heuristic function.

Planners are most certainly not only limited to navigation problems but also interaction between virtual characters. It helps to personify the characters and make the simulation more realistic.

One approach [31] proposes an event-centric planning framework for directing interactive narratives in complex 3D environment. They gives a series of tools that are defined by preconditions, postconditions, costs, and a centralized behaviour structure that simultaneously manages multiple participating actors and objects.

The problem domain is abstracted so that the planner does not have to deal with complex motion planning problems and have to worry about the articulation of the joints of an animated character. It just generates plans in the space of pre-authored narratively significant logical interactions between any number of characters.

This makes the planning problem much more tractable allowing the automated synthesis of stories in 3D virtual worlds. For example, if the user changes the world e.g. locks a door, or moves the guards – the planner is able to easily repair the narrative to accommodate user agency. Another example is shown in Fig. 2.16.



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

FIGURE 2.16: Characters exhibiting two cooperative behaviours: Two actors hide while a third lures a guard away (a), then sneak past once the guard is distracted (b). An actor lures the guards towards him (c), while a second presses a button to activate a trap (d).

2.4 Hierarchical Path finding

The cost of determining the shortest path between two points can grow exponentially with the size of the terrain and the allocated memory is very limited. Therefore, a hierarchical subdivision of the world environment is necessary under those restraints.

Hierarchical path finding has been studied in the last decade, it allows to compute the shortest and optimal route between two locations in large terrains based on a hierarchical graph. This significantly decreases the execution time and memory footprint in crowd simulation environments.

There has been some research recently focused on hierarchical path finding techniques using the A star algorithm such as HPA* [32] which is based on grid maps and clustering.

HPA* creates an abstract graph from a grid in order to minimize the complexity of the problem. This abstract graph is built by dividing the environment into squares clusters connected by entrances. Basically, the algorithm has two steps: the pre-processing step where the grids are grouped in a cluster with a user defined size. These clusters will be the nodes of the high level graph. Then, the entrances (connections between two clusters) are placed with one or two transitions.

The clusters are connected with inter-edges with cost 1.0 and the cost of intra-edges are calculated running regular A* [16] searches inside each cluster, for all pairs of abstract nodes that shared the same cluster. (See Fig. 2.17).

The second step is the online search which inserts the start and goal nodes into the abstract graph and searches the optimal path with A* between them. The low level graph is much smaller than the original one. This approach is only based on grids. See Fig. 2.18.

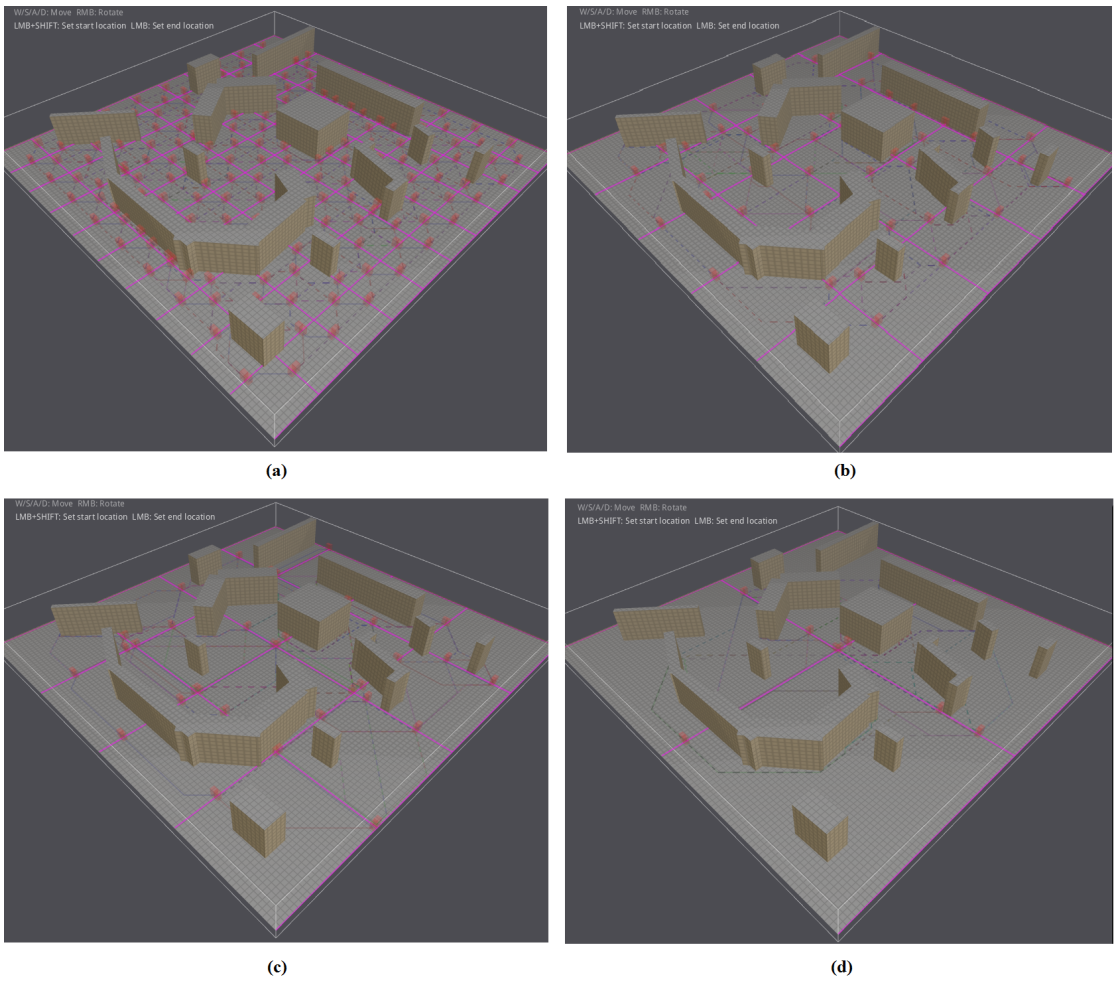


FIGURE 2.17: Upper graph level with different subdivision: 100 clusters (a). 25 clusters (b). 16 clusters (c). 4 clusters (d). (Model: map (1000 nodes)

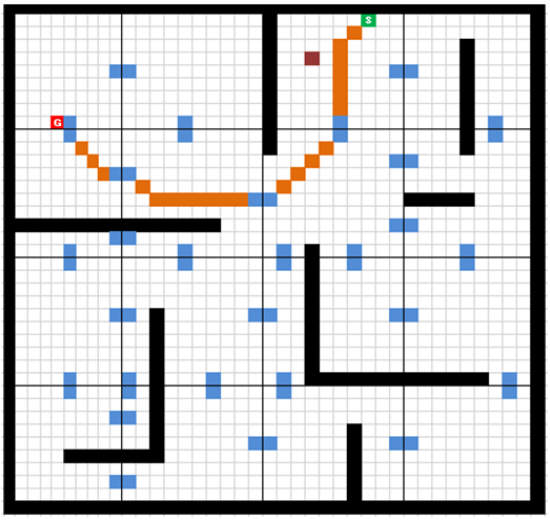


FIGURE 2.18: Abstract Graph: Black cells represent obstacles and walls. Entrances are blue cells. Green cell is the start and the red one is the goal. Path between them is with orange color.

Finally, HPA* smoothes the path in an attempt to undo some of the error introduced with the merging of cluster border vertices. Hovewer, the final path is still not optimal. (See Fig. 2.19).

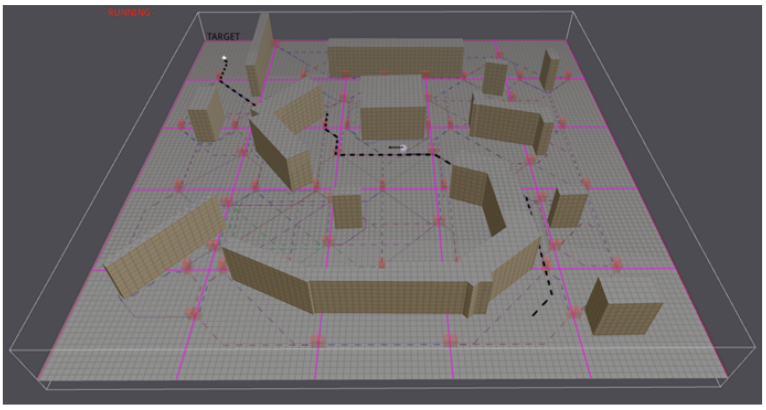


FIGURE 2.19: HPA* in Recast tool: Cluster division (purple). Entrances (red). Inter and intra edges (blue). Optimal path (black).

Since the abstract graph is much smaller than the original graph, search problems can be greatly simplified by using the abstract graph instead of the low level one. This algorithm is the based for the hierarchical approaches. However it only works on grids.

Another similar method based in HPA* but taking into account the size of the agents and terrain traversal capabilities is Hierarchical Annotated A* (HAA*) [33]. Basically, it is an extension of HPA* which allow multi-size agents to efficiently plan high quality paths in heterogeneous-terrain environments.

Each agent has a size property and a capability (ability to walk in a certain type of terrain). The clearance values (See Fig. 2.20) which is the size of maximum traversal area at each octile (cell in the grid) is calculated for each capability. The clearance values for different type of terrain are shown in the Fig. 2.20.

In order to find the shortest path, an adaptable A* is performed by taking into account the size and capability of each agent. It means the nodes with a clearance value greater than the size will be expanded.

The path planning is done over an abstract graph which is created in the same way as HPA* [32]. The entrances between clusters, inter-edges, intra-edges are calculated considering the agent properties.

Other interesting implementation is called DT-HPA* [34] where a decision tree is used to create a hierarchical subdivision instead of a normal clustering.

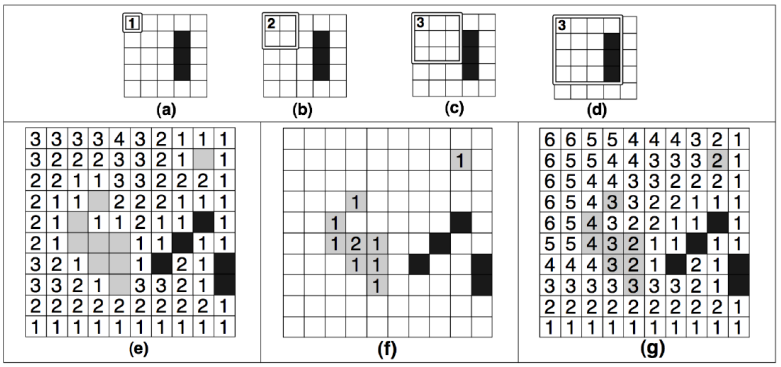


FIGURE 2.20: Clearance Values: (a)-(d) Computing clearance; the square is expanded until a hard obstacle is encountered. (e)-(g) Clearance values for different capabilities

This algorithm keeps in mind the terrain distribution in order to do the partition. The map is divided into small rectangular clusters which correspond to a leaf in the node of the tree. The classification algorithm considers each cell of the grid as a sample, the horizontal and vertical coordinates as a condition and if a cell is walkable or not as a category attribute for the classifier.

The decision tree has the following stop criteria:

- Entropy of the map is zero. A cluster only contains one type of terrain.
- Size of the map is less than a certain threshold. Number of clusters in the map.

Once the subdivision is created, the online path finding is performed in the same way as HPA* does.

2.5 Conclusion

In this chapter, we have reviewed the current state-of-the-art in path finding algorithms for crowd simulation. We have discussed the different terrain representations, subdivisions and dynamic searches on them, techniques for static and dynamic environments and hierarchical approaches based on weighted direct graphs; all of them try to improve a set of performance measures: completeness, optimality, time complexity and space complexity. We also have presented a GPU approach [25] where the path planning time has been reduced significantly but the synchronization between two maps is still a hard task to deal with.

The analysis of the current methods indicates a clear trend towards hierarchical subdivisions [32] [33] [34]. There has been a large research in the artificial intelligence and

videogames industry the last years. However those techniques are only based in grid subdivision.

One interesting direction of research in the hierarchical path finding would be applying the idea of hierarchical abstraction also to any kind of navigation mesh because, as this report explained, NavMeshes are the best suitable subdivision for complex environments [35].

Furthermore, after studying the previous work in this field, one can conclude that the combination of HPA algorithms in dynamic-aware environment constraints [25] [27] could give more realism to the behaviour of thousands of agents.

Finally, the discussion of techniques in this chapter has revealed that path finding remains a computational expensive and complicated problem because of the calculations in real time applications using large and detailed virtual worlds and character count. Therefore any research towards improving performance to solve path finding can be a useful contribution to this field.

Chapter 3

Our Approach

3.1 Introduction

Based on our conclusions over the current state of the art in path finding algorithms, we aim at using hierarchical approaches to speed up path planning in large scenarios. In order to apply our new method, we need an initial discretization of the 3D world. As we have explained in the state of the art methods for spatial subdivision, navigation meshes are the most accurate and used for that purposes. Another important of navigation meshes in comparison with grid approaches, is that the number of cells is much smaller and thus we are already starting with a significantly smaller graph for path finding.

Many tools have been proposed for a NavMesh subdivision. Recast Tool is an automatic open-source navigation mesh generator toolset for games [7]. The automatic NavMesh generation is done via Watershed Partitioning which creates a robust triangulation without overlaps and holes. This method used by Recast is inspired by the work by Haumont [11]. It is automatic, which means that we can input any geometry and it will output a robust mesh. It is also fast which means swift turnaround times for level designers. Recast tool also provides an optimized A* implementation into Detour project classes [7]. Fig. 3.1 shows the tool [1].

Recast tool generate a NavMesh from a triangle soup. It receives an arbitrary polygon soup with triangles marked as walkable. All of this part is done in the preprocessing step and has five steps to construct the navigation mesh as follows:

[1] All the test models were obtained using http://tf3dm.com/

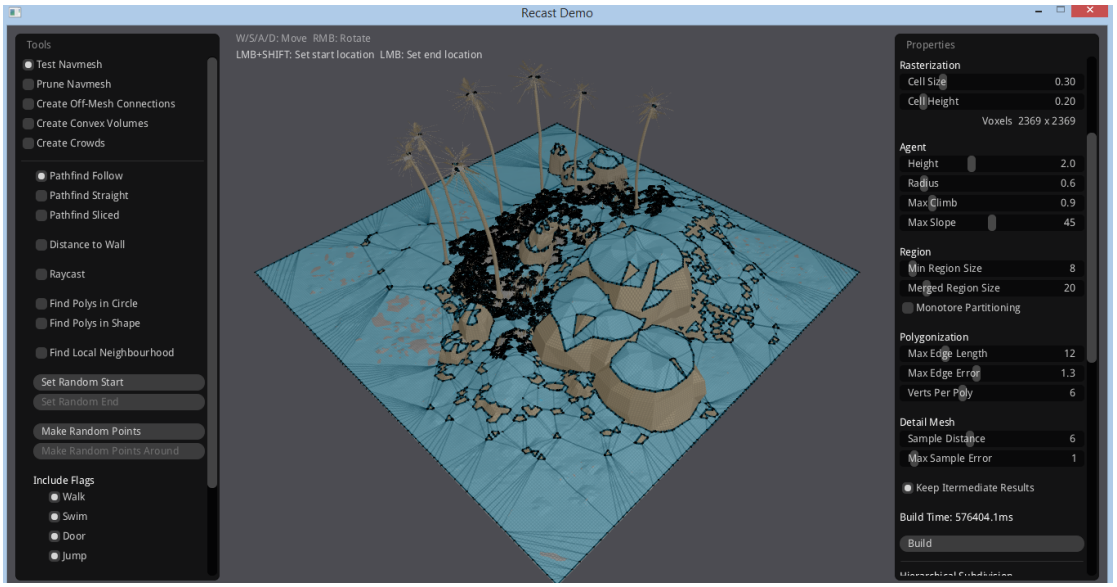


FIGURE 3.1: Recast Tool software. (Model: Tropical Islands (12666 polygons)).

