Robust and Scalable Navmesh Generation with multiple levels and stairs support

Guillaume Saupin
CEA,LIST,Laboratoire de Simulation Interactive
18 route du panorama
92260 Fontenay aux Roses, France
guillaume.saupin@cea.fr

Jeremie Le Garrec
CEA,LIST,Laboratoire de Simulation Interactive
18 route du panorama
92260 Fontenay aux Roses, France
jeremie.le-garrec@cea.fr

ABSTRACT

Automatically planning motion for robots or humans in a virtual environment is a complex task. The navigation cannot be done directly on the geometric scene. An internal representation of the environment is necessary. However, virtual environments complexity is growing at an important rate, and it is not unusual to work with millions of triangles and area as large as some square kilometers. In these cases, grid based methods require too much memory, and are too slow for A* algorithms. Polygons based methods are more memory and computation efficient, but they are difficult to generate and require complex preprocessing of the input mesh to ensure nice topological properties. In this paper, we propose an hybrid method, which uses 3D voxelization to generate a clean polygon mesh simple enough to perform fast A* search. Using this approach, we robustly generate *Navigation Meshes* for large environments with fine details, described by millions of triangles, without any assumption on the quality of the input mesh. The input mesh can contains holes, degenerated or intersecting triangles.

Keywords

navigation mesh generation, mesh simplification, path planning

1 INTRODUCTION

1.1 Motivations

Many tasks in computer animation require the computation of free paths. Computing them must be done in a very short time, as the requirement on the frame rate of this kind of application is very high.

To achieve such performances, path planning algorithms use internal representations detailed enough to model the complex virtual worlds, and simple enough to allow fast path request.

Computing these internal representations from the original meshes is a difficult task, virtual environments frequently being detailed, complex, large, and of various quality.

In this paper, we propose a method to efficiently and robustly compute polygon based maps, usually referred as *Navigation Meshes* from any triangles mesh.

1.2 Related works

As stated in apper [1], an important work has been done in the field of free path computation.

Lamarche [10] distinguish three approaches to path planning:

- Graph based methods, commonly referred as *roadmap* methods. The navigable environment is represented by a set of connected paths. They do not represent the entire environment and can lead to suboptimal paths. See papers [9] and [14].
- Cell decomposition methods, where the navigable environment is represented as a set of connected areas. The resulting map is referred as a *Navigation Mesh*. They were introduced by Snook [16].
- Potential fields methods, where the navigation is done according to the gradient of a potential field.

Paper [10] reviews of all these methods.

We choose to use *Navigation Meshes*, a good choice for applications requiring collision detection on environment boundaries, as well as path planning with arbitrary clearance.

1.2.1 Navigation Meshes

Path planning maps are usually based on two kinds of internal representations:

- Grid based discretization: the virtual environment is discretized using voxels.
- Polygon based discretization: the navigable areas are discretized using polygons.

In both cases, the discretization is associated with topological information regarding the connections between adjacent voxels or polygons.

Most of the proposed methods for path planning on *Navigation Meshes* are based on the seminal paper [3] introducing the A^* algorithm. This method has been applied with success on both polygonal [10] and voxels grids [1].

In the case of grid based method, generating the discretization and the topological information is a straightforward process. Some papers [2, 11, 12], propose implementation on GPU. However, for large environments with fine details, these method requires large amounts of memory, and became inefficient for path planning. This approach was introduced by Bandi in [1].

Polygons based methods are much more optimal internal representation. They require less memory and allow faster path finding. They can even be used for dynamic environments [17].

Many path requests can be done on such representations. Kallmann [7] explores the use of triangulation as *Navigation Meshes*. In paper [8] a method is proposed to compute shortest pathes with arbitrary clearance.

However, creating *Navigation Meshes* by hand is a tedious and error prone process, and automatically generate them from any input mesh is a challenging task.

Lamarche [10] has proposed a method using prismatic spatial subdivision, but it implies that all triangles intersections are materialized by edges. This assumption on the quality of the input mesh requires a potentially time consuming cleaning step.

Mononen [13] has published an open source implementation of *Navigation Mesh* generation similar to our method. He uses voxelization to extract walkable areas, and triangulate them to create the *Navigation Mesh*. However, this method is not scalable, and doesn't handle staircases or very large and detailed environment. He also doesn't guarantee that all the *Navigation Meshes* vertices lies on boundaries, which prevents the use of path planning algorithms with arbitrary clearance.

Finally, Oliva *et al.* [15] propose a method to automatically generate suboptimal *Navigation Meshes*, but they need clean polygons as an input.

We present a method that can robustly generate such polygonal maps from any input mesh, with low memory requirement, and in a matter of seconds. This method also handles ramps and staircases.

1.3 Contributions

In this paper we introduce new algorithms dedicated to produce a quality *Navigation Mesh* using a tiling process.

In particular, we address the difficult problem of merging the regions segmented independently on each tile without introducing intermediate vertices. The proposed algorithms perform this operation robustly, while using a tiling process allows to perform a significant part of the computations in parallel as well as allowing to handle large 3D environments.

