Hierarchical Path Finding to Speed up Crowd Simulation using Navigation Meshes

Carlos Fuentes

***Abstract*—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. A new hierarchical path finding solution based on navigation meshes is proposed in order to minimize the complexity in big environments. The method has two steps: Firstly, it creates the hierarchy tree based on a recursive partitioning. Then, the optimal path is found in the hierarchy at certain level. This approach performs much faster path finding calculations than a common A*. These claims are verified on big environments.**

***Index Terms*—A*, crowd simulation, graph algorithms, graph partitioning, hierarchical path finding, HPA*, navigation meshes, path planning, Recast, route planning, shortest path search.**

***Resumen*— La búsqueda de ruta es un problema común en los juegos de computador. La mayoría de los videojuegos requiere simular miles o millones de agentes que interactúan y navegan en un mundo 3D mostrando capacidades tales como persecución, la búsqueda o la intercepción de otros agentes. Se propone una nueva solución de búsqueda de ruta jerárquica basada en mallas de navegación con el fin de minimizar la complejidad en entornos grandes. El método tiene dos etapas: En primer lugar, se crea un árbol de jerarquía basada en un particionamiento recursivo. Después, el camino óptimo se encuentra en la jerarquía a cierto nivel. Este enfoque lleva a cabo cálculos de búsqueda de ruta más rápido que un A* común. Estas afirmaciones son verificadas en entornos grandes.**

***Palabras claves*—A*, algoritmos de grafos, búsqueda de ruta jerárquica, búsqueda de trayectoria más corta, HPA*, mallas de navegación, partición de grafos, planificación de trayectorias, Recast, simulación de multitudes.**

I. 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 behavior 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.

This new method proposes a new hierarchical path finding solution for big environments. A navigation mesh is used as abstract data structure to partition the 3D world. Then, a graph representation is extracted and considered 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 cannot 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.

II. RELATED WORK

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.

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

This paper was submitted by corresponding author on 13 February 2015 for review. This work was supported in part by the MOVING research group in the Polytechnic University of Catalonia.

The author, was with Polytechnic University of Catalonia, Barcelona, Spain. He is now an employee in ThoughtWorks (e-mail: cfuentes@thoughtworks.com).

the graph belongs to, the smaller the number of nodes the graph has.

A. *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, roadmaps, and navigation meshes (NavMesh).

1) *Regular Grids*

Regular grids are the simplest way to represent the 3D world. This regular cell grid is commonly used for 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 behaviors 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. 1).

If the virtual world is small and grid-like, this technique is useful. Also, this approach is ideal for 2D terrains where the

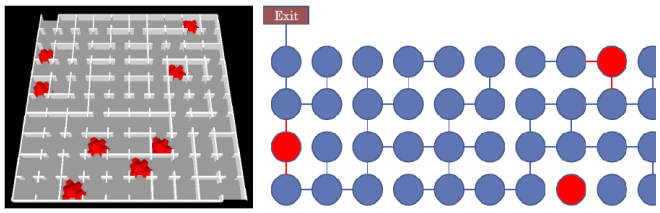

Fig. 1. Regular Grid Extraction with two states: World environment (a). Extracted graph for the upper part of the grid (blue nodes have opened state and red nodes have blocked state) (b).

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 do not 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, for grid searches, a path smoothing step needs to be applied in the post-processing phase resulting in expensive queries. (See Fig. 2).

Therefore, using large grids that are composed for thousands of cells could easily exceed the memory

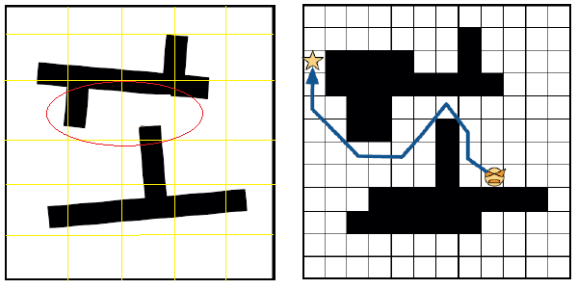

Fig. 2. Limitations of Grid Methods: Walkable and obstacle area inside the same cell (a). Restricted movement (fixed angles) (b).

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) *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 neighbors. 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 shortest path query. (See Fig. 3).

A roadmap can also use a Voronoi diagram classifier [6]. The Voronoi diagram is one of the most popular structure

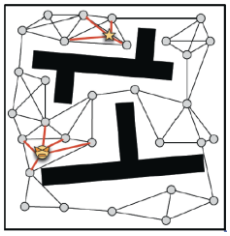

Fig. 3. Probabilistic Roadmap Planner: Connect Start and Goal node to the roadmap.

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 collision free from the entity is. 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. 4).

Unfortunately, the PRM method leads to low quality

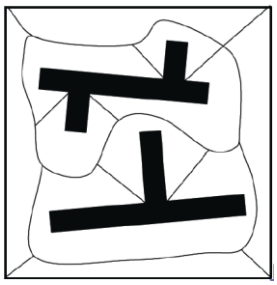

Fig. 4. Voronoi-Based Roadmaps: Retract PRM (nodes and edges) to the medial axis.