3.1.1 Voxelize the polygons.

The voxel mold is build from the input triangle mesh by rasterizing the triangles into a multi-layer heightfield. 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.

3.1.2 Build navigable space from solid voxels.

Some simple filters are applied to the voxel mold to prune out locations where the character would not be able to move, for instance: too steep slopes, too low places, etc. This is done by calculating the distance and the slope in each voxel. (See Fig. 3.2).

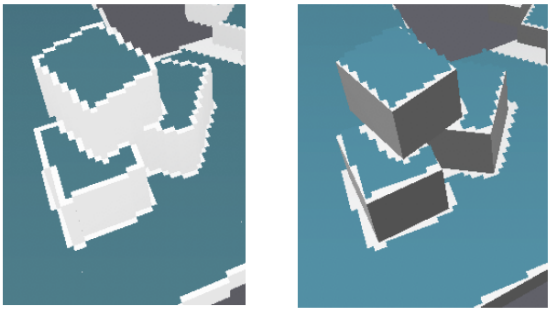


FIGURE 3.2: Build navigable space from solid voxels: Voxelization with walkable cells marked (a) and walkable cells overlayed on top of input geometry (b).

3.1.3 Build watershed partitioning and filter out unwanted regions.

The Watershed Transform [36] finds the catchment basins by building distance the transform of the input areas. It starts from the highest distance one slide at time. Then, it finds any new catchment basins and it fills them with a new ID. Finally, it expands existing regions. The catchment basins become the centers of the regions (See Fig. 3.3). Then, a filter pass is applied to remove small unconnected regions and merge small regions together. The result is set of nonoverlapping simple regions that can be used as basis for generating waypoint graphs.

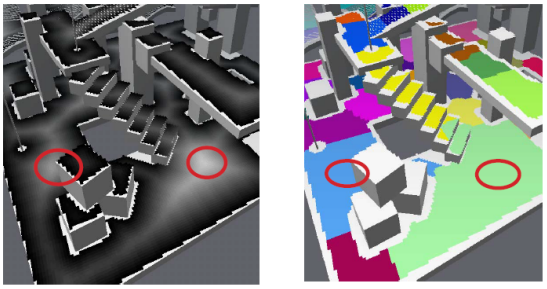


FIGURE 3.3: Build watershed partitioning and filter out unwanted regions: The catchment basins become the centers of the regions. (White areas represent lower region).

3.1.4 Trace and simplify region contours.

It searches the contours by finding a starting point to start tracing (region edge cell). Then it traces around the boundaries of the regions. The cell corner points which will form the polygon and the neighbour region ID are stored. Finally, the contours are simplified using Ramer-Douglas-Peucker algorithm [37]. The algorithm finds initial segments and locks vertices which are between two different regions, if the the region is not connected, it locks two extrema vertices. The algorithm iterates through all simplified segments and subdivides the segment at the point with maximum distance error between the vertex and the segment. The initial vertices allow later to find common edges between the polygons. (See Fig. 3.4). The result is a set of simple polygons. (See Fig. 3.5).

3.1.5 Triangulate the region polygons and build triangle connectivity.

Recast uses a modified algorithm from Computational Geometry in C for the triangulation of the polygons. The final step is to combine the triangles and find edge connectivity. The resulting polygons are finally converted to convex polygons which makes them

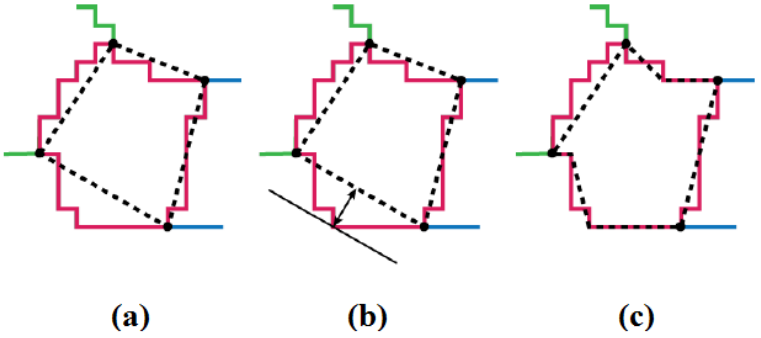


FIGURE 3.4: Ramer-Douglas-Peucker algorithm: Initial vertices at region edges (a); Find vertex with maximum error, and subdivide (b); Iterate untill certain error criteria is met (c).

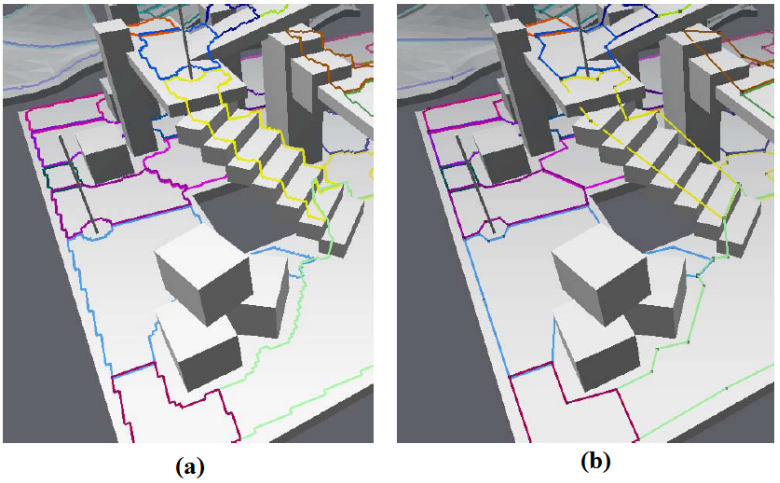


FIGURE 3.5: Contours: Traced contours (a); Simplified contours(b).

perfect for pathfinding and spatial reasoning about the level Fig. 3.6.

These five steps are illustrated in the Fig. 3.7. Recast is suitable for complex indoors and outdoors scenes with many levels. However, it generates walkable areas where the agents cannot even reach (see. Fig. 3.8). This is due to Recast only taking into account the characteristics of the geometry enclosed by the voxels. It does not consider the connectivity between potentially walkable polygons. This is a problem because the resulting navigation mesh is consider as the initial graph located in the lowest level of the hierarchy. All of these unreachable regions are deleted during the preprocessing step in our approach. This development will be explained in detail in the description of our method.

Once the spatial partition has been done by the Recast tool, we need to create the initial graph of our hierarchy scheme. Our approach has mainly two steps.

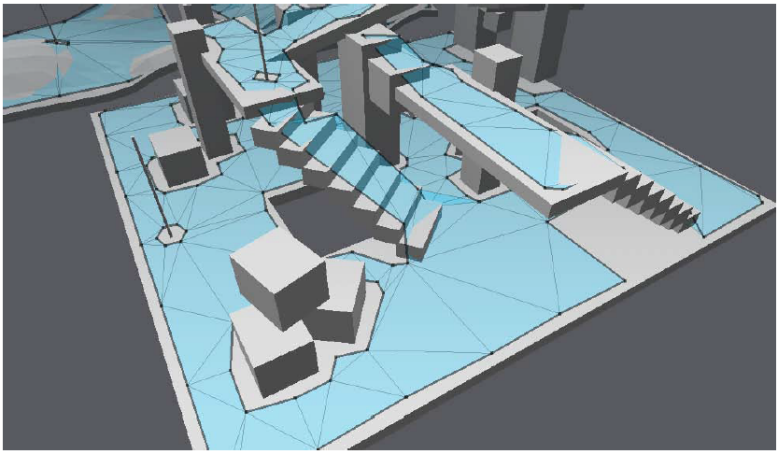


FIGURE 3.6: Triangulation

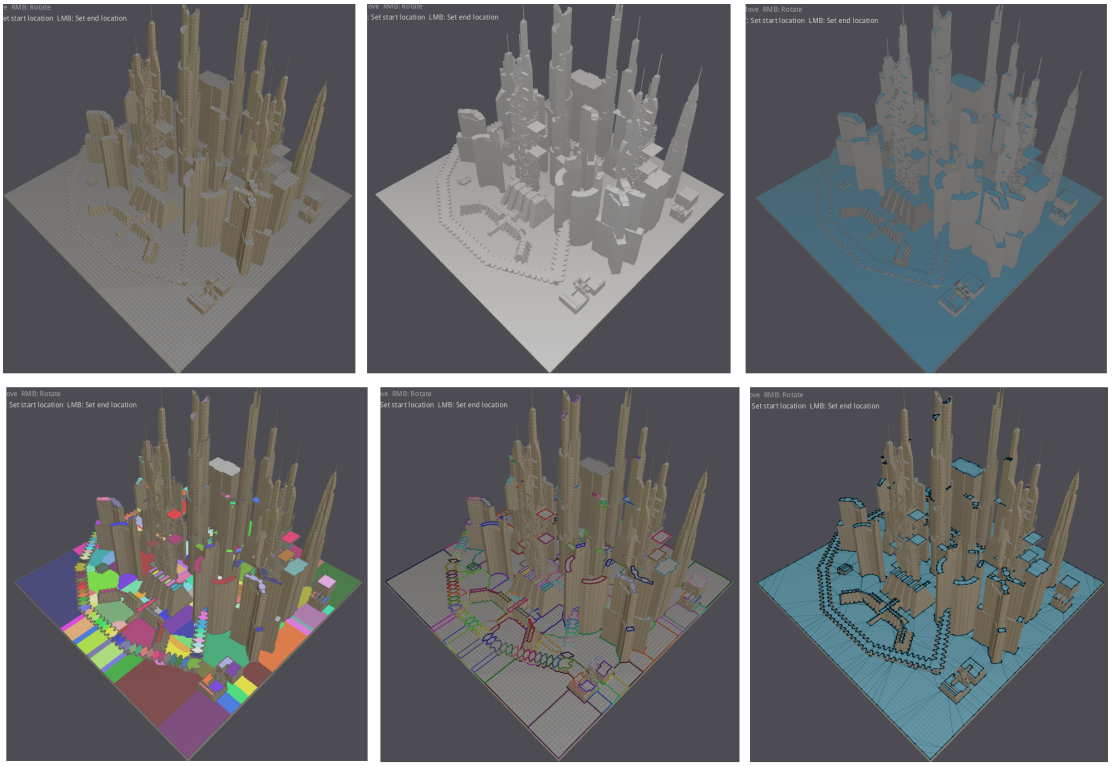


FIGURE 3.7: Recast steps (from left to right): Input mesh and the five steps of Recast process. (Model: Scifi City (2090 polygons)).

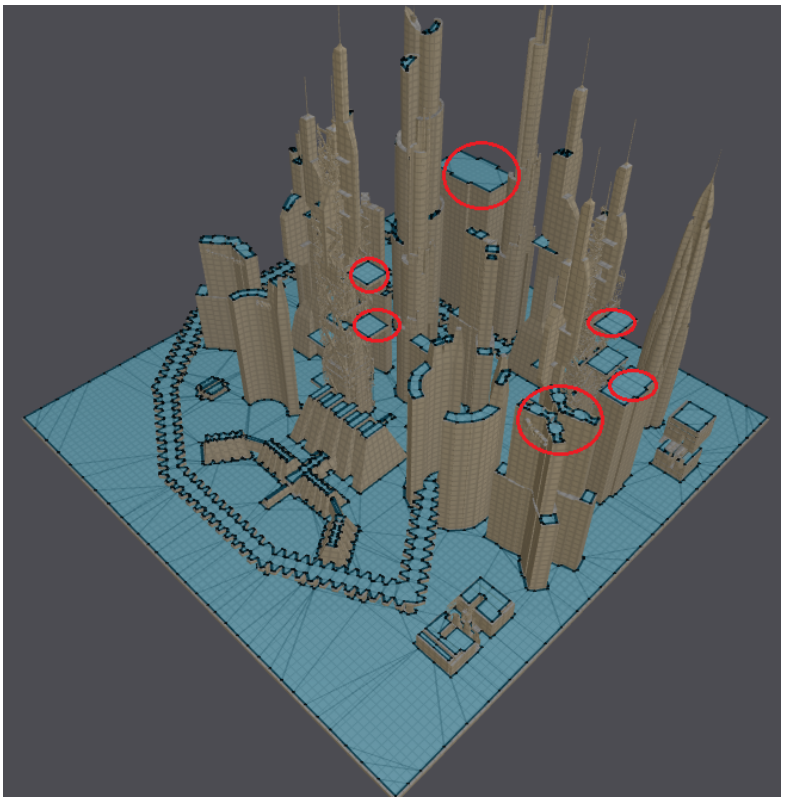


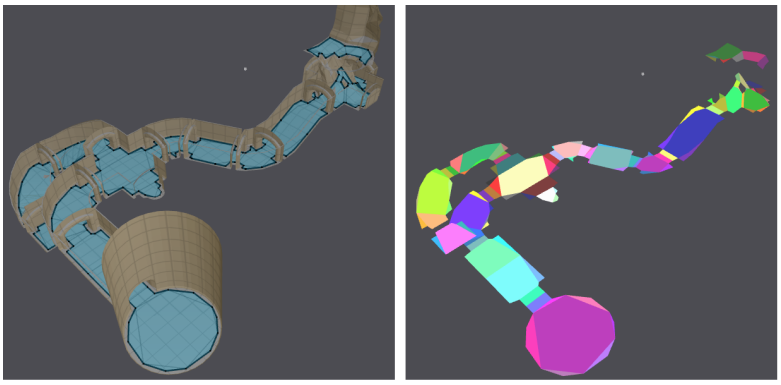
FIGURE 3.8: Recast tool: Unreachable walkable areas. (Model: Scifi City (2090 polygons)).

- Hierarchical Subdivision. This is done in the preprocessing part. We create a hierarchy of graphs. The graph in the lowest level is given by the Triangulation in the Recast tool.
- Path finding computation. Given any level in the precomputed hierarchy, we calculate the path finding over the graph in that level.