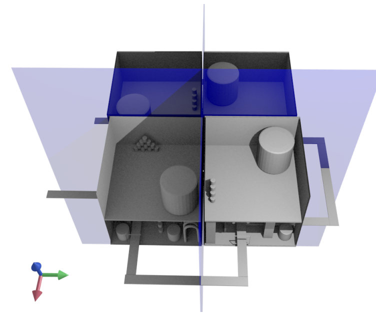


Figure 1: The input mesh has been split in four tiles.

1.4 Method overview

Our method does not impose any requirement on the input triangles, and can be applied without any preprocessing. It ensures the following properties:

1. No assumption is made on the quality of the input mesh.
2. All the vertices of the *Navigation Mesh* are on the boundary. There is no intermediate vertices, introduced by the generation. This is a strong requirement on the quality of the *Navigation Mesh*, allowing to use complex navigation algorithms.
3. It handles large input, with area extending on square kilometers, and containing small details. Scenes can contain tenth of millions of triangles.
4. It does not required too much memory.
5. It is parallelized to ensure good performances.

To guarantee the first requirement, we choose to transform the input mesh in a voxel grid. This transformation can be processed on any input mesh. However, depending on the geometrical scene dimension, and the voxels size, it can use a large amount of memory.

To satisfy points 3 and 4, *i.e.* to support large input and consume a reasonable amount of memory we propose to handle the voxelization tile by tile. See Figure 1. This tile by tile processing also provides good performances on multi core architectures as its parallelisation is straightforward.

However, it introduces an important pitfall : point 2 states that we don't want any vertices to be artificially introduced in the *Navigation Mesh* by the method. All the vertices of the resulting mesh must belong either to the boundaries or to the obstacles. But the tiling process can add artificial vertices on the tiles boundaries. We propose to carefully merge the tiles to avoid this problem.

This method boils down to six steps :

1. Filtering of the non navigable triangles of the input mesh
2. Voxelization of the filtered triangles
3. Tile based segmentation
4. Tile based contours extraction
5. Tiles merging region by region
6. Navigation graph building

The two first steps are detailed in section 2. The segmentation is described in section 3, while contour extraction is explained in section 4.1 and 4.2. Section 4.3 presents tile merging and section 5 graph building. Finally, results are discussed in Section 6.

2 VOXELIZATION

Most applications generating *Navigation Mesh* use as an input a geometry described by triangles. The quality of this triangulation depends on various factors: the geometric modeler, the designer, the triangulation process, the storage format, ... As a result, there is no guarantee on the topological properties of the input mesh.

In paper [10], a mesh cleaner is used to ensure these properties, but that implies the use of external libraries, and can be computationally intensive. And for very complex scenes, this cleaning can be intractable.

For these reasons, we design our navigation mesh generation method such that the input mesh doesn't have to satisfy any assumption. We don't perform any preprocessing on the mesh, and just need a list of triangles.

2.1 Tiling process

Instead, as Bandi [1] has proposed we voxelize the input mesh. However to ensure large input meshes processing, we work tile by tile, as shown in Figure 1. This limits the memory requirement, allows parallelization and authorizes generation of *Navigation Meshes* spreading on square kilometres with centimetric precision.

The tile size in $voxels^2$ can be set arbitrary, depending on the memory available on the computer and the algorithm implementation. In our applications, we usually choose a square tile of size between 256×256 and 1024×1024 $voxels^2$.

Note that splitting the input mesh in tiles creates artificial tiles boundaries that add artificial vertices which do not lies on real boundaries or obstacles. See Figure 6. This prevents the use of algorithms for planning with arbitrary clearance described in paper [8], as these algorithms require that the vertices of the triangulation belong to the obstacles.

Section 4.3 explains how to efficiently and robustly merge the tiles to remove these artificial points.

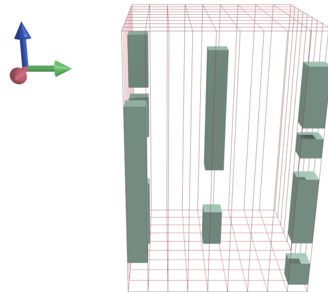


Figure 2: The vertical spans stored in a 2D grid.

2.2 Tiled Voxelization

To further improve the memory use, we don't use a 3D grid, but 2.5D grid, as in paper [13]. *I.e.* we use a 2D uniform grid in the X-Y plan, and each cell contains a list of vertical spans. See Figure 2. Each tile has its own 2.5D grid, whose each cell contains a list of spans.

Span have the following attributes:

- z_{min} z_{max} : min and max discrete altitude of the voxels span.
- l : a label identifying the associated region. Initially unset.

Note that z_{max}, the voxels span upper boundary might be a potential floor, whereas z_{min} might be a potential ceiling. The label l is originally undefined and is set in section 3, during the segmentation phase.

2.3 Filtering

2.3.1 *Slope filtering*

Before applying the tile based voxelization, it is interesting to filter the input to remove triangles than aren't navigable. For instance a man in a wheel chair cannot roll on a plan whose slope exceed 25%.