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 drives to larger graphs when many milestones are needed.

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 few 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 must 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 must facilitate the computation of quality paths. At least, locally shortest paths must be provided.
- **Arbitrary clearance.** A navigation mesh must provide an efficient mechanism for computing paths with arbitrary clearance from obstacles. No pre-computed clearance valued must be known.
- **Representation robustness.** A navigation mesh must be robust to degeneracies in the description of the environment. Each description of obstacles must be handled such as intersections, overlaps, etc.
- **Dynamic updates**. A navigation mesh must efficiently update itself to accommodate dynamic changes in the environment.

An example of the world representations are shown the Fig. 5. [1]

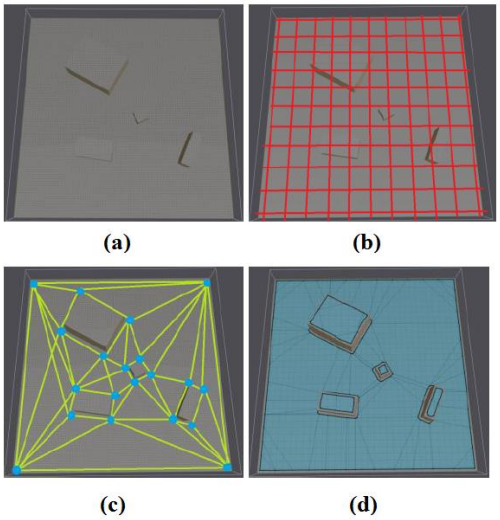

Fig. 5. Representations of the environment division. Empty map a). Regular Grid map b). Roadmap c). NavMesh d).

B. Hierarchical Subdivision

There has been some research recently focused on hierarchical path finding techniques using the A star algorithm such as HPA* [17] 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* [8] searches inside each cluster, for all pairs of abstract nodes that shared the same cluster. 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. Finally, HPA* softs the path in an

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

attempt to undo some of the error introduced with the merging of cluster border vertices. However, the final path is still not optimal. 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*) [19]. Basically, it is an extension of HPA* which allow multi-size agents to efficiently

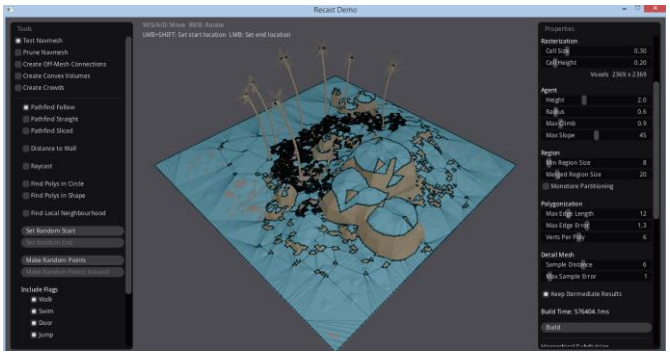

Fig. 7. Recast Tool software. (Model: Tropical Islands (12666 polygons)).

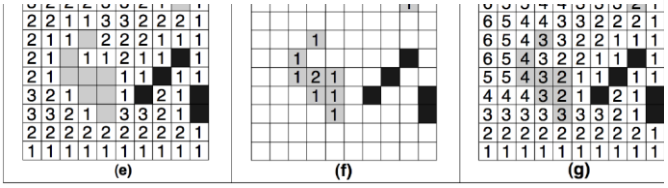

Fig. 6. Clearance Values: (a) - (d) Computing clearance; the square is expanded until a hard obstacle is encountered. (e) - (g) Clearance values for different capabilities.

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. 6) 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. 6.

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* [17]. The entrances between clusters, inter-edges, intra-edges are calculated considering the agent properties.

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

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 [8]. 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*) [9]. 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 is 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 [10]. 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 scratch but requires more memory.

Finally, Anytime Dynamic A* (AD*) [11] 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.

III. HIERARCHICAL PATH FINDING

This method is based in the HPA* algorithm described in the previous section. In order to apply this approach, an initial discretization is needed of the 3D world, and navigation meshes are the most accurate for this purpose.

One advantage of using navigation meshes in comparison with grid approaches, is that the number of cells is much smaller and thus the initial graph abstraction is smaller.

A. *NavMesh division*

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 [12]. It is automatic, which means that any geometry can be as an input and it will output a robust mesh. It is also fast which means swift turnaround times for level designers. Recast tool also

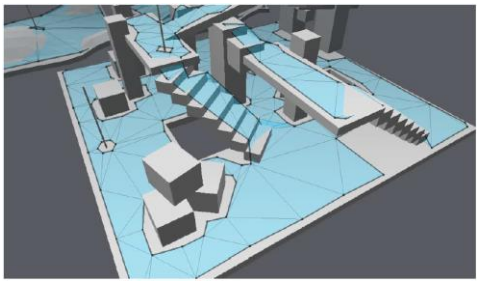

Fig. 12. Triangulation.

provides an optimized A* implementation into Detour project classes [7]. Fig. 7 shows the tool.[2]

Recast tool generates a NavMesh from a triangle soup. It receives an arbitrary polygon soup with triangles marked as walkable. The reconstruction is done in the preprocessing step of the algorithm and it is divided as follows:

1) Voxelize the polygons

The voxel mold is built 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.