3.2 Hierarchical Subdivision

The first step is to built the framework for hierarchical searches that is defined as a tree of graphs. We start to compute the lowest graph of the hierarchy ($G_0 = (V_0, E_0)$) by searching the polygons in the Recast triangulation. Each polygon becomes a new node of the graph. For each near polygon that shares the common border, we create an edge between them. See Fig 3.9.

Once the lowest level graph is created, we recursively build the upper levels of the hierarchy by partitioning each level until it reaches the minimum number of the nodes



(a) (b)

FIGURE 3.9: Hierarchical Subdivision: The NavMesh triangulation of the model (a). The graph of the lowest level (level 0) (b). Nodes are painted in different colors. Edges connects a node with its neighbours (Model: Dungeon (120 polygons)).

in a graph or certain threshold (maximum number of levels). The number of nodes which will be merged in each step is defined by the user.

In order to obtain an efficient subdivision of each graph, a graph partitioning algorithm is used to reduces the size of the graph by collapsing vertices and edges. We use the k-way multilevel algorithm (MLkP) [38] which is faster than others multilevel recursive bisection algorithms. The process is described as follows:

First of all, a series of successively smaller graphs is derived from the input graph, this is called "coarsening phase". Here, the size of the graph is successively decreased. Each graph is constructed from the previous graph by collapsing together a maximal size set of adjacent pairs of vertices. In order to have a good partitions, the weight of a new vertex should be equal to the sum of its previous vertices. Also, the new edges are the union of the edges of the its previous vertices to preserve the connectivity information in the coarser graph. The coarsening phase ends when the coarsest graph has a small number of vertices or if the reduction in the size of successively coarser graphs becomes too small.

After the coarsening phase, a k-way partitioning of the smallest graph is computed (initial partitioning phase). It is performed by using a multilevel bisection algorithm [38]. Each partition contains roughly $|V_0|/k$ vertex weight of the original. graph. The division is done by Kernighan–Lin (KL) partitioning algorithm [39] which finds a partition of a node into two disjoint subsets of equal size, such that the sum of the weights of the edges between those subsets is minimized.

Finally, in the uncoarsening phase, the partitioning of the smallest graph is projected to the successively larger graphs by refining the partitioning at each intermediate level. It

assigns the pairs of vertices that were collapsed together to the same partition as that of their corresponding collapsed vertex. After each projection step, the partitioning is refined using various heuristic methods to iteratively move vertices between partitions as long as such moves improve the quality of the partitioning solution. The uncoarsening phase ends when the partitioning solution has been projected all the way to the original graph.

The three phases of the multilevel paradigm are illustrated in Fig. 3.10.

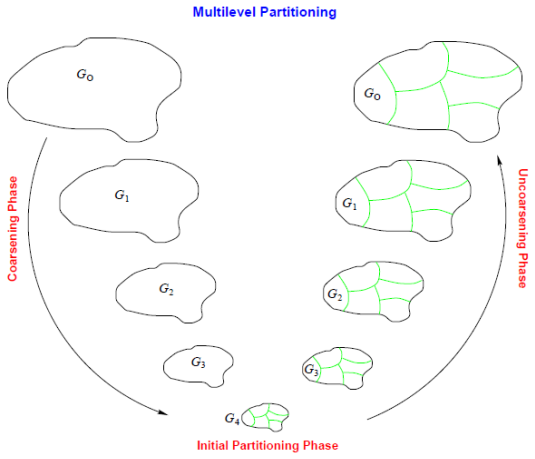


FIGURE 3.10: The three phases of multilevel k-way graph partitioning. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph

The procedure allows us to have partitions which ensures high quality edge-cuts. An edge-cut of the partition is defined as the number of edges whose incident vertices belong to different partitions. All of this operations make the algorithm more complex and hard to implement. We use an external implementation called METIS library [2]. It is a software package for partitioning unstructured graphs. It implements a collection of multilevel partitioning algorithms and is free only for educational and research purposes.

The algorithm 1 shows the steps of partitioning for each graph at each level in the hierarchy.

The iteration is done until either it reaches the maximum number of levels in the hierarchy (variable *levels*) or the graph cannot be subdivided. The number of merged nodes per level to create a new partition is defined by the variable $numMergedNodes$. The PartGraphKway function splits the parent graph into k parts using a multilevel k-way partitioning. The k parameter is given by the $numParts$ variable. This function returns

[2]METIS has been developed at the Department of Computer Science and Engineering at the University of Minnesota and is freely distributed. Its source code can downloaded directly from http://www.cs.umn.edu/~metis, and is also included in numerous software distributions for Unix-like operating systems such as Linux and FreeBSD.

Algorithm 1 Hierarchical subdivision algorithm

```
1:  procedure BUILDHIERARCHY(numMergedNodes,levels)
2:      repeat
3:          numParts ← parentGraph.numNodes/numMergedNodes
4:          if (numParts < 2) then return
5:          partitions ← METISPartGraphKway(parentGraph.numNodes, numParts)
6:          partitions ← checkPartitions(partitions, parentGraph.numNodes, numParts)
7:          currentGraph ← buildGraph(partitions)
8:          parentGraph ← currentGraph
9:          numLevels + +
10:     until numLevel < levels
```

the partitions in which the parent graph has been divided and that will become in the new nodes of the current graph. These partitions need to be checked before being part of the new graph. It means that for each partition, its subnodes must be linked and must have edges. Otherwise the current partition will not be taken into account for the next iterations. The new graph is created in the *buildGraph* function. The algorithm 2 illustrates the steps required to build a graph for each level.

Algorithm 2 Build Graph algorithm

```
    procedure BUILDGRAPH(partitions)
2:      for i ← 1, partitions do
            currentGraph.AddNode(i)
4:      for i ← 1, partitions do
            nodes ← findNodes(p)
6:          for node ← 1, nodes do
                for neighbour ← 1, node.numEdges do
8:                  if node.partition ≠ neighbour.partition then
                        currentGraph.AddEdge(node.parentNode, neighbour.parentNode)
10:                     portals.Add(node)
                for j ← 1, portals do
12:                 for k ← 1, portals do
                        if j ≠ k then
14:                         cost ← findPath(j, k)
                            currentGraph.AddIntraEdge(node, j, k, cost)
16:             portals.Clear()
```

Once we have the partitions, the new nodes and edges between partitions are created. Each partition has a set of portals which depends of the number of edges. A portal is the middle point in a common edge between to partitions. So, for each pair of portals in the partition, we run an A* between them in order to get the cost and the shortest path. This is called an IntraEdge. Each partition has stored the subpath and cost for reaching from one portal to another. In the Fig. 3.11, the partitions, portal and intraedges are illustrated.

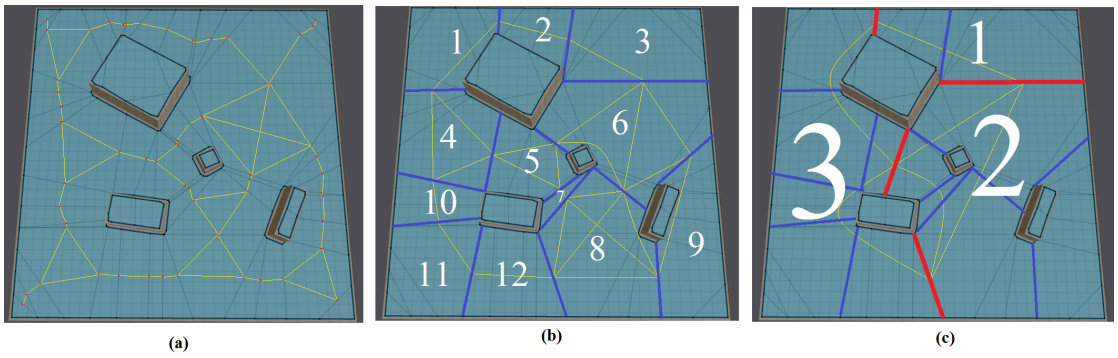


FIGURE 3.11: Hierarchical Subdivision: (Simple map, $numMergedNodes = 5, levels = 5$). Portals are presented with red dots. IntraEdges are painted with yellow lines. Partitions are exposed with black, blue and red separation lines respectively. Level 0 = 76 nodes (a), Level 1 = 12 nodes (b), Level 2 = 3 nodes (c). (Model: Simple Map (76 polygons)).

As an example of our process, we show a complex model with 5515 polygons to create the hierarchical subdivision performed during preprocessing. It is presented in Fig. 3.12.

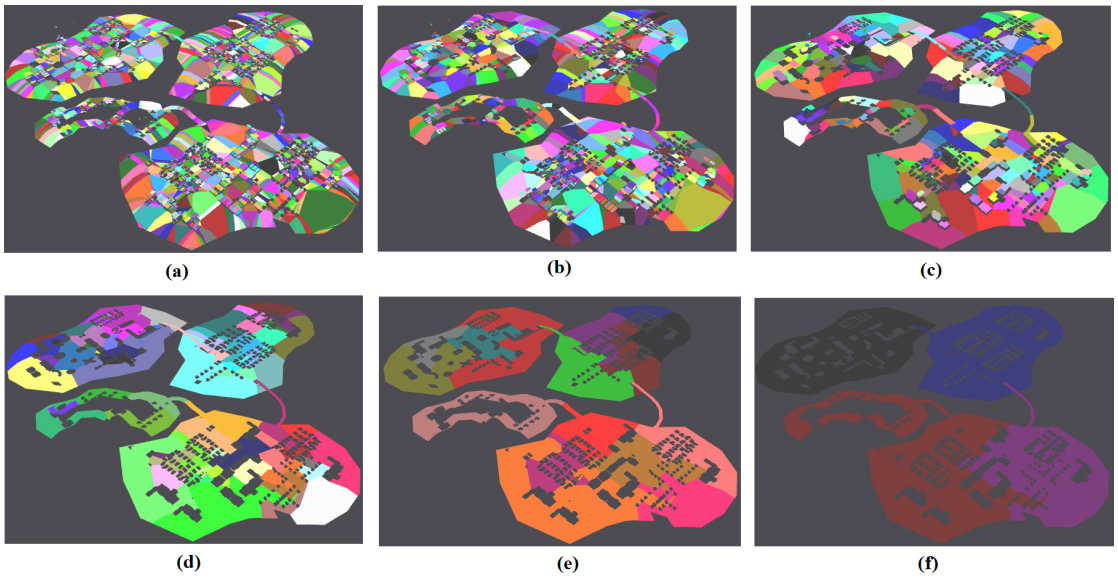


FIGURE 3.12: Hierarchical Graphs: (City Islands, $numMergedNodes = 3, levels = 10$). Level 0 = 5151 nodes (a), Level 1 = 1469 nodes (b), Level 2 = 316 nodes (c), Level 3 = 72 nodes (d), Level 4 = 17 nodes(e), Level 5 = 4 nodes (f). (Model: City Islands (5515 polygons)).

3.3 Path finding Computation

Once the hierarchical subdivision is created in the preprocessing step, the path finding computation can be done at any level of the hierarchy. The second step consists in searching the shortest path between and start node (S) to the goal node (G) in a

specific graph. This online seach brings a performance improvement as any graph in the hierarchy is smaller than the one in level 0. (See Fig. 3.16).

The path finding computation has the following five phases:

3.3.1 Find S and G al certain level

The first phase of this step gets S and G in a certain level in the hierarchy. This level is specified by the user in the Recast Tool. The algorithm receives an initial and end positions in the NavMesh environment. Then, we obtain S and G nodes in the graph at level 0 by searching for their positions. Finally, we recursively look for their parents by passing through all the levels in between until reach the desired level. If S and G nodes are in the same partition, we just run a normal A* between them and the path finding is completed.

3.3.2 Connect S and G to the graph

To be able to search for paths in a graph at certain level, Start and Goal nodes have to be part of the graph. We connect a temporal Start node to each portal in the partition that contains it. Then, we compute a normal A* between S and the center of each portal in the partition. The path nodes and costs are stored for each portal. Finally, we add a new intraedge between the start node the the portal inside the current partition. This step is repeated for G in its respective partition. (See Fig. 3.13).

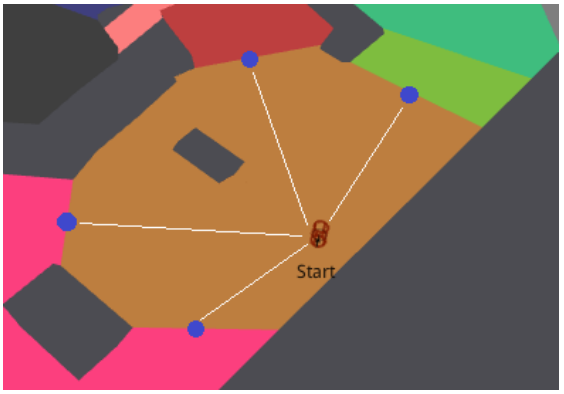


FIGURE 3.13: Connect Start node to the graph: Red circles are portals of the orange partition. White lines are the computed intraedges. Gray polygons are obstacles or no walkable areas.

For each search, S and G should change and the cost of inserting and deleting them is added to the total cost of finding a solution.

3.3.3 Search for a path between S and G at the highest level

Once the S and G are temporally linked to the graph. We perform a normal A* in the current graph. The time execution becomes faster because the number of nodes is significantly smaller than the graph at level 0.

3.3.4 Obtain optimal subpaths

The path planning computation give us all the partitions which are part of the optimal solution. For each of them, we recursively get the path nodes for each lower level until we reach the lowest level in the hierarchy. At the end, we obtain the full path to go from S to G at the level 0.

3.3.5 Delete temporal nodes

The nodes S, G and their edges are eliminated from the current graph.

The preprocessing step is shown in the Fig. 3.14 where a hierarchical subdivision has been applied on a map model. The S and G positions are denoted in white letters. In this sample, we find the shortest path from S to G at level 2. The online step is shown in Fig. 3.15. The start and goal nodes are connected to their respective portals and run a common A* at that level. Finally the sub paths are found until level 0 is reached.