2.3.2 *Tagging non navigable vertical spans*

It is also important to tag non navigable vertical spans. For instance, the first $span_1$ of two consecutive spans with the same x,y coordinates must be tagged as non navigable if the distance $\Delta z = z_{min,span_2} - z_{max,span_1}$ is less than a robot height.

3 SEGMENTATION

Once the 2.5D voxelization has been performed, we segment the vertical spans in connected regions. The vertical spans coordinates x-y are integers as well as their minimal and maximal $z_{min,max}$. This is important as it eases the segmentation process since we don't have to bother with additional epsilon parameters.

This segmentation is done on a per tile basis, and we choose to perform it slice by slice. This could introduce discontinuities in the regions, depending on the height of the vertical span, but these discontinuities can be easily removed as shown in section 5.2.

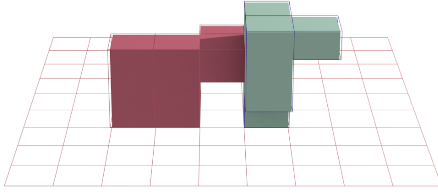


Figure 3: The green vertical spans are connected. They are neighbours and all have the same z_{max}. Some of the red vertical spans have green vertical spans as neighbours, but they don't have the same z_{max}, so they are not connected.

3.1 Flood fill

We use a very simple region growing algorithm to perform the segmentation. Mononen [13] uses the watershed based method proposed in paper [4], but we found this method to introduce over-segmentation in this context.

3.1.1 *Connected vertical spans*

Before proceeding to the segmentation itself, we must define the criteria satisfied by two vertical spans to be considered as connected:

1. They must have the same discrete z_{max}.
2. They must be neighbours, *i.e.* a connected vertical span must be one of the eight immediate neighbours of a vertical span.

These properties are summarized in Figure 3.

3.1.2 *Segmentation algorithm*

Knowing how to decide whether two vertical spans belong to the same zone, we can start the segmentation. The input of our algorithm is the 2.5D grid of vertical spans for the current tile. We use a flood fill. The basic idea of region growing algorithm is to start with a vertical spans, and to check whether his neighbours are connected. If this is the case, they belong to the same region and get the same label. The region growing is then continued using these connected neighbours.

We describe this simple algorithm in Algorithm 1 and 2.

Once the algorithm terminates, the list Z_s has been filled with lists of spans that are interconnected according to our criteria, and we are ready for extracting the contours of these regions.

Algorithm 1 Region growing based segmentation

Require: S filled with all the vertical spans of the current tile
1: **while** stack S is not empty **do**
2: Pop vertical span v from the stack S
3: Set a new label l_{new} to v
4: Add an empty new list L_v of regions in Z_s.
5: Add v to this new list L_v.
6: Push v to Z_c
7: **while** Z_c is not empty **do**
8: Pop vertical span v_i from Z_c
9: call handleNeighbours(v_i, Z_c, L_v)
10: **end while**
11: **end while**

Algorithm 2 Handle vertical span neighbours depending on their connexion

1: handleNeighbours(v_i, Z_c, L_v)
2: **for** each neighbour cells c_n of v_i of the 2.5D grid **do**
3: **for** each vertical span v_n of c_n **do**
4: **if** v_n and v_i are connected **then**
5: Set v_n label as l_{new}
6: Add v_n to list L_v.
7: Push v_n into stack Z_c
8: **end if**
9: **end for**
10: **end for**

4 MERGING TILE REGION

4.1 Corner Points Extraction

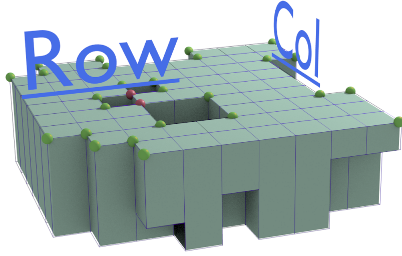
4.1.1 *Identifying contour edges*

The Figure 4 represents a typical region, with complex boundaries, holes, and connected holes. Contours connect the vertices stressed by the small spheres. These vertices can be easily identified by looking at the vertical spans connectivity. Let's call these vertices *corner points*. When looking at this region, two observations can be made:

1. If you browse contours rows by rows and columns by columns, it appears that there is an alternation in the connection between the *corner points*. Hence the first corner point of a row always connect to the second one ; the second one is never connected to the third ; the third always connected to the fourth ; ... This is in fact an application of the Jordan Theorem.
2. However, there is an exception to this rule: vertices shared by two holes. This is the case for the two red spheres. Let's call these vertices *slash* and *antislash* points depending on the holes position. See Figure 4.

4.1.2 Slash *and* Antislash *Points*

The third observation can be easily circumvented by identifying the *slash* and *antislash* and by duplicated them. Once duplicated, the second observation is always true. See Figure 7.



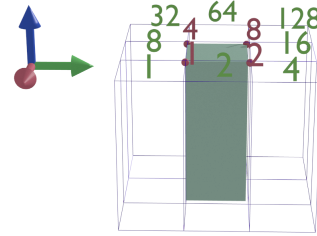


Figure 4: *Corner points* extraction. The upper red sphere is a *slash* point, as its surrounding holes form a *slash* diagonal. The lower red sphere is an *antislash* point, as its surrounding holes form an *antislash* diagonal.