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. 8).

3) Build watershed partitioning and filter out unwanted regions

The Watershed Transform [13] finds the catchment basins

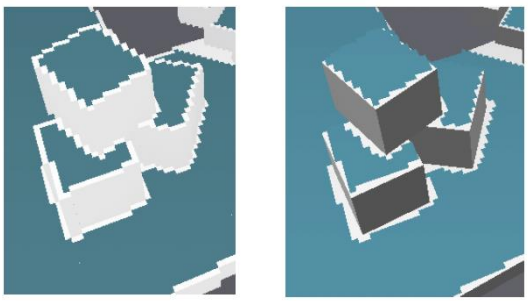

Fig. 8. Build navigable space from solid voxels: Voxelization with walkable cells marked (a) and walkable cells overlayed on top of input geometry (b).

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

by building the distance 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. 9). Then, a filter pass is applied to remove small unconnected regions and merge small regions together. The result is a set of nonoverlapping simple regions that can be used as basis for generating waypoint graphs.

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

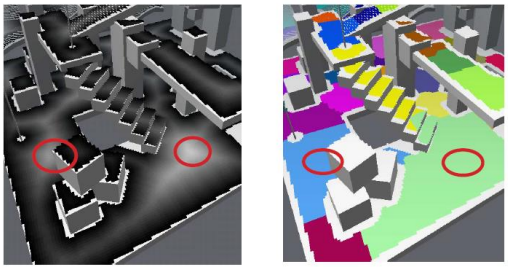

Fig. 9. Build watershed partitioning and filter out unwanted regions: The catchment basins become the centers of the regions. (White areas represent lower region).

boundaries of the regions. The cell corner points which will form the polygon and the neighbor region ID are stored. Finally, the contours are simplified using Ramer-Douglas-Peucker algorithm [14]. The algorithm finds initial segments and locks vertices which are between two different regions, if the region is not connected, it locks two extreme 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. 10). The result is a set of simple polygons. (See Fig. 11).

5) Triangulate the region polygons and build triangle connectivity

Recast uses a modified algorithm from Computational

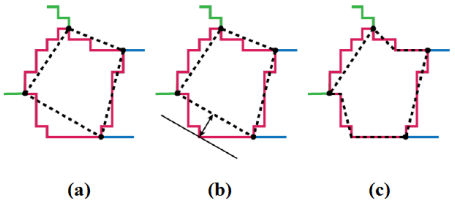

Fig. 10. Ramer-Douglas-Peucker algorithm: Initial vertices at region edges (a); Find vertex with maximum error, and subdivide (b); Iterate until certain error criteria is met (c).

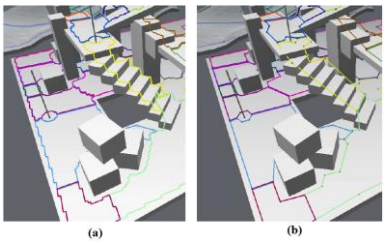

Fig. 11. Contours: Traced contours (a); Simplified contours (b).

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 perfect for path finding and spatial reasoning about the level. See Fig. 12.

These five steps are illustrated in the Fig. 13. 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. 14). 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 this approach.

Once the spatial partition has been done by the Recast tool, the creation of the initial graph is performed. The method has mainly two steps:

- Hierarchical Subdivision. This is done in the preprocessing part. The hierarchy of graphs is created. 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, the path finding is calculated over the graph in that level.

B. *Hierarchical Subdivision*

The first step is to build the framework for hierarchical searches that is defined as a tree of graphs. The lowest graph of

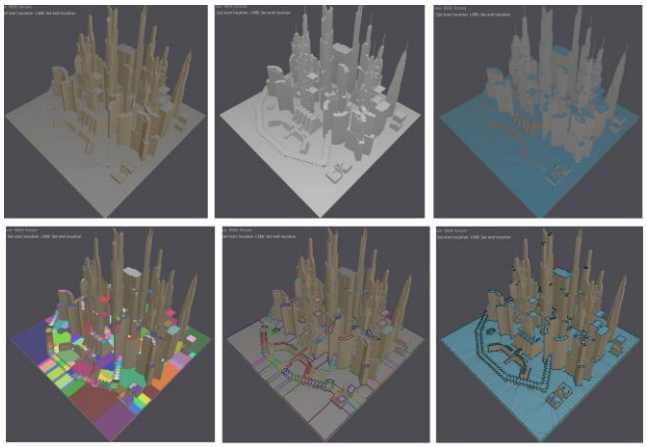

Fig. 13. Recast steps (from left to right): Input mesh and the five steps of Recast process. (Model: Scifi City (2090 polygons)

the hierarchy (G_0= (V_0, E_0)) is computed 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, an edge is created between them. See Fig. 15.