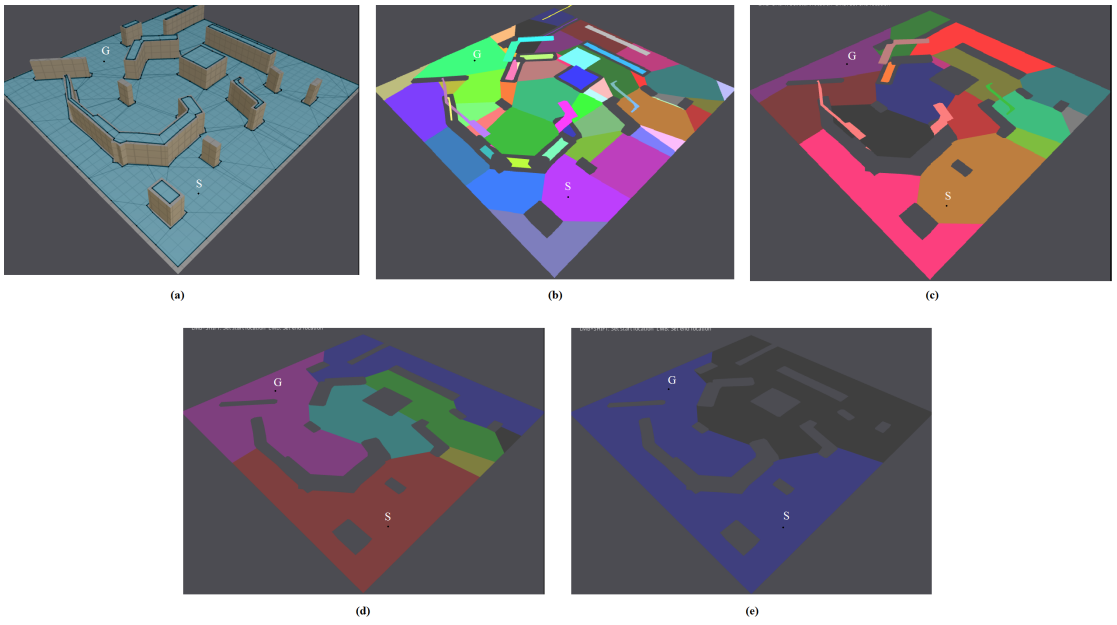


FIGURE 3.14: Hierarchical Subdivision: The graphs in the hierarchy from the lowest level (From (a) to (e)). The start and goal nodes are written in white.

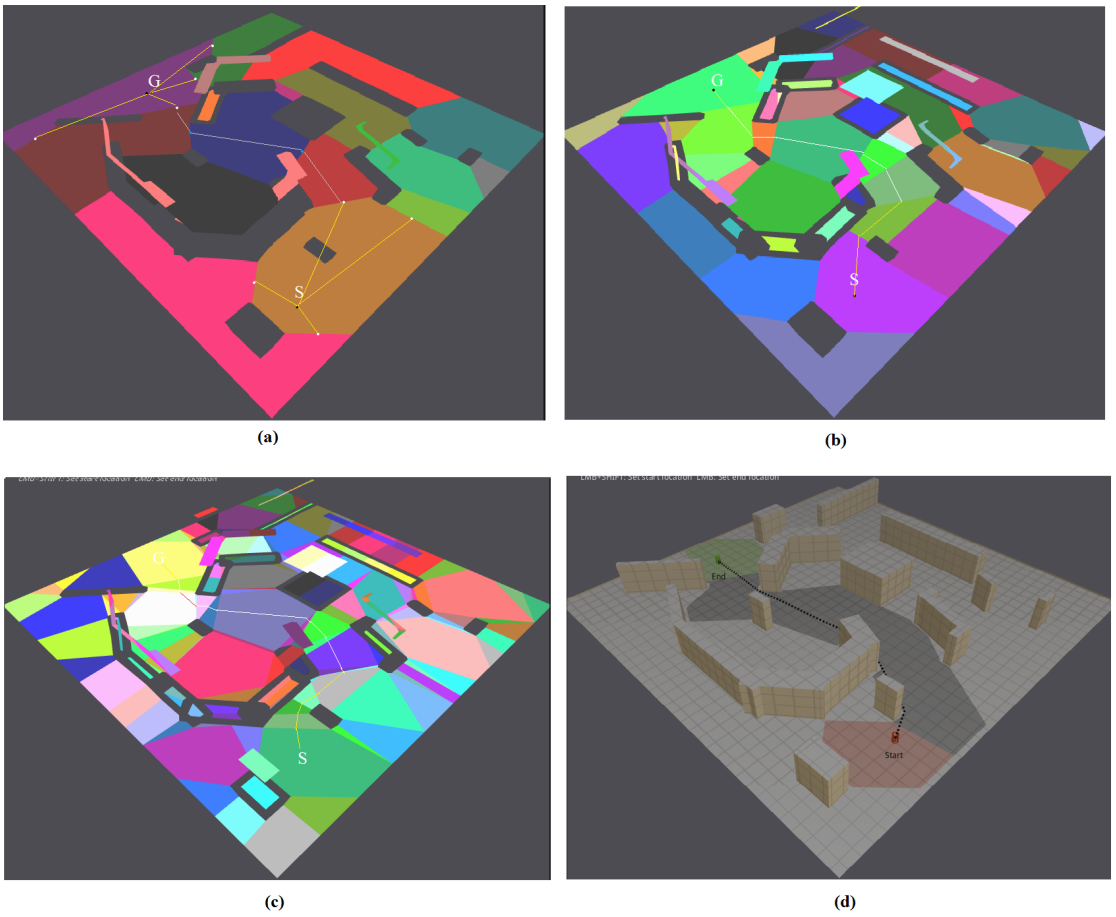


FIGURE 3.15: Path finding Computation: S and G are linked to their partitions at level 2 (a). Sub paths are calculated until level 0 is reached. (Level 1 (b) and Level 0 (c)). The final result of the shortest path between S and G at level 2 (d).

The complete path from the start to the end position in the navigation mesh is achieved in a faster way than computing a normal path finding at the level 0. The comparison and results of our method in different and large challenging models are presented in the next chapter.

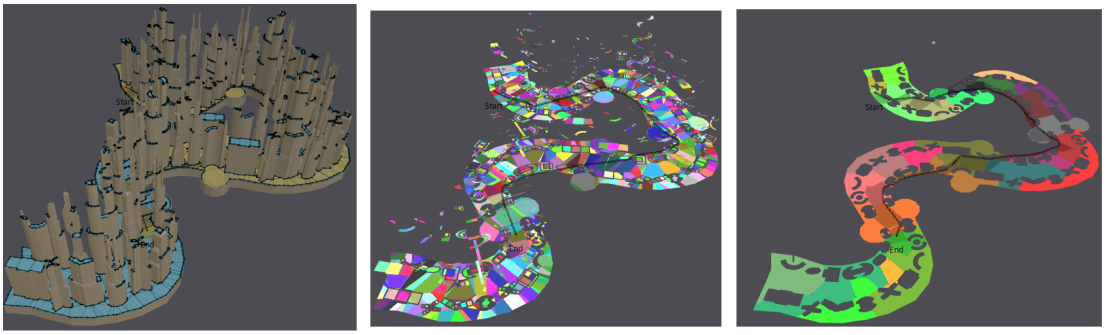


FIGURE 3.16: Path finding Computation: (Serpentine Islands, $numMergedNodes = 4, levels = 10$) Path finding at level 0 (3908 nodes) (a). Path finding at level 0 where each node has a different color (b). Path finding at level 3 (28 nodes) (c).

Chapter 4

Results and Discussion

On this chapter, we show the results obtained in different models with a variety of sizes to measure the improvement achieved with the proposed method. The comparison is based on the speed time for calculating path finding between the start and goal node. Furthermore, we analyse how the time taken for each of the steps we explained in Chapter 3 affects the total time. Also, we examine the impact of varying the number of merged nodes for the different levels on the performance of path finding with our suggested method. Our performance results have been tested on an Intel® Core™ i7 processor with NVIDIA® GeForce® 610M graphics card and 8 GB of RAM. Detailed information of the output that we obtained in our experiments for each model is shown in Appendix A.

4.1 Performance test

Our performance tests are based on: the number of nodes in each level of the hierarchy and the measure of the execution time (milliseconds) of path queries. We performed our approach in different sample models [1] as shown in Table 4.1 and Fig. 4.1.

4.1.1 Number of Nodes

We compared the number of resulting nodes per level in the hierarchy as we increase the number of merged nodes from one level to the next one. As we expected the higher the level, the lower the number of nodes. As an example, the Fig. 4.2 shows the results obtained for the Sirus City. The chart shows number of nodes falls steadily over the

[1] All the test models were obtained for free in http://tf3dm.com/

TABLE 4.1: Sample maps

Map Name	Number of Nodes
Dungeon	120
Map	208
City Colony	2615
Serpentine City	3908
City Islands	5515
Tropical Islands	12666
Sirus City	18738

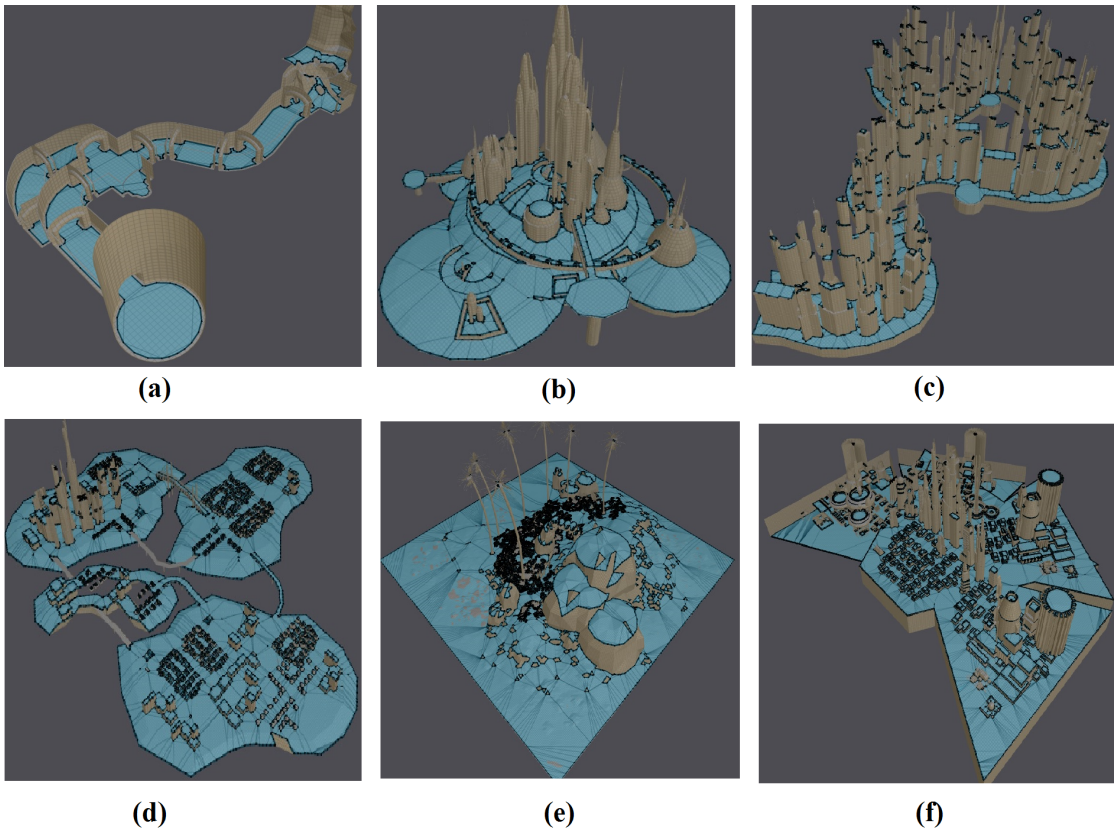


FIGURE 4.1: Sample maps: (Dungeon (a). City Colony (b). Serpentine City (c). City Islands (d). Tropical Islands (e). Sirus City (f).

upper levels in the hierarchy until either one partition cannot be divided any more or the maximum level has been reached.

The division does not only depend on the $numMergedNodes$ parameter. As we explained in Chapter 3, every time that a partition is created, we check whether this partition has connections (edges) with other partitions. If not, then this new nodes are not taken into account for the next level, as they cannot be any further merged. Thus, the consecutive subdivisions do not have an exact segmentation depending exclusively on with the $numMergedNodes$ parameter.

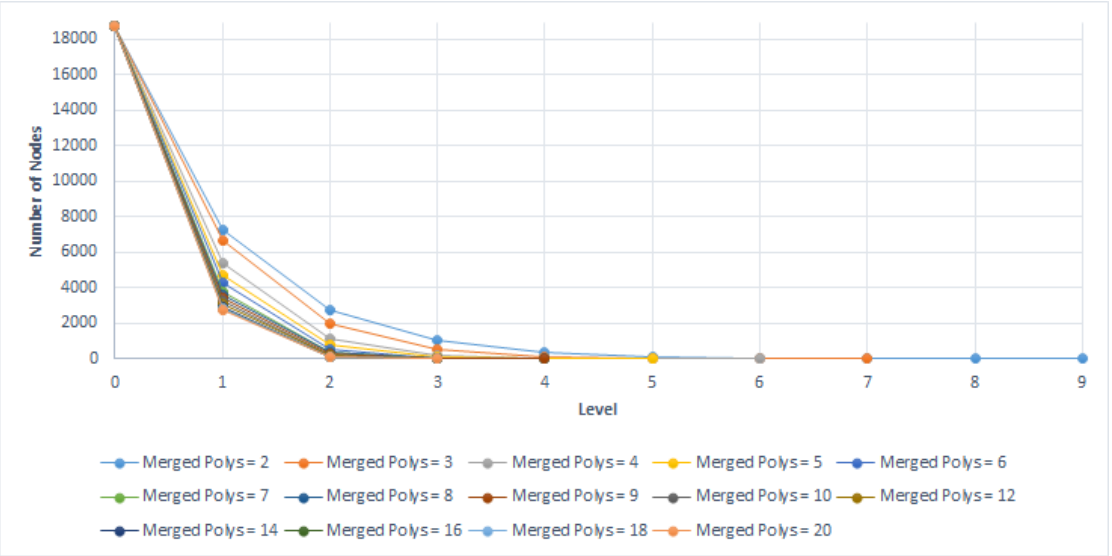


FIGURE 4.2: Level vs Number of Nodes (Sirus City).

4.1.2 Time execution

Firstly, we have analysed the total time taken by the path finding computation by our approach at different levels against the execution time of performing path planning in level 0 (i.e without any hierarchy). We showed three different samples to compare the advantages and disadvantages for each case.

The first scenario is a simple map created with Blender tool [40]. It has an initial graph with 208 nodes. An A* was run over the graph. The total time was $0.347ms$ as we present in the Fig. 4.3. This line chart compares the total execution time per level in the hierarchy. Each line represents the number of merged polys from one level to the next one. It is clear that the execution time for two merged nodes decreases until it reaches the lowest time in level two ($0.14ms$). This time is twice faster than the level 0 time. Merging more than two nodes for this scenario does not offer any further speed up. The gray line ($numMergedNodes = 4$) rise sharply after the level two. This time ($2.122ms$) is six times worse than execution time at level 0.

Note that hierarchical path finding benefits from performing A* in graphs with smaller number of nodes, but we also introduce new steps that can impact performance depending on the caracteristics of the scenario. We will evaluate each of this steps in detail in the following sections.

Fig. 4.4. shows that the path found in the shortest time which was reached by the level two of mergedPolys = 2. At this level, the graph only contains 24 nodes (See Appendix A Table A.2), which allows to have a faster search to the goal.