Figure 5: The 4 vertices of the vertical span floor are labelled 1,2,4 and 8, whereas the neighbour vertical spans are labelled 1, 2, 4, 8, 16, 32 and 128. Hence, the neighbour configuration of a vertical span can be encoded on 8 bits *i.e.* a unsigned char.

4.1.3 Corners Points Extraction

Corner points are defined by their connectivity. As shown in Figure 5, the floor of each vertical span has 4 vertices, and these vertices are *corner points* if they are not surrounded by 4 vertical spans belonging to the same region. We define the neighbour configuration of a vertical span as the integer n_c on 8 bits encoding its neighbours.

For instance, a vertical span with no neighbours vertical spans has $n_c = 0$. A vertical span surrounded by vertical spans has $n_c = 1+2+4+8+16+32+64+128 = 255$. And a vertical span neighbours of vertical span 8 and 16 has $n_c = 8+16$.

Knowing this configuration, it is straightforward to identify the *corner points* using the following mask:

1. point 1 is corner if : $n_c\&11 \notin \{11,8,2\}$
2. point 2 is corner if : $n_c\&22 \notin \{22,2,16\}$
3. point 4 is corner if : $n_c\&104 \notin \{104,8,64\}$
4. point 8 is corner if : $n_c\&208 \notin \{208,16,64\}$

Slash and *antislash* points can also be detected similarly. These kind of points are *corner points*, and must pass the tests above as well as:

1. point 1 is a *antislash* point if : $n_c\&11 = 1$
2. point 2 is *slash* point if : $n_c\&22 = 4$
3. point 4 is *slash* point if : $n_c\&104 = 32$
4. point 8 is *antislash* point if : $n_c\&208 = 128$

Hence, to identify corners points, for each region of a tile, we iterate through all the spans, compute their neighbour configuration, apply the tests above, and store them region by region in L_c.

4.1.4 Handling vertices on the tile boundaries

The input mesh being processed tiles by tiles, some navigable regions lying on more than one tile are artificially cut into multiple regions. Regions merging is then required, and is described in section 4.3.

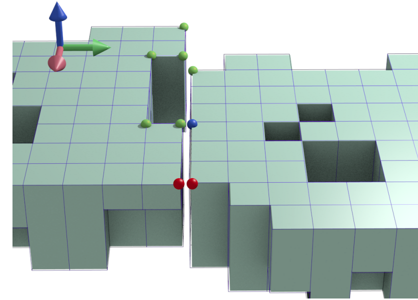


Figure 6: Handling tiles boundaries. Green and red spheres indicate corners point detected using the method above. Red ones must be removed as they are artificially added by tiling. Blue ones are added by taking into account neighbour tiles border vertical spans to ensure continuity.

However, to merge regions, we need to ensure some continuity on adjacent tiles boundaries. It is also important to distinguish vertices that are really *corner points* from vertices that are artificially labelled *corner points* due to the tiling process. See Figure 6.

Borders *corner points* can be classified in three categories :

1. Real *corner points*. These are the green spheres of Figure 6.
2. *corner points* if adjacent tiles are taken into account. These are the blue spheres of Figure 6.
3. False *corner points* that wouldn't exist without the tiling. These are the red spheres of Figure 6.

There is nothing special to do with the first class of points.

For the second and third classes of points, we must apply the method above on tile extended by a band of one vertical span on each of its four boundaries. Points of the second class will be detected by this operation, while points of the third class will be removed. It is then straightforward to identify these points by comparison with the *corner points* of the unextended tile.

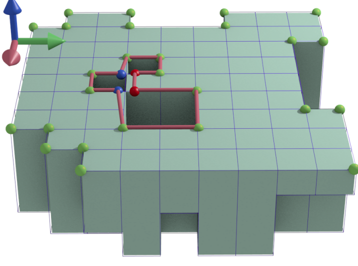


Figure 7: Contour extraction for duplicated *corner points*. Blue point is the first duplicate, while red one is the second. Vertical connection follow the rules of Table 1.

4.2 Contour Extraction

Once we have identified the *corner points* of the region extracted during the segmentation, we proceed to the extraction of the contour. This extraction is done region by region. Remember that all the vertices in a region have the same z coordinate.

Our algorithm use two main structures:

- A list *rows* of corner vertices, stored rows by rows
- A list *cols* of corner vertices, stored columns by columns

The idea behind this algorithm is to exploit the fact that we are using discrete points in conjunction with the *even-odd rule* given by the Jordan theorem. We process the vertices row by row, and col by col, and add them in the *rows* and *cols* structures. Special *slash* and *antislash* points are duplicated. See Algorithm 3.

Algorithm 3 Fill *rows* and *cols* lists.