Once the lowest level graph is created, the upper levels of the hierarchy are recursively built by partitioning each level until it reaches either the minimum number of the nodes in a graph or a 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,

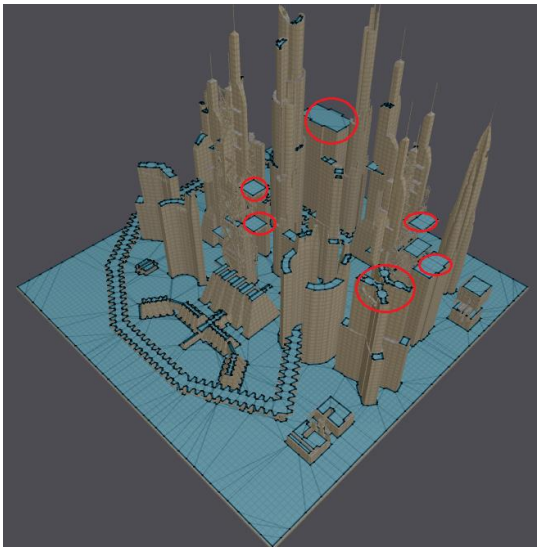

Fig. 14. Recast tool: Unreachable walkable areas. (Model: Scifi City (2090 polygons)).

the *k*-way multilevel algorithm (MLkP) [15] is used to reduce the size of the graph by collapsing vertices and edges. This algorithm is faster than others multilevel recursive bisection algorithms. The process is described as follows:

First of all, a series of successively smaller graphs is

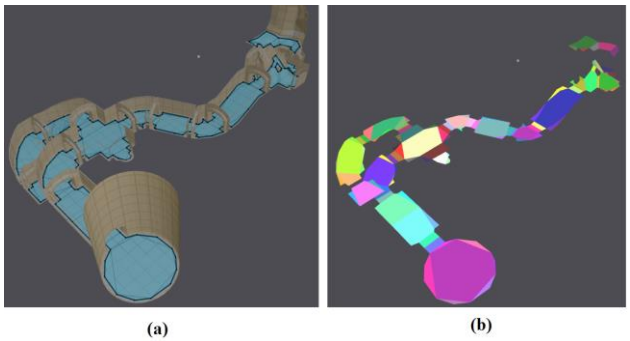

Fig. 15. Hierarchical Subdivision: The NavMesh triangulation of the model (a). The graph of the lowest level (level 0) (b). Nodes are painted in different colors. Edges connects a node with its neighbors (Model: Dungeon (120 polygons)).

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 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 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 [15]. Each partition contains roughly $|V_0|/k$ vertex weight of the original graph. The division is done by Kernighan–Lin (KL) partitioning algorithm [16] 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. 16.

The procedure allows 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. An external implementation is used for this purpose. It is called METIS library[3]. 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.

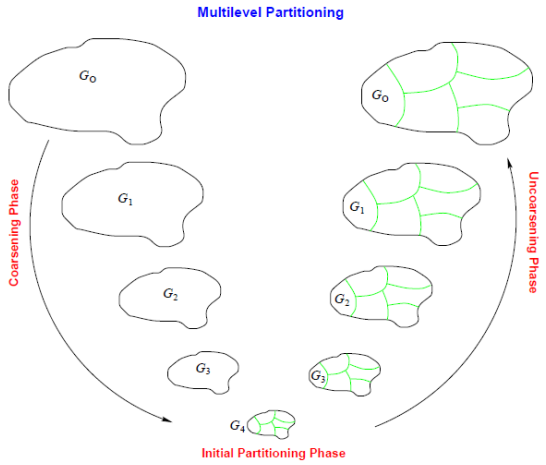

Fig. 16. The three phases of multilevel *k*-way graph partitioning. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph.

Algorithm 1 Hierarchical subdivision algorithm

```
1:  procedure BUILDHIERARCHY(numMergedNodes,levels)
2:      numLevels = 0
3:      repeat
4:          numParts ← parentGraph.numNodes/numMergedNodes
5:          if (numParts < 2) then return
6:          partitions ← METISPartGraphKway(parentGraph.numNodes, numParts)
7:          partitions ← checkPartitions(partitions, parentGraph.numNodes, numParts)
8:          currentGraph ← buildGraph(partitions)
9:          parentGraph ← currentGraph
10:         numLevels++
11:     until numLevels < levels
```

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 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.

Once the partitions are established, 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, an A* is calculated 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. 17, the partitions, portal and intraedges are illustrated.

[3] 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 2 Build Graph algorithm

```
procedure BUILDGRAPH(partitions)
    for i ← 1, partitions do
        currentGraph.AddNode(i)
    for i ← 1, partitions do
        nodes ← findNodes(p)
        for node ← 1, nodes do
            for neighbour ← 1, node.numEdges do
                if node.partition ≠ neighbour.partition then
                    currentGraph.AddEdge(node.parentNode, neighbour.parentNode)
                    portals.Add(node)
            for j ← 1, portals do
                for k ← 1, portals do
                    if j ≠ k then
                        cost ← findPath(j, k)
                        currentGraph.AddIntraEdge(node, j, k, cost)
            portals.Clear()
```

An example of the hierarchical subdivision step is shown in the Fig 18.