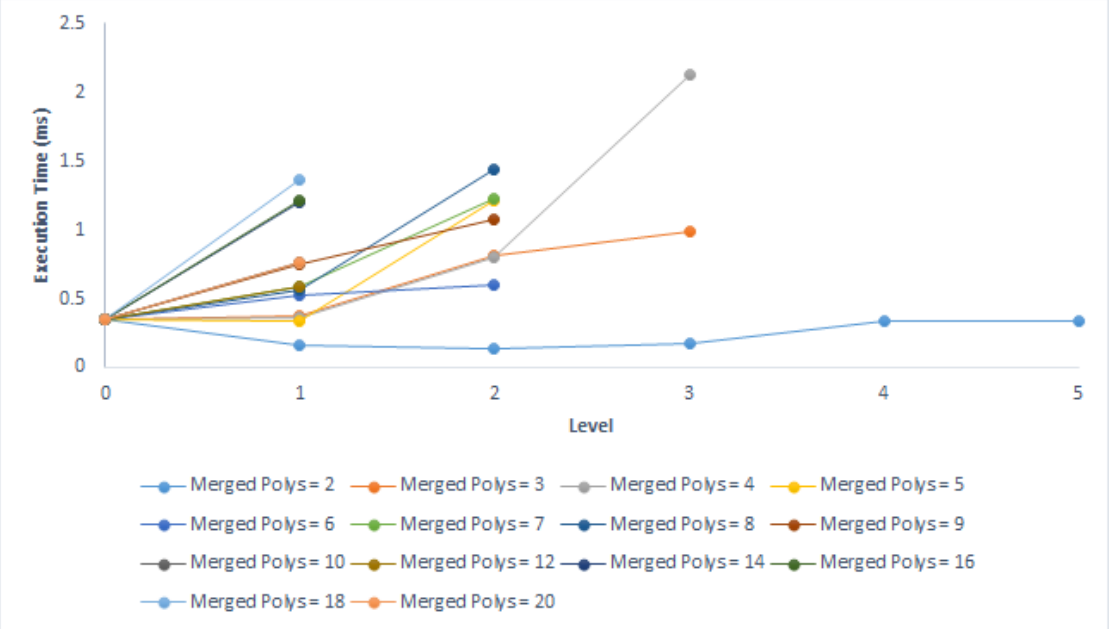


FIGURE 4.3: Level vs Execution Time (Map).

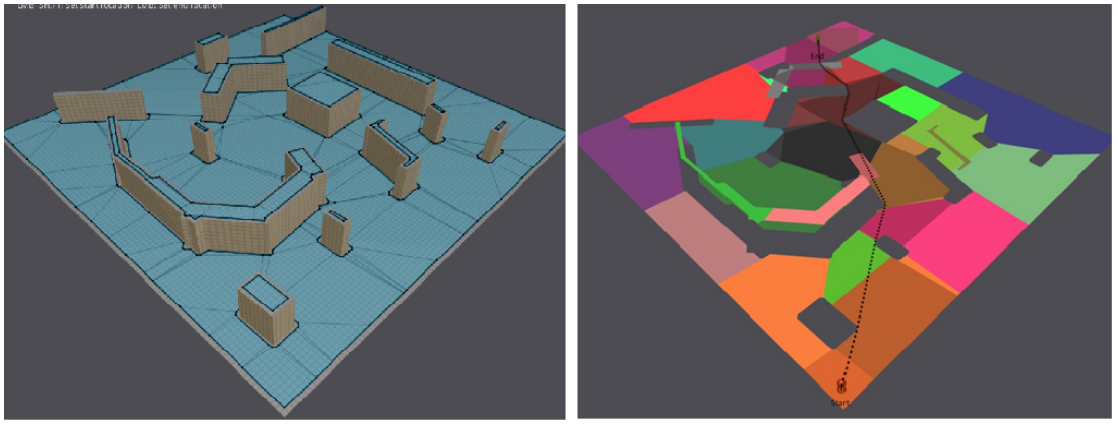


FIGURE 4.4: Map model (a). Shortest path ($numMergedNodes = 2, level = 2$) (b)

The next scenario is a bigger map with 2615 nodes. To begin, the Fig. 4.5 show the computation time over City Colony. It can be seen clearly that the execution time dramatically falls in the first level of the hierarchy. Afterwards, there is a slowly improvement depending on the number of merged nodes 2, 4, and 5 at level two. The other cases have a gentle upward trend but still lower than the computation time at level 0. The highest value reached is $1.43ms$ with $numMergedNodes = 14$ at level two. In contrast, the fastest time is $0.142ms$ (six time faster) for the case of 5 merged nodes in level 2. (See Fig. 4.6).

The last case is the largest scenario, the Tropical Islands sample with 12666 nodes in its

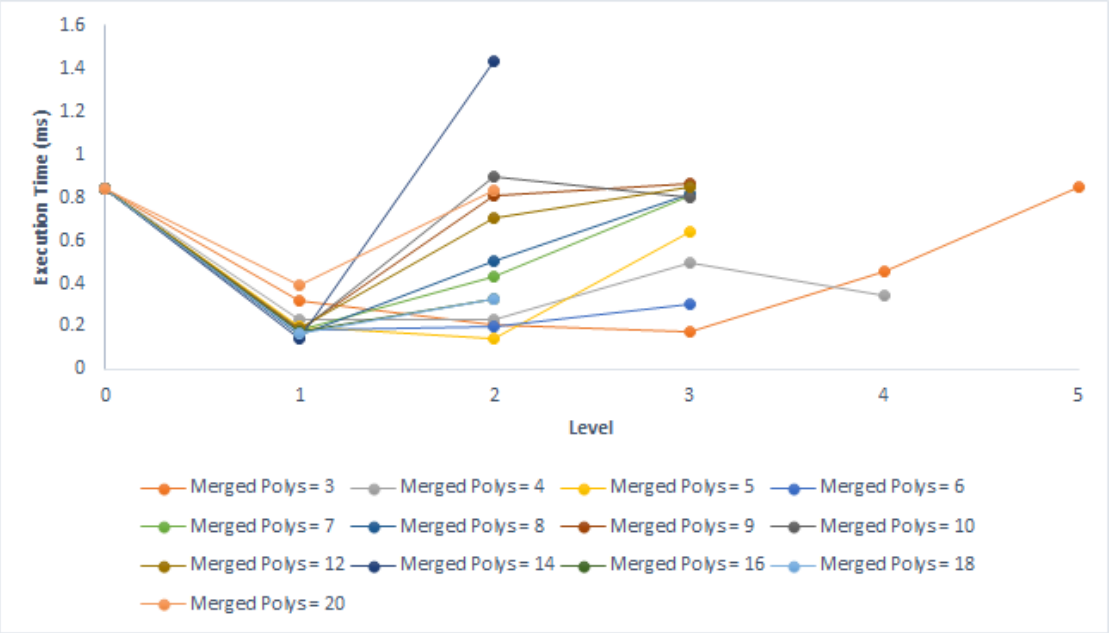
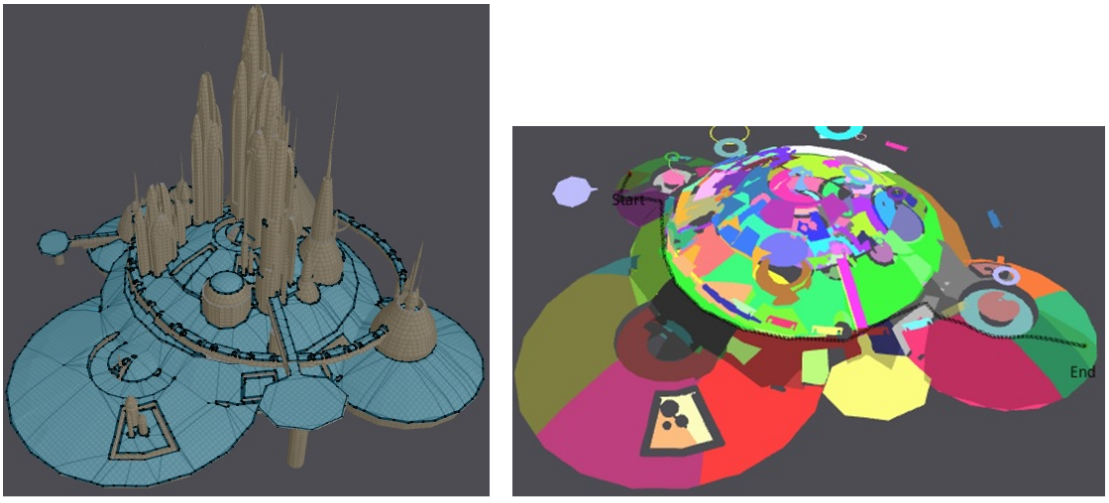


FIGURE 4.5: Level vs Execution Time (City Colony).



(a) **(b)**

FIGURE 4.6: City Colony model (a). Shortest path ($numMergedNodes = 5, level = 2$) (b)

initial graph. It follows the same pattern as the two previous samples showed before. There is a slowly plunge until it reaches the highest level in the hierarchy. On the other hand, The time when $numMergedNodes = 2$ decreased slightly over all levels. (See Figs. 4.7 and 4.8. The fastest time is 1.625 (four times faster) for the case of 16 merged nodes in level 2.

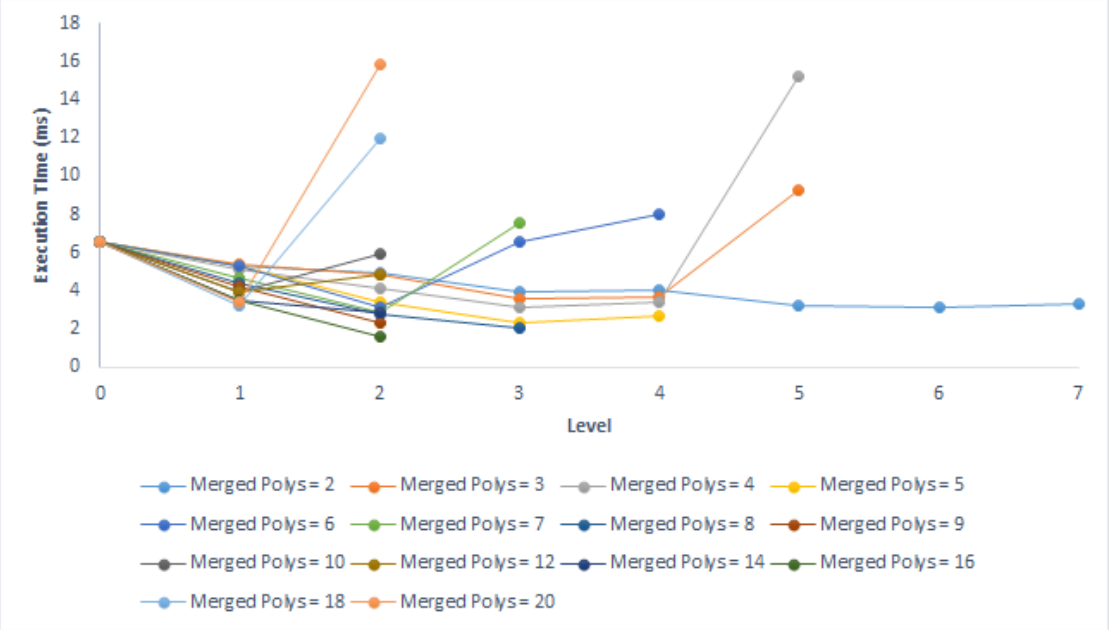
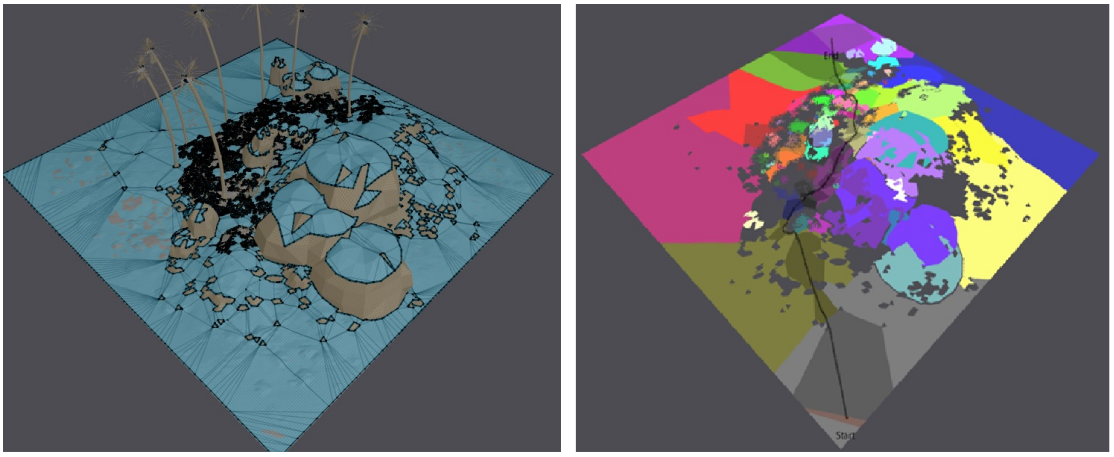


FIGURE 4.7: Level vs Execution Time (Tropical Islands).



(a) **(b)**

FIGURE 4.8: Tropical Islands model (a). Shortest path ($numMergedNodes = 16, level = 2$) (b)

The rest of the models illustrated in the Table 4.1 have the same expected behaviour regarding the number of nodes and computation time at certain level. For more information, we refer the reader to the tables in Appendix A.

To better understand where the bottlenecks of our algorithm appear, we compare the partial times for computing the entire online step (described in Section 3.1.2) for each level in the hierarchy. The online search has been divided in five steps in order to analyse the partial times in this process. See pseudo-code 3.

Algorithm 3 Online Search Pseudo-code

```
procedure ONLINESEARCH(startNodeId, startNodePos, endNodeId, endNodePos, level)
Find S and G al certain level:
    sNode ← getNode(startNodeId)
    gNode ← getNode(endNodeId)
    if level == 0 then
        path ← findPath(startNodeId, startNodePos, endNodeId, endNodePos, 0)
        return path
Connect S and G to the graph:
    linkStartToGraph(sNode)
    linkGoalToGraph(gNode)
Search for a path between S and G at the highest level:
    tempPath ← findPath(sNode.id, startNodePos, gNode.id, endNodePos, level)
Obtain optimal subpaths:
    for all subpath ∈ temPath do
        path ← getSubpath(subpath, level − 1)
Delete S and G:
    deleteNode(sNode)
    deleteNode(eNode)
    return path
```

In particular, we choose the Sirus City model to be evaluated. Each partial time is plotted in order to discuss the possible disadvantages in the next section. Detailed information of the partial times we have obtained for Sirus city model is shown in table 7 in Appendix A.

4.1.2.1 Find S and G al certain level

The Fig. 4.9 illustrates the time of getting the partitions to which Start and Goal nodes respectively belong to a certain level. Those partitions are getting by recursively search S and G positions in the upper levels of the hierarchy until they reach a predetermined level. Overall, the time has a gradual rise in the entire hierarchy. This was expected as the higher the level we want to reach, the more time is consumed. Notice that the total time for this step is not significant in the total time (values less than $0.005ms$).