Require: L_c: the list of *corner points* stored region by region
1: **for** each region r **do**
2:     **for** each corner point $s(x,y)$ in r **do**
3:         **if** $s(x,y)$ is a *slash* or *antislash* point **then**
4:             Add s twice to $rows[y]$.
5:             Add s twice to $cols[x]$
6:         **else**
7:             Add s to $rows[y]$
8:             Add s to $cols[x]$
9:         **end if**
10:     **end for**
11: **end for**

first \ second	*normal*	*slash*	*antislash*
normal	$p \to p$	$p \to p_2$	$p \to p_1$
slash	$p_1 \to p$		$p_1 \to p_1$
antislash	$p_2 \to p$	$p_2 \to p_2$	

Table 1: Vertical connection between corner point depending on their type : *normal, slash, antislash*. p_1 and p_2 stands respectively for the first and second duplicate of *slash* or *antislash* points.

Then, we connect vertices by pair, horizontally, and vertically, using the *even-odd rule*, with special attention for duplicated points. See Algorithm 4 and Table 1. As a result we get $\mathcal{N}_h$ and $\mathcal{N}_v$ that give respectively the horizontal and vertical neighbour of each vertices $v \in L_c$.

Algorithm 4 Build contour graph

Require: *rows* and *cols*
1: **for** each *row* of *rows* **do**
2:     Sort *corner points* of *row* in increasing x order.
3: **end for**
4: **for** each *col* of *cols* **do**
5:     Sort *corner points* of *col* in increasing y order.
6: **end for**
7: **for** each *row* of *rows* **do**
8:     **for** each $n-1$ *corner points* s_i in *row* **do**
9:         **if** i is even **then**
10:             Set $\mathcal{N}_h(s_i) = s_{i+1}$
11:             Set $\mathcal{N}_h(s_{i+1}) = s_i$
12:         **end if**
13:     **end for**
14: **end for**
15: **for** each *col* of *cols* **do**
16:     **for** each $n-1$ *corner points* s_i in *col* **do**
17:         **if** i is even **then**
18:             **if** s_i and s_{i+1} are not a *slash* neither a *antislash* points **then**
19:                 Set $\mathcal{N}_v(s_i) = s_{i+1}$
20:                 Set $\mathcal{N}_v(s_{i+1}) = s_i$
21:             **else**
22:                 set s_i neighbour according to Table 1
23:             **end if**
24:         **end if**
25:     **end for**
26: **end for**

Finally, we connect all the vertices together to get all the contours of the region, including holes contour, and we store them in *conts*. See Algorithm 5.

Algorithm 5 Connect all contours vertices.

Require: L_c, $\mathcal{N}_v$ and $\mathcal{N}_h$
Ensure: The list of contours *conts*
1: **while** all vertices in L_c have not been connected **do**
2:     Pick a non connected vertex s in L_c
3:     Create a new empty contour *cont*, *i.e.* an empty list of vertices
4:     Add s into *cont*.
5:     Let s_c be the vertical neighbour of s
6:     **while** $s_c \neq s$ **do**
7:         Add s_c into *cont*
8:         **if** $\mathcal{N}_h(s_c)$ already belongs to a contour **then**
9:             Set $s_c = \mathcal{N}_v(s_c)$
10:         **else**
11:             Set $s_c = \mathcal{N}_h(s_c)$
12:         **end if**
13:     **end while**
14:     Add *cont* to *conts*
15: **end while**

4.3 Regions merging

The algorithms used to extract the contours are performed tiles by tiles. This creates two difficulties :

- It artificially splits regions : as shown in Figure 6, if a navigable region of the input mesh lies on two or more tiles, it is cut in two or more unconnected regions.
- It creates artificial points. They augment the complexity of the resulting navigation mesh and preclude the possibility of using path planning with arbitrary clearance.

These difficulties can be circumvented by performing inter-tiles regions merging.

4.3.1 Vertices based merging

To reduce the complexity of the merging by one order of magnitude, we don't merge region on a vertical span basis, but on a border vertices basis. We proceed using an incremental approach. We process the regions of each tile sequentially.

Algorithm 6 Handle vertical spans neighbours depending on their connexion

Require: That each tile regions and contours have been computed
1: **for** each tile t **do**
2:     **for** each region r of t **do**
3:         Get the contour c of r
4:         Get the vertices L_{v_3} of third class in the contour c
5:         **if** some regions L_{sr} of L_r have some points in L_{v_3} **then**
6:             Merge the regions in L_{sr} in a unique region r_u
7:             Add the region r_u to L_r
8:         **else**
9:             Add the region r to L_r
10:         **end if**
11:     **end for**
12: **end for**
13: **for** each region r of L_r **do**
14:     Remove the useless vertices of the third class
15: **end for**

First, we store the vertices artificially added by the tilling, those of the third class, in a dedicated array L_{v_3}. These vertices will be removed. If there is no such points, we create a new region, fill it with all the vertices and contour information, and add it to L_r.

Otherwise, we test these vertices, which by construction are shared with other regions, against the vertices of the region already stored in L_r. Again, if there is no such region, we create a new region, add all the vertices and contour information into, and add it to L_r. Otherwise, this gives us the list L_{sr} of regions whose contours share these vertices. We merge the regions in L_{rs} into a unique region r_u and add it to L_r. The regions listed in L_{rs} are removed from L_r.

Once all the regions of all the tiles have been processed this way, we iterate through the resulting regions, and remove the points of the third category.