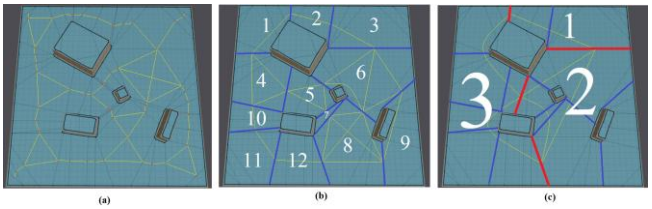

Fig. 17. Hierarchical Subdivision: (Simple map, *numMergedNodes* = 5, *levels* = 5). Portals are presented with red dots. IntraEdges are painted with yellow lines. Partitions are exposed with black, blue and red separation lines respectively. Level 0 = 76 nodes (a), Level 1 = 12 nodes (b), Level 2 = 3 nodes (c). (Model: Simple Map (76 polygons)).

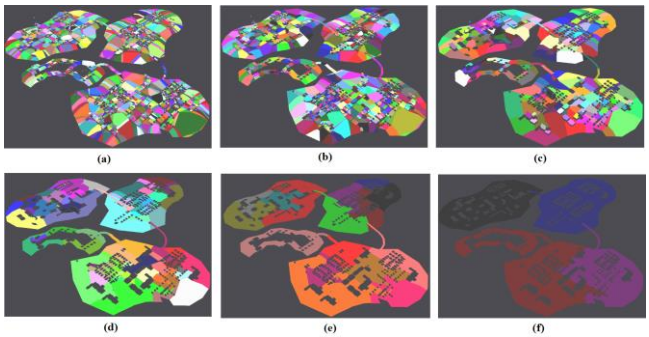

Fig. 18. Hierarchical Graphs: (City Islands, $numMergedNodes = 3, levels = 10$). Level 0 = 5151 nodes (a), Level 1 = 1469 nodes (b), Level 2 = 316 nodes (c), Level 3 = 72 nodes (d), Level 4 = 17 nodes(e), Level 5 = 4 nodes (f). (Model: City Islands (5515 polygons)).

C. *Path finding Computation*

Once the hierarchical subdivision is created, the path finding computation can be done at any level of the hierarchy. The second step consists in searching the shortest path from the start node (*S*) to the goal node (*G*) in a specific graph. This online search brings a performance improvement as any graph in the hierarchy is smaller than the one in level 0. (See Fig. 22).

The path finding computation has the following five phases:

1) *Find S and G at a 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, *S* and *G* nodes are obtained in the graph at level 0 by searching for their positions. Finally, their parents are recursively searched by passing through all the levels in between until reach the desired level. If *S* and *G* nodes are in the same partition, a normal A* is run between them and the path finding is completed.

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. A temporal Start node is connected to each portal in the partition that contains it. Then, an A* is computed between *S* and the center of each portal in the partition. The path nodes and costs are stored for each portal. Finally, a new intraedge is added between the start node the portal inside the current partition. This step is repeated for *G* in its respective partition. (See Fig. 19).

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.

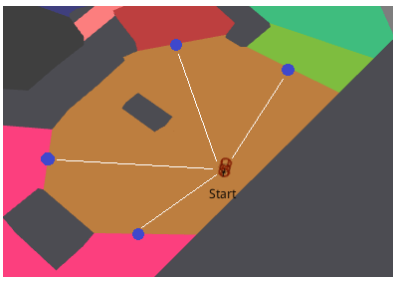

Fig. 19. Connect Start node to the graph: Blue circles are portals of the orange partition. White lines are the computed intraedges. Gray polygons are obstacles or no walkable areas.

3) *Search for a path between S and G at the highest level*

Once the *S* and *G* are temporally linked to the graph, an A* search is performed in the current graph. The time execution becomes faster because the number of nodes is significantly smaller than the graph at level 0.

4) *Obtain optimal subpaths*

The path planning computation gives all the partitions which are part of the optimal solution. For each of them, the path nodes are recursively got for each lower level until the lowest level is reached in the hierarchy. At the end, the full path is obtained to go from *S* to *G* at the level 0.

5) *Delete temporal nodes*

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

The preprocessing step is shown in the Fig. 20, where a hierarchical subdivision has been applied on a map model. The *S* and *G* positions are denoted in white letters. In this sample,

the shortest path is found from *S* to *G* at level 2. The online step is shown in Fig. 21. The start and goal nodes are connected to their respective portals and run a common A* at level 2. Finally the sub paths are found until level 0 is reached.

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.

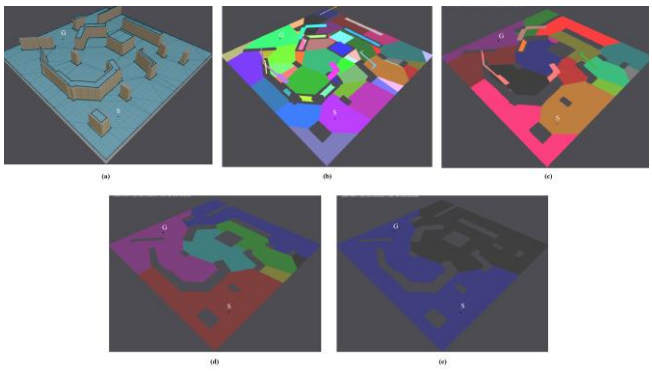

Fig. 20. Hierarchical Subdivision (From (a) to (e)). The start and goal nodes are written in white.