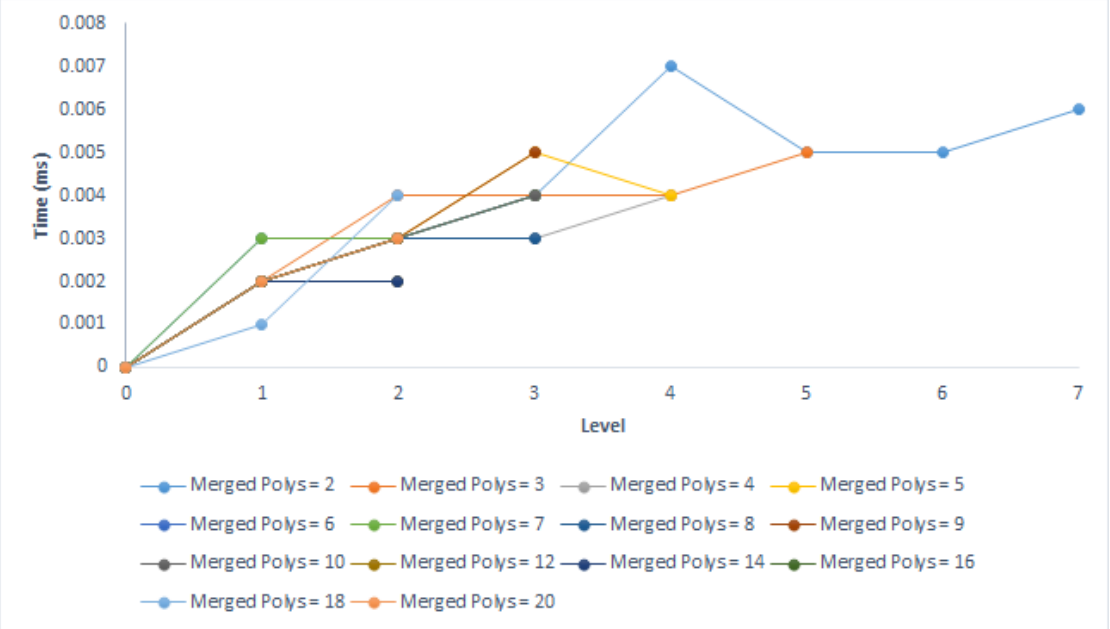


FIGURE 4.9: Get S and G Time vs Level

4.1.2.2 Connect S and G to the graph

The chart shows the time of connecting S and G to each of the portals in their respective partitions. S and G are linked by performing an A* from those nodes to each portal in the current partition. Then, the path nodes and cost are stored by adding an intraedge.

The figure shows an upward trend throughout the levels of the hierarchy. The figure exposes a particular strong growth in the highest levels due to the number of portals being bigger in the upper levels. (See Fig. 4.10). This particular behaviour is due to the fact that we have to execute as many A* searches as the number of portals the partition has. The table below shows the number of portals in S and G for the two worst cases in Sirus City model.

TABLE 4.2: Number of portals (Sirus city)

Merged Polys	Level	Number Portals in S	Number Portals G	Connect S and G time
2	1	3	6	0.077
2	2	3	5	0.067
2	3	6	4	0.189
2	4	13	8	0.986
2	5	10	7	1.436
2	6	16	8	7.124
2	7	21	36	17.546
10	1	3	5	0.058
10	2	5	3	0.195
10	3	18	10	14.381

The worst times are highlighted in red. For the case when $mergedPolys = 2$ and $level = 7$, only the partial time is four time slower than performing an A* al level 0 ($17.546ms$). This time spends almost the 92.5% of the total time. In the rest of the cases, the partial time is not relevant. Please see table 7 in Appendix A for further information.

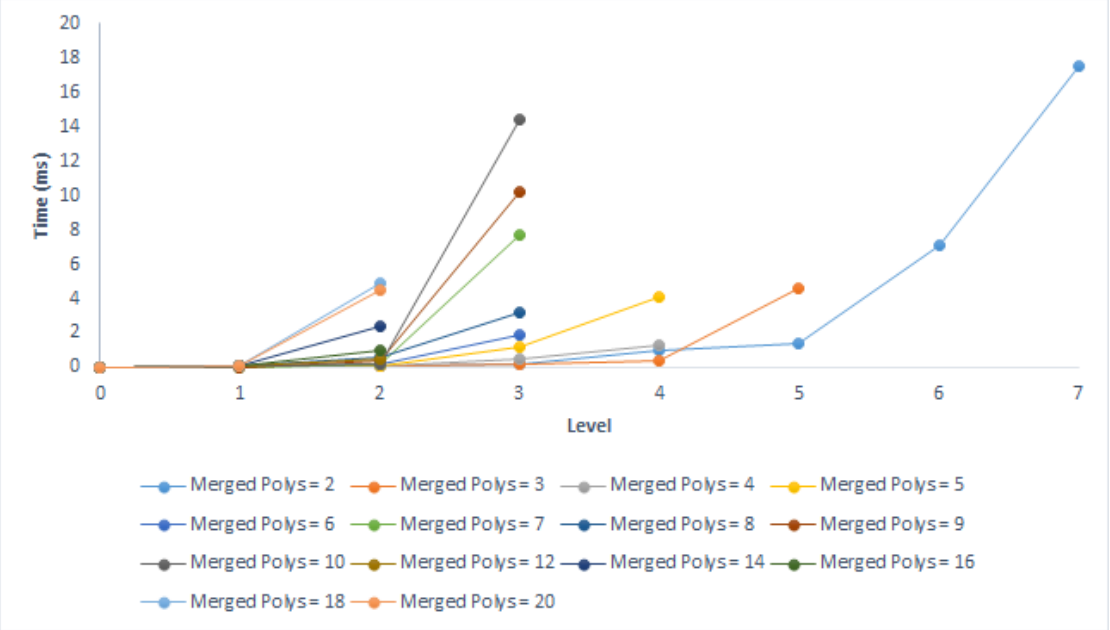


FIGURE 4.10: Link S and G Time vs Level

4.1.2.3 Search for a path between S and G at the highest level

Regarding the path finding calculation, an A* computation is faster when the searching is done in a higher level in the hierarchy. A regular A* is performed between the start and goal nodes at certain level. The Fig. 4.11 shows the general gradual decline for all the cases. The lower the number of nodes, the faster the exploration is. The number of nodes in a specific level highly depends on how many nodes were merged in its lower level. For instance, for the case when $mergedPolys = 2$ and $level = 10$, the number of nodes was 22. The partial time was $1.333ms$. When $mergedPolys = 10$ and $level = 3$, the number of nodes was 11 with a partial time $0.287ms$.

4.1.2.4 Obtain optimal subpaths

The chart 4.13 shows the time of getting the subpaths for each level. The subpaths are obtained by recursively get the stored paths in each nodes of the lower levels until we

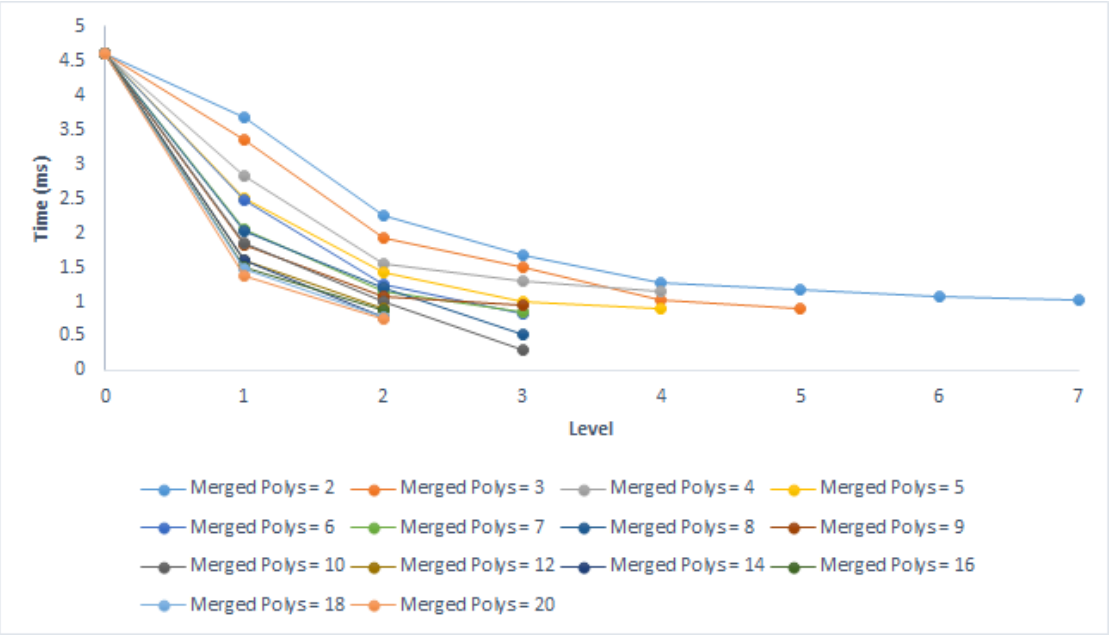


FIGURE 4.11: A* Time vs Level

reach the level 0. Those subpaths has the nodes which become the optimal path at level 0.

As an overall trend, the time of getting subpaths increased fairly slowly until the penultimate level of the hierarchy. Then, this time gradual decline in the highest level. This is due to there are no nodes between S and G. Also, there are not intermediate paths between them. Fig. 4.12 illustrates this scenario.

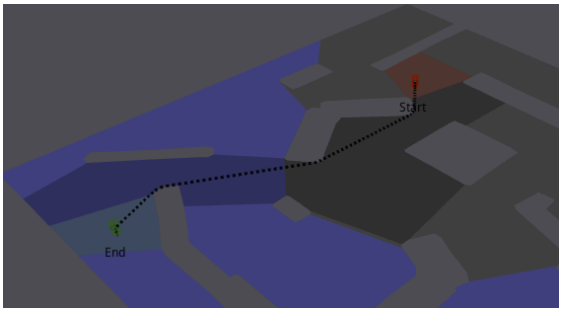


FIGURE 4.12: Obtain optimal subpaths. The map model has two nodes at level 3. No intermediate paths are calculated

Therefore, the execution time is really low and strictly depends on the map environment and the number of merged nodes.

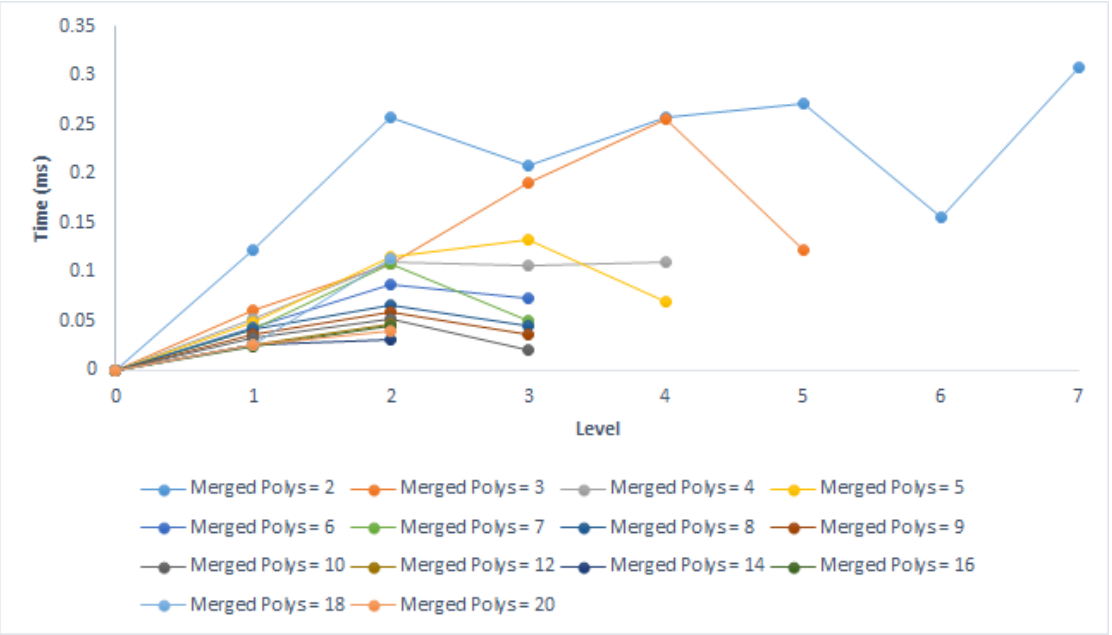


FIGURE 4.13: Time vs Level

4.1.2.5 Delete S and G

Deleting temporal S and G nodes have an insignificant time compare to other partial times during the online process. Those times are below $0.001ms$ as we can see in the Appendix A.

4.2 Discussion

Our performance tests have been applied over different models size. Due to the number of nodes in each level of the hierarchy, our technique allow us to reduce the number of nodes successively for higher levels of the hierarchy thus reducing path finding calculations.

Regarding execution time, the results shown above give us a clear view about the special cases in our approach. Firstly, the total time of connecting S and G becomes extremely high as the number of portals in the partition increases. This represents a significant increase in the total time. The cost of linking the Start and Goal node becomes critical when the partition has a lot of edges. The number of edges per partition grows as we increase the level in the hierarchy. Also, the number of portals in a specific partition depends on the quality of the partition. Some examples of very large times when connecting nodes can been seen in the table A5 (Merged Polys = 3, level = 5; Merged Polys = 5, level = 4) and table A7 (Merged Polys = 2, level = 7; Merged Polys = 10, level = 3) of Appendix A.

The time of getting the sub paths had relatively low values. This happens because there are no intermediate paths between the Start and End nodes. Those sub paths are already stored in each partition respectively. Therefore, it is not necessary to get recursively intermediate paths.

The Fig. 4.14 clearly illustrates these two cases. In the Sirus model, for the case of $numMergedNodes = 10$ and $level = 3$ the time of connecting S and G is $14.38ms$ (three times slower than a basic A* in level 0). We can see in the figure how the number of portals is 18 for the start partition and 10 for the end one. This division is not a good one for this path. Also, the time of getting the subpaths ($0.021ms$) is really small because there are not intermediate paths between start and end partitions.

FIGURE 4.14: Special case: Start partition has 18 portals (red points) and end partition has 10 (blue points)

As we have seen in our results, the overall execution times depend strongly on the partition and the path we are looking for. Even thought there are particular cases for which the overall time can be slower than just A* in level 0, our approach gives better results in most cases for large and complex 3D world representations.

Our final conclusion of this work, is that having hierarchical path finding over navigation meshes can drastically reduce path finding calculations. But it is necessary for each particular scenario to run a few test cases (different values when merging cells between levels) to determine which values can offer the speed up for your particular scenario.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

With the booming growth of video games, there is a great demand on path finding algorithms. In this master thesis, we presented a hierarchical path finding framework to speed up crowd simulation for large 3D environments. The approach has two steps: Preprocessing and online search. Preprocessing step builds the hierarchy of levels whereas that online process deals with the path finding search. Our method has a tree hierarchy of graphs where the searching can be performed at any level.