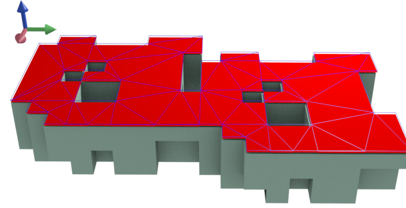


Figure 8: The triangulation for a unique region originally spreading on two different tiles.

See Algorithm 6 for a complete overview.

After this step, we have a list of regions defined by their vertices and their associated contours, and these regions are completely free from any artifact coming from the tiling process.

5 BUILDING THE NAVIGATION GRAPH

5.1 Triangulation

At this step, we have a list of unconnected and independent regions, defined by their contour, and which might contains holes. In order to display them, and apply our path planning algorithm, they must be triangulated.

This step is quite straightforward. We just have to identify the external contours, which surround the region, and the internal contours, which surround the potential holes.

5.1.1 Labelling contours

We sort the edges list $L_e(r)$ of each region r such that the first edge e_0 in $L_e(r)$ contains the bottom left vertex of the region contours.

Hence, using the Algorithm 7, we label the contours of the region so that the contour with label 0 is the external contour. All the others contours are internal contours surrounding holes.

5.1.2 Identifying holes edges

The Algorithm 7 also stores the bottom left vertex v_s for each contour. These vertices are used during the triangulation as seed to remove triangles inside holes. Figure 8 shows an example of region triangulation with holes.

5.1.3 Convex Partitioning

Finally, we use the simple convex partitioning algorithm presented in paper [6] to simplify our mesh by regrouping triangles into convex polygons. The method proposed in paper [15] can be used for a more optimal convex partitioning.

Algorithm 7 Label contours of region r

Require: $L_e(r)$ filled with all the edges of the current region
1: Fill the stack S_e with the edges of $L_e(r)$
2: Set the current contour label $l_c = 0$
3: **while** S_e is not empty **do**
4: Pop an edge e from the stack S_e
5: Let a_e, b_e be the vertices of e
6: Set a_e and b_e contour label to l_c
7: Set the bottom left vertices of current contour $\mathscr{L}_{ower}(l_c) = a_e$
8: Create an empty stack S_i
9: Find the edge e_a sharing a_e with e
10: **if** there is such a e_a edge **then**
11: Remove e_a from S_e
12: Push e_a into S_i
13: **end if**
14: Find the edge e_b sharing b_e with e
15: **if** there is such a e_b edge **then**
16: Remove e_b from S_e
17: Push e_b into S_i
18: **end if**
19: **while** S_i is not empty **do**
20: Pop edge e_i from S_i
21: Let a_{e_i}, b_{e_i} be the vertices of e_i
22: Set a_{e_i} and b_{e_i} contour label to l_c
23: Update $\mathscr{L}_{ower}(l_c) = a_e$ according to vertices a_{e_i}, b_{e_i} positions
24: Find the edge $e_{a_{e_i}}$ sharing a_{e_i} with e_i
25: **if** there is such a $e_{a_{e_i}}$ edge **then**
26: Remove $e_{a_{e_i}}$ from S_e
27: Push $e_{a_{e_i}}$ into S_i
28: **end if**
29: Find the edge $e_{b_{e_i}}$ sharing b_{e_i} with e_i
30: **if** there is such a $e_{b_{e_i}}$ edge **then**
31: Remove $e_{b_{e_i}}$ from S_e
32: Push $e_{b_{e_i}}$ into S_i
33: **end if**
34: **end while**
35: Increment current contour label $l_c = l_c + 1$
36: **end while**

This algorithm is applied to all the regions detected in the previous step.

5.2 Connection Graph

After the partitioning, our input mesh has been transformed into a list of unconnected regions. To enable navigation between these regions, we must build a navigation graph linking the regions, to allow navigation on the whole navigation mesh

This navigation graph will be used for path planning using the A* algorithm [3].

5.2.1 *Reconnect discontinuities due to discretization*

Due to the voxelization process, some regions that were originally connected might have been artificially disconnected depending on the voxel height. These regions must be reconnected.

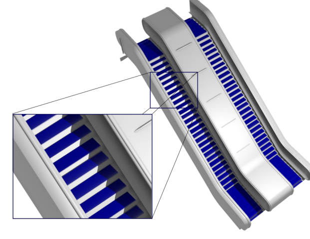


Figure 9: The blue polygons are the *Navigation Mesh* generated for this escalator.

Two regions that have been disconnected by the voxelization must have edges in common. The vertices of these edges have the same x, y discrete coordinates, and their discrete z coordinate must differ by only 1 voxel height.

Hence, to reconnect these regions, we simply add a link in the navigation graph that connects two polygons of two different regions which have each an edge satisfying the constraint above.