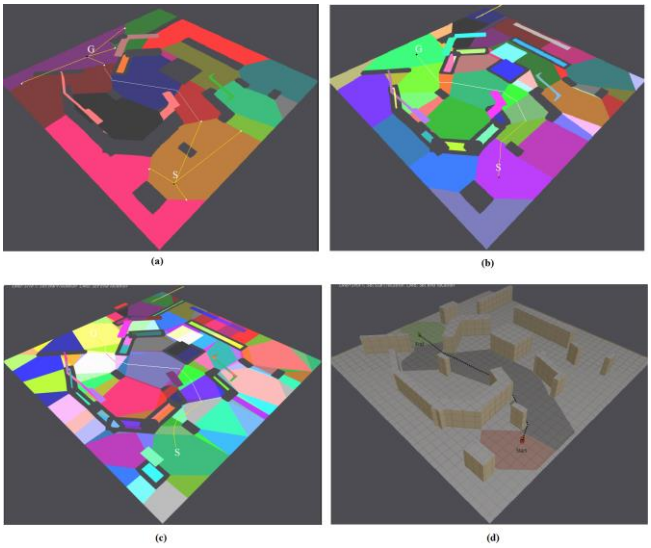

Fig. 21. Path finding Computation: *S* and *G* are linked to their partitions at level 2 (a). Sub paths are calculated until level 0 is reached. (Level 1 (b) and Level 0 (c)). The final result of the shortest path between *S* and *G* at level 2 (d).

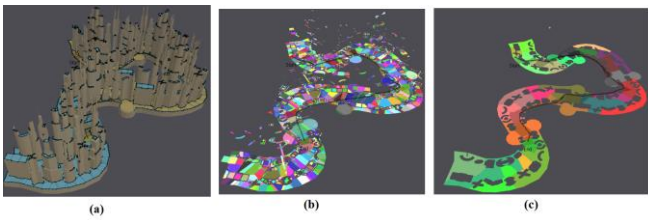

Fig. 22. Path finding Computation: (Serpentine Islands, *numMergedNodes* = 4, *levels* = 10) Path finding at level 0 (3908 nodes) (a). Path finding at level 0 where each node has a different color (b). Path finding at level 3 (28 nodes) (c).

IV. Results and Discussion

The results were 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, the analysis is focused on how the time taken for each of the steps affects the total time. Also, the impact is explored by varying the number of merged nodes for the different levels on the performance of path finding with the suggested method. The performance results have been tested on an Intel® Core™ i7 processor with NVIDIA® GeForce® 610M graphics card and 8 GB of RAM.

A. Performance Test

The 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.

1) Number of Nodes

The number of resulting nodes is compared in each level in the hierarchy as the number of merged nodes is increased from one level to the next one. It is expected that the higher the level, the lower the number of nodes. As an example, the Fig. 23 shows the results obtained for the Sirus City. The chart shows number of nodes falls steadily over the 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 but every time that a partition is created. This partition is checked whether it has

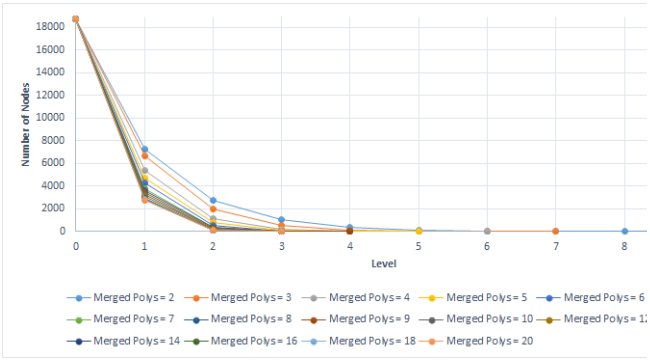

Fig. 23. Level vs Number of Nodes (Sirus City).

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.

2) Time Execution

Firstly, the total time of performing a path finding computation was analyzed at different levels against the execution time of performing path planning in level 0 (i.e without any hierarchy).

The sample model is a map with 2615 nodes in its initial graph. 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. To begin, the Fig. 24 shows the computation time over City Colony. It can

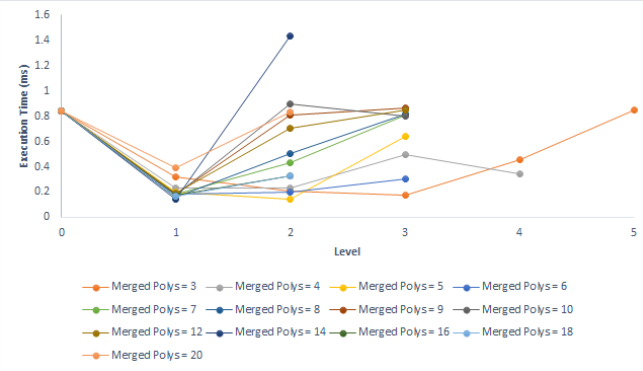

Fig. 24. Level vs Execution Time (City Colony).