The main contributions of our approach are:

- A path planning algorithm for arbitrary graph types that could be applied in any kind of 3D world representation.
- A recursive partition of a graph based on reducing the connector edges.
- A hierarchy of graphs to find the fastest time execution for path planning.

Our evaluation has shown a significant improvement in path finding time execution. The method has better results when the path planning is performed in big world representations(5 or 6 times faster). For small models, a common A* is enough. The trade-off between the chosen level and the size of partitions is important. Also, our approach shows better performance in non widespread environments such as Serpentine city where our solution is six times faster than a normal A*.

The framework presented in this dissertation inspired by HPA* approach but we also provide multi-level search and present a new algorithm that works over any kind of environment division.

5.2 Future Work

Despite our improvements, there is still a large amount of work that could be done to obtain either fast path finding searches or good path quality. Some of the enhancements that could be incorporated with our current framework. For instance, the improvement of linking Start and Goal node to the current graph. This is an important issue to address in the future. A way of reducing link-time would be to replace A* with a version of Dijkstra's algorithm that does not flush the pool of visited vertices between searches. Also, those nodes could be somehow stored in order to reduce the time execution of connecting and deleting.

It would also be interesting to study if some steps of the online search could be parallelized using a GPU implementation. For example, the process of linking S and G could be performed in separated threads for each portal as well as getting the subtpath for each partition. Also, our approach could be extended to work under dynamic environments where the replanning could be done only at certain level of the hierarchy.

Appendix A

Performance Test

In this appendix, we provide all the results for the different sample maps. The output show the execution time for each level at different number of merged nodes. The fastest time is highlighted with blue color. Each column is described as follows:

- Merged Polys. Number of polygons which have been merged to create a new node.
- Level. Number of level in the hierarchy. The original graph is at the level 0.
- Number Nodes. Number of nodes in that level.
- Get S and G. Time in milliseconds to get Start and Goal Node at that level.
- Connect S and G. Time in milliseconds to link the Start and End position with the portals of the partition respectively.
- A*. Time in milliseconds to get the optimal path at that level.
- Subpaths. Time in milliseconds to recursively get the nodes at the level 0.
- Total Time. Sum of the previous times.

TABLE A.3: City Colony (2615 Nodes)

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	2615	0	0	0.839	0	0	0.839
3	1	868	0.002	0.028	0.261	0.025	0	0.316
3	2	288	0.003	0.048	0.118	0.042	0	0.211
3	3	71	0.003	0.088	0.037	0.047	0.001	0.176
3	4	14	0.004	0.408	0.02	0.022	0	0.454
3	5	4	0	0	0.849	0	0	0.849
4	1	711	0.001	0.016	0.192	0.021	0.001	0.231
4	2	173	0.004	0.114	0.08	0.036	0	0.234
4	3	27	0.003	0.416	0.042	0.038	0	0.499
4	4	5	0.004	0.332	0.005	0.001	0	0.342
5	1	602	0.001	0.028	0.154	0.017	0	0.2
5	2	125	0.004	0.044	0.05	0.044	0	0.142
5	3	11	0.002	0.626	0.012	0.001	0	0.641
6	1	530	0.001	0.035	0.135	0.014	0	0.185
6	2	101	0.003	0.126	0.045	0.028	0	0.202
6	3	7	0.002	0.295	0.005	0.001	0.001	0.
7	1	467	0.002	0.048	0.119	0.015	0.001	0.185
7	2	79	0.003	0.363	0.042	0.025	0	0.433
7	3	5	0	0	0.806	0	0	0.806
8	1	433	0.001	0.039	0.105	0.013	0	0.158

Continuation of Table A.3

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
8	2	68	0.002	0.439	0.043	0.022	0	0.506
8	3	5	0	0	0.819	0	0	0.819
9	1	398	0.001	0.059	0.109	0.011	0.001	0.181
9	2	53	0.002	0.741	0.037	0.026	0	0.806
9	3	4	0	0	0.862	0	0	0.862
10	1	368	0.002	0.052	0.108	0.009	0.001	0.172
10	2	44	0.003	0.876	0.016	0.001	0.001	0.897
10	3	4	0	0	0.8	0	0	0.8
12	1	338	0.001	0.096	0.088	0.009	0	0.194
12	2	29	0.002	0.675	0.014	0.009	0.001	0.701
12	3	3	0	0	0.846	0	0	0.846
14	1	325	0.001	0.049	0.084	0.009	0.001	0.144
14	2	25	0.003	1.408	0.018	0.001	0	1.43
16	1	304	0.002	0.082	0.085	0.009	0.001	0.179
16	2	19	0.003	0.32	0.005	0.001	0.001	0.33
18	1	296	0.001	0.088	0.072	0.009	0	0.17
18	2	17	0.002	0.323	0.005	0.001	0	0.331
20	1	289	0.003	0.314	0.066	0.008	0	0.391
20	2	16	0	0	0.835	0	0	0.835

End of Table

Table A.4: Serpentine City (3908 Nodes)

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	3908	0	0	4.692	0	0	4.692
2	1	1357	0.002	0.029	1.734	0.128	0	1.893
2	2	466	0.003	0.069	1.085	0.222	0	1.379
2	3	127	0.003	0.287	0.852	0.314	0	1.456
2	4	49	0.006	0.689	0.747	0.399	0	1.841
2	5	15	0.004	1.425	0.475	0.397	0	2.301
2	6	4	0.005	5.087	0.199	0.529	0	5.82
2	7	1	0.007	0	4.41	0	0	4.417
3	1	1200	0.003	0.024	1.595	0.091	0	1.713
3	2	361	0.002	0.142	1.028	0.176	0.001	1.349
3	3	76	0.003	0.355	0.784	0.265	0.001	1.408
3	4	18	0.004	0.759	0.477	0.271	0	1.511
3	5	5	0.004	4.105	0.151	0.173	0.001	4.434
4	1	1001	0.002	0.033	1.396	0.089	0	1.52
4	2	229	0.003	0.045	0.896	0.17	0	1.114
4	3	28	0.003	0.325	0.657	0.264	0	1.249
4	4	7	0.003	1.775	0.148	0.205	0	2.131
5	1	867	0.002	0.05	1.263	0.085	0.001	1.401
5	2	145	0.003	0.162	0.854	0.223	0	1.242
5	3	17	0.003	0.499	0.64	0.318	0	1.46
5	4	3	0.003	4.951	0.041	0.146	0	5.141

Continuation of Table A.7

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
6	1	787	0.002	0.043	1.164	0.073	0	1.282
6	2	99	0.003	0.228	0.718	0.138	0	1.087
6	3	11	0.003	1.63	0.594	0.181	0	2.408
7	1	713	0.002	0.073	1.078	0.075	0	1.228
7	2	62	0.002	0.467	0.687	0.133	0	1.289
7	3	7	0.003	1.734	0.431	0.123	0	2.291
8	1	672	0.002	0.055	1.058	0.057	0	1.172
8	2	40	0.003	0.417	0.572	0.139	0.001	1.132
8	3	4	0.002	1.723	0.041	0.07	0	1.836
9	1	650	0.002	0.117	0.973	0.06	0	1.152
9	2	38	0.003	0.917	0.557	0.106	0.001	1.584
9	3	4	0.003	1.437	0.087	0.093	0	1.62
10	1	617	0.001	0.056	0.935	0.073	0	1.065
10	2	24	0.003	0.631	0.444	0.091	0.001	1.17
10	3	2	0.002	6.447	0.011	0.001	0	6.461
12	1	582	0.002	0.072	0.857	0.05	0	0.981
12	2	16	0.003	1.252	0.404	0.083	0.001	1.743
14	1	562	0.002	0.112	0.834	0.047	0.001	0.996
14	2	11	0.003	1.326	0.572	0.097	0	1.998
16	1	542	0.002	0.123	0.782	0.047	0	0.954
16	2	8	0.002	3.629	0.201	0.062	0	3.894

Continuation of Table A.7

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
18	1	538	0.002	0.083	0.777	0.041	0.001	0.904
18	2	8	0.003	1.822	0.095	0.046	0.001	1.967
20	1	524	0.001	0.129	0.749	0.046	0	0.925
20	2	6	0.003	0.987	0.087	0.064	0	1.141

End of Table

TABLE A.5: City Islands (5515 Nodes)

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	5515	0.001	0	6.325	0	0	6.326
2	1	2025	0.003	0.278	4.062	0.118	0.001	4.462
2	2	781	0.003	0.312	2.931	0.178	0	3.424
2	3	342	0.006	0.55	2.64	0.265	0	3.461
2	4	141	0.003	0.498	2.546	0.345	0.001	3.393
2	5	49	0.005	0.871	2.362	0.432	0.001	3.671
2	6	12	0.005	4.276	1.03	0.546	0	5.857
3	1	1783	0.002	0.165	3.805	0.055	0	4.027
3	2	519	0.003	0.127	2.676	0.102	0	2.908
3	3	151	0.004	0.696	2.574	0.155	0	3.429
3	4	35	0.005	1.971	2.21	0.202	0	4.388
3	5	11	0.005	17.413	2.284	0.264	0.001	19.967

Continuation of Table A.5

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
4	1	1469	0.002	0.159	3.304	0.046	0	3.511
4	2	316	0.003	0.1	2.335	0.089	0	2.527
4	3	72	0.004	0.218	2.691	0.134	0	3.047
4	4	17	0.004	2.301	2.313	0.207	0	4.825
4	5	4	0.007	6.301	0.308	0.106	0.001	6.723
5	1	1272	0.002	0.075	3.085	0.041	0.001	3.204
5	2	224	0.003	0.291	2.436	0.098	0	2.828
5	3	46	0.006	3.32	2.053	0.163	0	5.542
5	4	10	0.005	13.684	1.889	0.326	0	15.904
6	1	1126	0.002	0.043	2.829	0.065	0	2.939
6	2	144	0.002	0.413	2.44	0.091	0	2.946
6	3	24	0.003	1.517	2.553	0.136	0.001	4.21
7	1	1017	0.003	0.112	2.755	0.043	0	2.913
7	2	111	0.002	0.41	2.443	0.08	0.001	2.936
7	3	15	0.004	6.56	1.903	0.071	0.001	8.539
8	1	945	0.002	0.074	2.619	0.038	0.001	2.734
8	2	82	0.003	1.238	2.198	0.069	0	3.508
8	3	9	0.003	4.06	0.863	0.071	0	4.997
9	1	891	0.001	0.216	2.585	0.04	0	2.842
9	2	65	0.003	0.654	2.687	0.08	0.001	3.425
10	1	842	0.002	0.257	2.534	0.033	0	2.826

Continuation of Table A.5

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
10	2	53	0.003	3.054	2.152	0.084	0	5.293
12	1	765	0.002	0.064	2.402	0.04	0	2.508
12	2	34	0.004	2.686	2.248	0.203	0	5.141
14	1	728	0.002	0.181	2.4	0.031	0	2.614
14	2	28	0.003	2.463	1.64	0.07	0	4.176
16	1	690	0.002	0.176	2.41	0.039	0	2.627
16	2	22	0.003	3.868	1.835	0.089	0	5.795
18	1	672	0.001	0.35	2.311	0.037	0.001	2.7
18	2	20	0.002	6.966	1.981	0.057	0.001	9.007
20	1	645	0.002	0.212	2.304	0.032	0	2.55
20	2	13	0.004	3.397	0.818	0.177	0	4.396

End of Table

TABLE A.6: Tropical Islands (12666 Nodes)

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	12666	0	0	6.536	0	0	6.536
2	1	4868	0.003	0.033	5.147	0.108	0	5.291
2	2	2074	0.005	0.037	4.697	0.233	0.001	4.973
2	3	832	0.004	0.184	3.465	0.286	0.001	3.94
2	4	314	0.005	1.009	2.472	0.599	0.001	4.086
2	5	132	0.004	0.899	1.651	0.708	0.001	3.263

Continuation of Table A.6

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
2	6	53	0.005	1.572	0.989	0.551	0	3.117
2	7	24	0.005	1.886	0.658	0.77	0	3.319
3	1	4257	0.004	0.039	5.083	0.27	0	5.396
3	2	1550	0.004	0.05	4.44	0.375	0.001	4.87
3	3	464	0.005	0.192	3.103	0.26	0	3.56
3	4	154	0.005	0.771	2.167	0.777	0	3.72
3	5	48	0.005	7.356	1.298	0.642	0	9.301
4	1	3382	0.003	0.061	4.962	0.088	0	5.114
4	2	1014	0.005	0.074	3.607	0.413	0	4.099
4	3	182	0.004	0.448	2.3	0.411	0	3.163
4	4	44	0.004	1.76	1.227	0.411	0	3.402
4	5	9	0.004	14.577	0.362	0.291	0	15.234
5	1	2841	0.002	0.037	5.19	0.084	0	5.313
5	2	769	0.002	0.151	3.008	0.223	0	3.384
5	3	103	0.003	0.49	1.762	0.083	0.001	2.339
5	4	20	0.004	1.948	0.664	0.094	0.001	2.711
6	1	2469	0.003	0.036	5.118	0.12	0	5.277
6	2	573	0.003	0.261	2.554	0.341	0	3.159
6	3	64	0.005	4.976	1.206	0.401	0	6.588
6	4	8	0.005	7.219	0.518	0.307	0	8.049
7	1	2121	0.002	0.029	4.605	0.07	0	4.706

Continuation of Table A.6

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
7	2	421	0.003	0.449	2.284	0.12	0.001	2.857
7	3	38	0.004	6.577	0.866	0.148	0.001	7.596
8	1	1876	0.003	0.035	4.343	0.07	0	4.451
8	2	314	0.003	0.245	2.386	0.137	0	2.771
8	3	30	0.003	1.342	0.537	0.159	0	2.041
9	1	1718	0.002	0.041	4.165	0.023	0	4.231
9	2	227	0.003	0.227	1.973	0.139	0.001	2.343
10	1	1524	0.002	0.036	3.918	0.022	0	3.978
10	2	160	0.003	3.82	1.971	0.14	0	5.934
12	1	1388	0.002	0.054	3.826	0.067	0.001	3.95
12	2	126	0.002	3.289	1.39	0.142	0.001	4.824
14	1	1263	0.002	0.056	3.421	0.068	0	3.547
14	2	107	0.002	1.614	1.16	0.125	0	2.901
16	1	1188	0.002	0.05	3.422	0.069	0.001	3.544
16	2	84	0.003	0.524	0.996	0.102	0	1.625
18	1	1107	0.002	0.071	3.133	0.052	0.001	3.259
18	2	69	0.003	10.36	1.473	0.119	0	11.955
20	1	1058	0.002	0.081	3.231	0.063	0	3.377
20	2	58	0.002	14.791	0.997	0.053	0	15.843

End of Table

Table A.7: Sirus City (18738 Nodes)