5.2.2 *Stairs handling*

Handling stairs is quite similar to disconnected region reconnection. See Figure 9. Two regions r_a and r_b are identified as connected stairs if there exists two edges $e_a \in r_a$ and $e_b \in r_b$ such that :

- $\Delta z < max_{climb}$
- $||v1^a_{x,y} - v1^b_{x,y}|| < r$ and $||v2^a_{x,y} - v2^b_{x,y}|| < r$

Where $v1^a$ and $v2^a$ are the two vertices of e_a, $v1^b$ and $v2^b$ are the two vertices of e_b, and Δz is the difference between the z coordinates of e_a and e_b. Note that $v1$ and $v2$ have the same z coordinate when they belongs to the same region.

A link is added in the navigation graph for each two polygons whose one of the edges satisfy these requirements.

6 RESULTS

6.1 Settings used

Our method uses only three parameters :

- The voxel dimensions: c_x, c_y, c_z
- The maximum height of a step: max_{climb}
- The tile size in voxels: n_{vx}

To evaluate the robustness of our method, we choose 5 different settings for these parameters. See Table 2.

The default set uses sensible parameters, that allow a good precision while keeping the number of voxels at a reasonable level.

The setting S2 use small voxels, with exotic dimensions.

Table 2: The settings used to test the robustness of our method

Set.	c_x	c_y	c_z	max_{climb}	$n_v x$
def.	0.05	0.05	0.02	0.5	256
S2	0.0356	0.0356	0.02	0.5	256
S3	0.0666	0.0666	0.5010	0.5	256
S4	0.666	0.666	0.587	0.5	256
S5	0.05	0.05	0.02	0.5	467

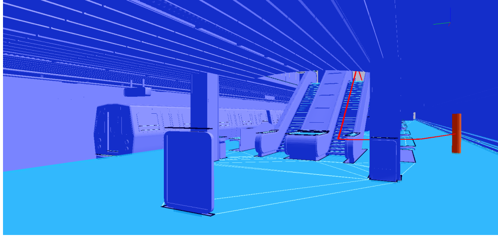


Figure 10: The subway station using the default settings. The navigation mesh is represented in light blue. An example of path found using the escalator is shown by the red line. The path planning query took 1.329ms.

Table 3: Results for the subway station

Set.	#polys	#voxels	#links	time (s)
def.	12 288	7021 * 804 * 415	18 088	14.122
S2	11 550	9861 * 1129 * 443	24 358	26.035
S3	2342	5271 * 603 * 17	4 334	5.314
S4	286	527 * 60 * 14	650	0.726
S5	11 188	8776 * 1005 * 553	23 916	19.79

The setting S3 use high voxels with a tight basis, also with exotic dimensions. Moreover, the voxels dimension c_z is bigger than the value max_{climb}. This means that it will be impossible to detect steps.

The setting S4 use big voxels, whose dimension c_z is bigger than the value max_{climb}. This setting should create coarser navigation mesh, with less polygons, but in a shorter time, as less vertical spans are created.

Finally, the setting S5 just changes the size of the tiles. This should just change the performances.

6.2 Navigation Mesh generation results

The 5 settings presented above have been tested on various meshes. We choose to present the results for 3 representative meshes. See Figure 10, 11 and 12.

A coherent navigation mesh have been generated for each pair settings/3D scene.

6.2.1 Subway station

This scene has 568 418 triangles. The results of the navigation meshes generation are summarized in Table 3.

The navigation meshes have been generated in a short time for all settings. The resulting navigation meshes are very simple, and contain a small number of polygons in comparison with the original 568 418 triangles.

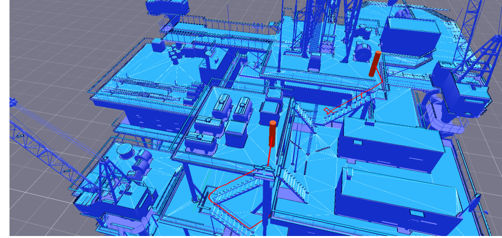


Figure 11: The platform using the default settings. This is a complex scene on multiple levels, with many details : pipe, stairs, cranes, ... An example of path found on multiple level using the stairs is shown by the red line. The path planning query took 4.836ms.

Table 4: Results for the platform.

Set.	#polys	#voxels	#links	time (s)
def.	21 331	800 * 1045 * 2404	41 864	9.727
S2	32 976	1124 * 1468 * 2572	65 488	12.586
S3	9 511	601 * 785 * 96	18 834	3.339
S4	537	60 * 78 * 82	972	0.097
S5	30 424	1000 * 1307 * 3206	59 616	15.812

The results are coherent with the parameters. Smaller voxels leads to more detailed navigation meshes, with more polygons. Using smaller voxels also increased the generation time.

As shown in figure 10, the escalator have been detected with the defaults parameters and the settings S2 and S5. The detection of stairs allows to use walking algorithms like the one described in paper [5], which was impossible in *Recast* [13].

6.2.2 Platform

The scene platform has 119 601 triangles. The results of the navigation meshes generation are summarized in Table 4.

This scene is a complex one, with multiples interconnected levels. There is a lot of details, with many pipes, stairs, cables, cranes, ... There are also intersecting triangles so the method proposed by Lamarche [10] would not work on this 3D scene without prior cleaning. Cleaning a complex scene like this one can be tedious and remove important details.