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. 25).

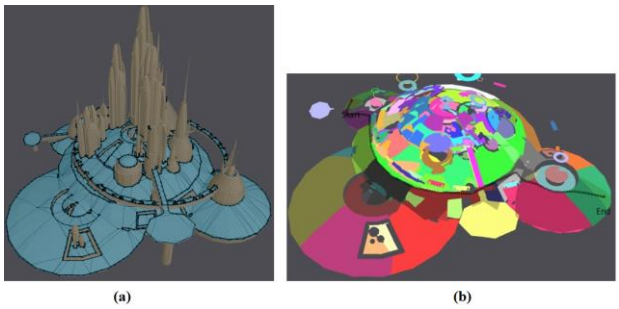

Fig. 25. City Colony model (a). Shortest path (*numMergedNodes* = 5, *level* = 2) (b)

To better understand where the bottlenecks of our algorithm appear, the partial times are compared for computing the entire online step for each level in the hierarchy. The online search has been divided in five steps in order to analyze the partial times in this process. See algorithm 3.

3) *Find S and G at certain level*

The Fig. 26 illustrates the time of getting the partitions to which Start and Goal nodes respectively belong to a certain level. Those partitions are obtained by recursively searching *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

Algorithm 3 Online Search Pseudo-code

```
procedure ONLINESEARCH(startNodeId, startNodePos)
    endNodeId, endNodePos, level
    Find S and G at 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
```

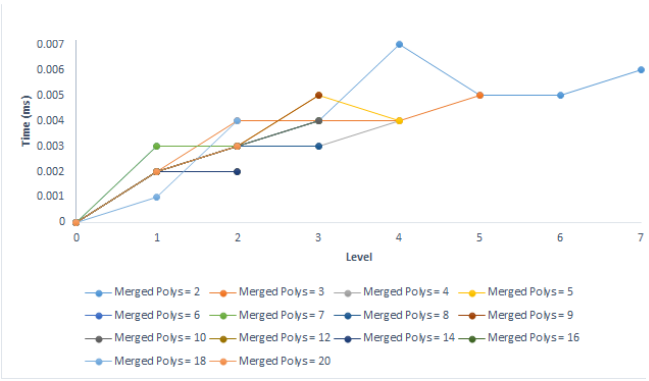

Fig. 26. Get *S* and *G* Time vs Level

significant in the total time (values less than *0.005 ms*).

4) *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

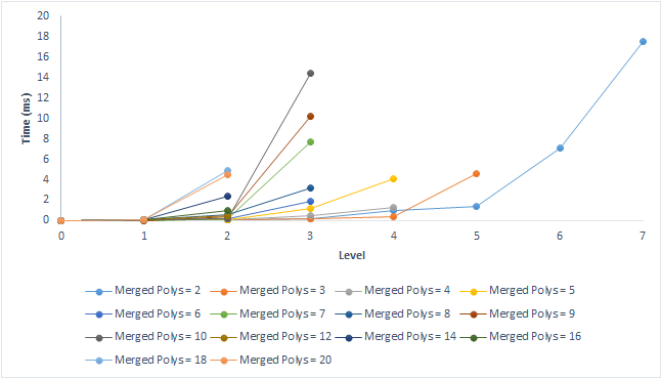

Fig. 27. Link *S* and *G* Time vs Level

stored by adding an intraedge.

The figure shows an upward trend throughout the levels of the hierarchy. It exposes a particular strong growth in the highest levels due to the number of portals being bigger in the upper levels. (See Fig. 27). This particular behavior is due to the fact that we have to execute as many A* searches as the number of portals the partition has.

5) 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. 28 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

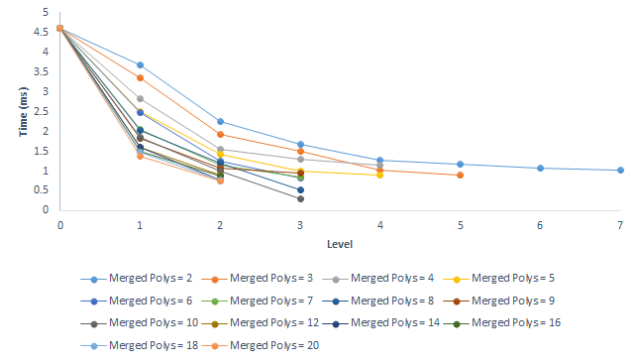

Fig. 28. A* Time vs Level

11 with a partial time *0.287ms*.

6) Obtain optimal subpaths

The chart 30 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 reach the level 0. Those subpaths have 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 k. Also, there are not intermediate paths between them. Fig. 29 illustrates this

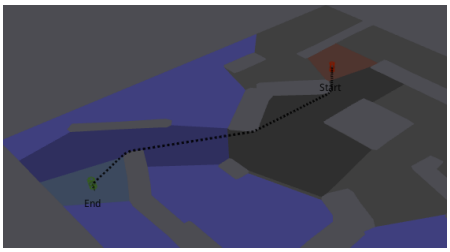

Fig. 29. Obtain optimal subpaths. The map model has two nodes at level 3. No intermediate paths are calculated.