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	18738			4.606	0	0	4.606
2	1	7274	0.003	0.077	3.685	0.122	0	3.887
2	2	2746	0.003	0.067	2.259	0.258	0.001	2.588
2	3	1021	0.004	0.189	1.686	0.209	0	2.088
2	4	367	0.007	0.986	1.285	0.258	0.001	2.537
2	5	142	0.005	1.436	1.183	0.272	0	2.896
2	6	56	0.005	7.124	1.196	0.155	0	8.48
2	7	22	0.006	17.546	1.333	0.308	0	19.193
3	1	6638	0.002	0.07	3.351	0.061	0	3.484
3	2	1989	0.004	0.142	1.925	0.109	0.001	2.181
3	3	533	0.004	0.201	1.488	0.19	0.001	1.884
3	4	148	0.004	0.357	1.033	0.255	0	1.649
3	5	49	0.005	4.577	0.91	0.123	0	5.615
4	1	5424	0.002	0.043	2.837	0.052	0	2.934
4	2	1115	0.003	0.113	1.557	0.11	0	1.783
4	3	188	0.003	0.474	1.305	0.107	0	1.889
4	4	36	0.004	1.288	1.156	0.11	0.001	2.559
5	1	4714	0.002	0.056	2.513	0.049	0.001	2.621
5	2	778	0.003	0.091	1.425	0.115	0	1.634
5	3	106	0.005	1.167	1.004	0.133	0	2.309
5	4	15	0.004	4.05	0.904	0.07	0.001	5.029

Continuation of Table A.7

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
6	1	4277	0.002	0.045	2.467	0.043	0.001	2.558
6	2	580	0.003	0.186	1.243	0.088	0	1.52
6	3	67	0.004	1.916	0.836	0.074	0	2.83
7	1	3816	0.003	0.029	2.04	0.042	0	2.114
7	2	418	0.003	0.188	1.152	0.109	0	1.452
7	3	38	0.004	7.735	0.858	0.05	0	8.647
8	1	3625	0.002	0.056	2.024	0.042	0	2.124
8	2	346	0.003	0.627	1.206	0.067	0	1.903
8	3	24	0.003	3.245	0.531	0.045	0	3.824
9	1	3446	0.002	0.035	1.824	0.036	0.001	1.898
9	2	288	0.003	0.364	1.083	0.06	0	1.51
9	3	18	0.005	10.243	0.958	0.036	0	11.242
10	1	3302	0.002	0.058	1.851	0.034	0	1.945
10	2	243	0.003	0.195	0.995	0.052	0	1.245
10	3	11	0.004	14.381	0.287	0.021	0.001	14.694
12	1	3101	0.002	0.081	1.612	0.026	0	1.721
12	2	186	0.002	0.521	0.899	0.048	0	1.47
14	1	2962	0.002	0.061	1.596	0.027	0	1.686
14	2	137	0.002	2.398	0.779	0.032	0	3.211
16	1	2871	0.002	0.062	1.494	0.024	0	1.582
16	2	117	0.003	1.042	0.863	0.046	0.001	1.955

Continuation of Table A.7

Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
18	1	2802	0.001	0.057	1.483	0.026	0.001	1.568
18	2	98	0.004	4.883	0.776	0.113	0	5.776
20	1	2753	0.002	0.066	1.363	0.027	0	1.458
20	2	86	0.003	4.468	0.741	0.04	0	5.252

End of Table

TABLE A.1: Dungeon (120 Nodes)

Parameters			Time Execution (ms)					
Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	120	0	0	0.226	0	0	0.226
2	1	56	0.002	0.046	0.088	0.036	0	0.172
2	2	26	0.003	0.027	0.046	0.043	0.001	0.12
2	3	10	0.002	0.022	0.022	0.049	0.001	0.096
2	4	5	0.003	0.122	0.011	0.027	0	0.163
2	5	2	0.004	0.122	0.005	0.001	0	0.132
3	1	49	0.001	0.017	0.088	0.014	0.001	0.121
3	2	15	0.003	0.013	0.028	0.028	0.001	0.073
3	3	5	0.004	0.172	0.015	0.025	0.001	0.217
4	1	36	0.002	0.025	0.054	0.01	0	0.091
4	2	9	0.003	0.039	0.016	0.02	0	0.078
5	1	28	0.001	0.009	0.051	0.009	0	0.07
5	2	5	0.003	0.103	0.011	0.012	0.001	0.13
6	1	21	0.001	0.013	0.033	0.007	0	0.054
6	2	3	0.003	0.093	0.007	0.005	0	0.108
7	1	20	0.001	0.021	0.031	0.007	0	0.06
7	2	2	0.002	0.125	0.005	0.001	0	0.133
8	1	18	0.001	0.024	0.026	0.005	0	0.056
8	2	2	0.002	0.127	0.005	0.001	0	0.135
9	1	17	0.002	0.026	0.027	0.007	0.001	0.063
10	1	15	0.002	0.034	0.023	0.006	0	0.065
12	1	13	0.001	0.032	0.02	0.005	0	0.058
14	1	11	0.001	0.026	0.014	0.003	0	0.044
16	1	11	0.001	0.036	0.015	0.003	0	0.055
18	1	9	0.001	0.044	0.014	0.003	0.001	0.063
20	1	9	0.001	0.043	0.011	0.002	0.001	0.058

TABLE A.2: Map (208 Nodes)

Parameters			Time Execution (ms)					
Merged Polys	Level	Number Nodes	Get S and G	Connect S and G	A*	Subpaths	Delete S and G	Total time
0	0	208	0	0	0.347	0	0	0.347
2	1	66	0.002	0.018	0.129	0.013	0.001	0.163
2	2	24	0.002	0.025	0.088	0.025	0	0.14
2	3	10	0.003	0.078	0.055	0.036	0	0.172
2	4	4	0	0	0.343	0	0	0.343
2	5	1	0	0	0.331	0	0	0.331
3	1	63	0.001	0.237	0.121	0.01	0	0.369
3	2	16	0.003	0.701	0.082	0.028	0	0.814
3	3	5	0.002	0.917	0.045	0.023	0.001	0.988
4	1	52	0.002	0.244	0.109	0.009	0	0.364
4	2	13	0.002	0.708	0.07	0.018	0	0.798
4	3	2	0.003	2.076	0.041	0.002	0	2.122
5	1	46	0.001	0.232	0.1	0.009	0.001	0.343
5	2	7	0.002	1.133	0.064	0.019	0.001	1.219
6	1	40	0.001	0.428	0.089	0.007	0	0.525
6	2	6	0.002	0.545	0.046	0.012	0.001	0.606
7	1	37	0.001	0.489	0.088	0.008	0	0.586
7	2	4	0.002	1.154	0.061	0.002	0.001	1.22
8	1	33	0.002	0.468	0.085	0.007	0	0.562
8	2	4	0.002	1.375	0.041	0.025	0.001	1.444
9	1	30	0.001	0.665	0.081	0.005	0	0.752
9	2	3	0.002	1.053	0.017	0.001	0	1.073
10	1	28	0.001	0.674	0.08	0.004	0	0.759
10	2	2	0	0	0.333	0	0	0.333
12	1	26	0.002	0.506	0.078	0.004	0.001	0.591
12	2	2	0	0	0.336	0	0	0.336
14	1	23	0.002	1.119	0.075	0.006	0.001	1.203
16	1	21	0.001	1.134	0.067	0.004	0.001	1.207
18	1	22	0.002	1.297	0.064	0.004	0	1.367
20	1	19	0.001	0.696	0.059	0.003	0	0.759

Bibliography

[1] Céline Loscos, David Marchal, and Alexandre Meyer. Intuitive crowd behaviour in dense urban environments using local laws. *TPCG*, pages 122–129, 2003. URL `http://dblp.uni-trier.de/db/conf/tpcg/tpcg2003.html#LoscosMM03`.

[2] Y. Chrysanthou F. Tecchia, C. Loscos. Visualizing crowds in real-time. *Computer Graphics forum*, December 2002.

[3] Franco Tecchia, Céline Loscos, Ruth Conroy, and Yiorgos Chrysanthou. Agent behaviour simulator (abs): A platform for urban behaviour development. *In GTEC'2001*, pages 17–21, 2001.

[4] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*, pages 566–580, 1996.

[5] A.; Mooijekind M.; Overmars M.H. Nieuwenhuisen, D.; Kamphuis. Creating small roadmaps for solving motion planning problems. *IEEE International Conference on Methods and Models in Automation and Robotics*, pages 531–536, 2005.

[6] R. Geraerts and M.H. Overmars. Automatic construction of high quality roadmaps for path planning. *Utrecht University: Information and Computing Sciences*, 2004.

[7] Mikko Mononen. Navigation-mesh toolset for games. *GitHub Recast and Detour*, November 2014. URL `https://github.com/memononen/recastnavigation`.

[8] L. P. Chew. Constrained delaunay triangulations. *Proceedings of the Third Annual Symposium on Computational Geometry*, pages 215–222, 1987. doi: 10.1145/41958.41981. URL `http://doi.acm.org/10.1145/41958.41981`.

[9] Marcelo Kallmann. Dynamic and robust local clearance triangulations. *ACM Trans. Graph.*, 33(5):161:1–161:17, September 2014. ISSN 0730-0301. doi: 10.1145/2580947. URL `http://doi.acm.org/10.1145/2580947`.

[10] Ramon Oliva and Nuria Pelechano. A generalized exact arbitrary clearance technique for navigation meshes. *ACM SIGGRAPH conference on Motion in Games*, 2013.

[11] D. Haumont, Olivier Debeir, and François X. Sillion. Volumetric cell-and-portal generation. *Comput. Graph. Forum*, 22(3):303–312, 2003. URL `http://dblp.uni-trier.de/db/journals/cgf/cgf22.html#HaumontDS03`.

[12] Ramon Oliva and Nuria Pelechano. Automatic generation of suboptimal navmeshes. 7060:328–339, 2011. URL `http://dblp.uni-trier.de/db/conf/mig/mig2011.html#OlivaP11`.

[13] F. Lamarche. TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints. *Computer Graphics Forum (Proc of Eurographics 2009)*, 28(2), 2009.

[14] Carl-Johan Jorgensen and Fabrice Lamarche. From geometry to spatial reasoning : automatic structuring of 3D virtual environments. *Motion In Games*, 2011. URL `http://www.irisa.fr/mimetic/Fabrice.Lamarche/Articles/MIG2011-spatial-reasonning.pdf`.

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

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

[17] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. Ara* : Anytime a* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems 16*, pages 767–774, 2004. URL `http://papers.nips.cc/paper/2382-ara-anytime-a-with-provable-bounds-on-sub-optimality.pdf`.

[18] Sven Koenig and Maxim Likhachev. D*lite. *Eighteenth National Conference on Artificial Intelligence*, pages 476–483, 2002. URL `http://dl.acm.org/citation.cfm?id=777092.777167`.

[19] Maxim Likhachev, David Ferguson , Geoffrey Gordon, Anthony (Tony) Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[20] William Lee and Ramon Lawrence. Trading space for time in grid-based path finding. *AAAI*, 2013. URL http://dblp.uni-trier.de/db/conf/aaai/aaai2013.html#LeeL13.

[21] Nathan R. Sturtevant. Memory-efficient abstractions for pathfinding. *AIIDE*, pages 31–36, 2007. URL http://www.aaai.org/Papers/AIIDE/2007/AIIDE07-006.pdf.

[22] Ramon Lawrence and Vadim Bulitko. Database-driven real-time heuristic search in video-game pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games*, 5(3):227–241, 2013. URL http://dblp.uni-trier.de/db/journals/tciaig/tciaig5.html#LawrenceB13.

[23] Vadim Bulitko, Yngvi Björnsson, and Ramon Lawrence. Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding. *Journal of Artificial Intelligence Research (JAIR)*, 39:269–300, 2010.

[24] Norman I. Badler Tianyu Huang, Mubbasir Kapadia and Marcelo Kallmann. Path planning for coherent and persistent groups. *Proceedings of the IEEE International Conference on Robotics and Automation*, 2014.

[25] Mubbasir Kapadia, Kai Ninomiya, Francisco Garcia, , and Norman I. Badler. Constraint-aware navigation in dynamic environments. *Motion on Games*, 2013.

[26] C.W. Warren. Global path planning using artificial potential fields. *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, 1:316 – 321, 1989. URL http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=100007&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D100007.

[27] Wouter G. van Toll, Atlas F. Cook, IV, and Roland Geraerts. Real-time density-based crowd simulation. *Comput. Animat. Virtual Worlds*, 23(1):59–69, February 2012. ISSN 1546-4261. doi: 10.1002/cav.1424. URL http://dx.doi.org/10.1002/cav.1424.

[28] Mubbasir Kapadia Francisco Garcia and Norman I. Badler. Gpu-based dynamic search on adaptive resolution grids. *Proceedings of the IEEE International Conference on Robtics and Automation*, 2014.

[29] Cory D. Boatright Mubbasir Kapadia, Francisco Garcia and Norman I. Badler. Dynamic search on the gpu. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[30] Kalin Gochev, Benjamin J. Cohen, Jonathan Butzke, Alla Safonova, and Maxim Likhachev. Path planning with adaptive dimensionality. *SOCS*, 2011. URL http://dblp.uni-trier.de/db/conf/socs/socs2011.html#GochevCBSL11.

[31] Alexander Shoulson, Max L. Gilbert, Mubbasir Kapadia, and Norman I. Badler. An event-centric planning approach for dynamic real-time narrative. *Proceedings of Motion on Games*, pages 99:121–99:130, 2013. doi: 10.1145/2522628.2522629. URL `http://doi.acm.org/10.1145/2522628.2522629`.

[32] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004.

[33] Daniel Harabor and Adi Botea. Hierarchical path planning for multi-size agents in heterogeneous environments. *CIG*, pages 258–265, 2008. URL `http://dblp.uni-trier.de/db/conf/cig/cig2008.html#HaraborB08`.

[34] Yan Li, Lan-Ming Su, and Wen-Liang Li. Hierarchical path-finding based on decision tree. *RSKT*, 7414:248–256, 2012. URL `http://dblp.uni-trier.de/db/conf/rskt/rskt2012.html#LiSL12`.

[35] Marcelo Kallmann and Mubbasir Kapadia. Navigation meshes and real-time dynamic planning for virtual worlds. *ACM SIGGRAPH 2014 Courses*, pages 3:1–3:81, 2014. doi: 10.1145/2614028.2615399. URL `http://doi.acm.org/10.1145/2614028.2615399`.

[36] Jos B.T.M. Roerdink and Arnold Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundam. Inf.*, 41(1,2):187–228, April 2000. ISSN 0169-2968. URL `http://dl.acm.org/citation.cfm?id=2372488.2372495`.

[37] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE COMPUTER GRAPHICS AND APPLICATIONS*, 21:24–35, 2001.

[38] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997. URL `http://dx.doi.org/10.1137/S1064827595287997`.

[39] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

[40] Blender 2.72. *3D creation for everyone, free to use for any purpose.*, November 2014. URL `http://www.blender.org/`.