Moreover, in paper [10], the *House* scene has a complexity similar to this *Platform* scene, *i.e.* 120 160 triangles, and they create the navigation mesh in 904.32s. Our method is about 10 times faster.

The stairs have been detected with the defaults parameters and the settings S2 and S5.

6.2.3 Buildings

The scene building has 1 121 127 triangles. See Table 5 for all results.

This scene is a not very complex, but is very large. Its dimensions are about $500 * 500 * 16m^3$. This lead to

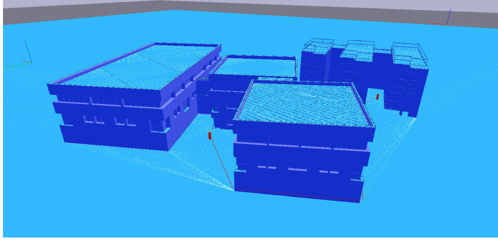


Figure 12: The buildings scenes using the default settings. It's a very large scene. The path planning query took 5.581ms.

Table 5: Results for the buildings.

Set.	#polys	#voxels	#links	time (s)
def.	117 488	8463 * 10977 * 804	235 964	229.854
S2	180 173	11886 * 15417 * 860	364 132	612.833
S3	54 139	6353 * 8241 * 32	107 122	63.863
S4	2 622	635 * 824 * 27	4 980	1.591
S5	164 577	10579 * 13721 * 1073	333 554	400.257

a huge number of voxels when using small voxels dimension. Using the default parameters, we manage to create the navigation mesh in less than 4 minutes. Pure voxels method like the one detailed in pape [1] can't handle scene large as this one with a resolution as low as 0.05*0.05*0.02 m^3 as the number of voxels make it impossible to compute the path planning in a reasonable time. The memory requirement, when no tiling is used, would also prevent the use of the method proposed in paper [1].

Our method successfully creates the navigation meshes for each settings, and we were able to handle path planning request in less than 5ms.

7 CONCLUSION

In this paper we have detailed all the necessary steps to robustly generate *Navigation Meshes* from any triangles scene. Our method support very large scenes, with stairs, and uneven terrains. Absolutely no assumption is made on the quality of the input mesh. The resulting *Navigation Meshes* support path planning with arbitrary clearance. Moreover, the performances are excellent, as we are about 10 time faster than the method presented in paper [10].

We have applied this method with success in various context: serious gaming, security, virtual human walk, robot control, or crowd simulation. We plan to further improve our algorithm by supporting Multi Resolution Analysis, and by automatically setting the voxels size depending on the level of details in each tile. We will also explore hierarchical representation of uneven terrains in order to reduce navigation graph complexity.

8 REFERENCES

[1] Srikanth Bandi and Daniel Thalmann. Space discretization for efficient human navigation. *Computer Graphics Forum*, 17(3):195–206, 1998.

[2] Elmar Eisemann and Xavier Décoret. Single-pass GPU Solid Voxelization and Applications. In *GI '08: Proceedings of Graphics Interface 2008*, volume 322, pages 73–80, Windsor, ONT, Canada, 2008. Canadian Information Processing Society.

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

[4] Denis Haumont, Olivier Debeir, and François X. Sillion. Volumetric Cell-and-Portal Generation. In *Computer Graphics Forum*, volume 3-22 of *EUROGRAPHICS Conference Proceedings*, pages 303 – 312, Grenade, Espagne, 2003. Blackwell Publishers.

[5] Andrei Herdt, Holger Diedam, Pierre-Brice Wieber, Dimitar Dimitrov, Katja Mombaur, and Moritz Diehl. Online Walking Motion Generation with Automatic Foot Step Placement. *Advanced Robotics -Utrecht-*, 24(5-6):719–737, 2010.

[6] Stefan Hertel and Kurt Mehlhorn. Fast triangulation of simple polygons. In Marek Karpinski, editor, *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 207–218. Springer Berlin Heidelberg, 1983.

[7] Marcelo Kallmann. Navigation Queries from Triangular Meshes Motion in Games. volume 6459 of *Lecture Notes in Computer Science*, chapter 22, pages 230–241. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

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

[9] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566 –580, aug 1996.

[10] Fabrice Lamarche. Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. *Comput. Graph. Forum*, pages 649–658, 2009.

[11] Duoduo Liao. Gpu-accelerated multi-valued solid voxelization by slice functions in real time. In *Proceedings of the 24th Spring Conference on Computer Graphics*, SCCG '08, pages 113–120, New York, NY, USA, 2010. ACM.

[12] Ignacio Llamas. Real-time voxelization of triangle meshes on the gpu. In *ACM SIGGRAPH 2007 sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[13] Mikko Mononen. Recast navigation, May 2012.

[14] D. Nieuwenhuisen, A. Kamphuis, and M. H. Overmars. High quality navigation in computer games. *Sci. Comput. Program.*, 67(1):91–104, June 2007.

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

[16] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. *Game Programming Gems*, pages 288–304, 2000.

[17] Wouter van Toll, Atlas F. Cook IV, and Roland Geraerts. A navigation mesh for dynamic environments. *Journal of Visualization and Computer Animation*, 23(6):535–546, 2012.