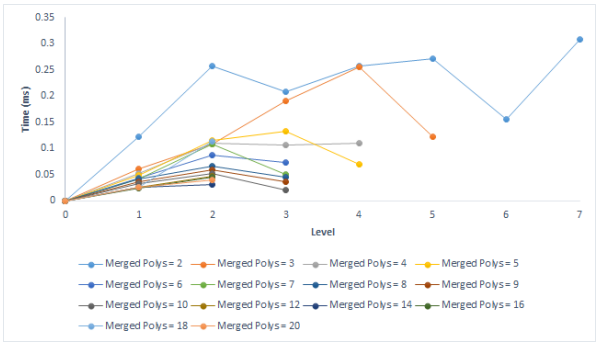

Fig. 30. Time vs Level.

scenario.

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

7) Delete S and G

Deleting temporal *S* and *G* nodes have an insignificant time compare to other partial times during the online process.

V. Conclusion

With the booming growth of video games, there is a great demand on path finding algorithms. In this method, a new hierarchical path finding framework is presented 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. The approach has a tree hierarchy of graphs where the searching can be performed at any level.

The main contributions of this 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.

The 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 than A*). For small models, a common A* is enough. The trade-off between the chosen level and the size of partitions is important. Also, the approach shows better performance in non-widespread environments.

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

VI. Future Work

Despite the 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 the 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, this approach could be extended to work under dynamic environments where the replanning could be done only at certain level of the hierarchy.

ACKNOWLEDGMENT

The research reported in this document/presentation was based on the master thesis of the author. The research, views and conclusions contained in this document are those of the author and his advisor. The author would like to thank to Nuria Pelechano Gomez for her tutoring during his studies and research in the Polytechnic University of Catalonia.

REFERENCES

[1] Loscos, Céline and Marchal, David and Meyer, Alexandre, *Intuitive Crowd Behavior in Dense Urban Environments using Local Laws*., Theory and Practice of Computer Graphics, 2003. Proceedings, pp. 122 - 129. Jun 2003.

[2] F. Tecchia, C. Loscos, Y. Chrysanthou, *Visualizing Crowds in Real-Time*, Computer Graphics forum, pp. 753 - 765, Dec 2002.

[3] Franco Tecchia and Céline Loscos and Ruth Conroy and Yiorgos Chrysanthou, *Agent Behaviour Simulator (ABS): A Platform for Urban Behaviour Development*, In GTEC'2001, pp. 17-21, 2001.

[4] Lydia Kavraki and Petr Svestka and Jean-claude Latombe and Mark Overmars, *Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*, Robotics and Automation, vol. 12, pp. 566 - 580, Aug 1996.

[5] Nieuwenhuisen, D.; Kamphuis, A.; Mooijekind, M.; Overmars, M.H., *Creating Small Roadmaps for Solving Motion Planning Problems*, IEEE International Conference on Methods and Models in Automation and Robotics, pp. 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] M. Mononen. (2015, April 8), *Navigation-mesh Toolset for games*, GitHub Recast and Detour, 2014. Available: https://github.com/memononen/recastnavigation.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael., *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems, Science, and Cybernetics, vol. 4, pp. 100 - 107, July 1968.

[9] Maxim Likhachev and Geoffrey J. Gordon and Sebastian Thrun., *ARA*: Anytime A* with Provable Bounds on Sub-Optimality*, Advances in Neural Information Processing Systems 16, 2004.

[10] Koenig, Sven and Likhachev, Maxim. D**Lite*, Eighteenth National Conference on Artificial Intelligence, pp. 476 - 483, 2002.

[11] 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), Jun 2005.

[12] Haumont, D. and Debeir, Olivier and Sillion, François X., *Volumetric Cell-and-Portal Generation*, Comput. Graph. Forum, 2003.

[13] Roerdink, Jos B.T.M. and Meijster, Arnold., *The Watershed Transform: Definitions, Algorithms and Parallelization Strategies*., Fundam. Inf. Forum, vol. 22, pp. 303 - 312, Sep 2003.

[14] David P. Luebke., *A Developer's Survey of Polygonal Simplification Algorithms*., IEEE Computer Graphics and Applications., vol. 21, pp. 24 - 35, May 2001.

[15] G. Karypis and Vipin Kumar., *Multilevel k-way Partitioning Scheme for Irregular Graphs.*, Journal of Parallel and Distributed Computing., vol. 48, pp. 96 - 129, Jan 1998.

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

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

[18] Joseph O'Rourke, *Computational Geometry in C.*, 2nd ed, Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On, pp. 258 - 265, Dec. 2008.

[19] Daniel Harabor and Adi Botea., *Hierarchical path planning for multi-size agents in heterogeneous environments*, CIG, 2008.

Carlos Fuentes received the Master degree in Computer Graphics and Virtual Reality from Polytechnic University of Catalonia, Barcelona in 2014. From 2013 to 2014, he was a research assistant with the Moving Research Group in UPC. His research interest includes crowd simulation and autonomous behavior. He is currently working in ThoughtWorks Ecuador.