NEOGEN: Near Optimal Generator of Navigation Meshes for 3D Multi-Layered Environments.

R. Oliva, N. Pelechano

Universitat Politcnica de Catalunya

Abstract

In this paper we introduce a novel automatic method for generating near optimal navigation meshes from a 3D multi-layered virtual environment. Firstly, a GPU voxelization of the entire scene is calculated in order to identify and extract the different walkable layers. Secondly, a high resolution render is performed with a fragment shader to obtain the 2D floor plan of each layer. Finally, a convex decomposition of each layer is calculated and layers are linked in order to create a Navigation Mesh of the scene. Results show that our method is not only faster than previous work, but also creates more accurate NavMeshes since it respects the original shape of the static geometry. It also provides a significantly lower number of cells and avoids ill-conditioned cells and T-Joints between portals that could lead to unnatural character navigation.

Keywords: Cell-and-Portal Graphs, path planning.

1. Introduction

Character navigation in complex scenes is commonly performed by having a Navigation Mesh (NavMesh), which encodes a convex decomposition of the scene. The NavMesh is represented with a Cell-and-Portal Graph (CPG), where cells are convex regions and portals are the edges shared by convex regions that a character can traverse. Path finding algorithms such as A* are then used over those NavMeshes.

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

Previous work is either not fully automatic, cannot handle any geometry, and/or provides CPGs with far too many cells or ill-conditioned cells (a cell where vertices are practically collinear which occurs when any of the internal angles is close to 0, or when the ratio Area/Perimeter of the polygon is close to 0). An over-segmented partition has two main problems. Firstly, the performance of the path finding algorithm directly depends on the dimensions of the generated graph, so the fewer cells we have, the faster this step will be. Secondly, depending on the underlying local movement algorithm being used, an over-segmented partition may end up with characters walking in zig-zags through a long convex space as they are forced to go through unnecessary portals, or in portals so close together that they add too many unnecessary nearby attractors and therefore complexity when trying to achieve natural looking local movement.

The architecture presented in this paper is novel, as it overcomes all limitations described above and presents an entire pipeline to go automatically from a 3D multilayer environment given as a polygon soup, to the final navigation mesh which adjusts tightly to the original geometry.

The main contribution of this paper is a novel GPU based method to generate a CPG for a given 3D scene (with slopes, steps and other obstacles). The algorithm starts by performing a GPU voxelization of the geometry to classify the different layers and calculate a *cutting shape*, *CS*. The *CS* is a depth filter used by the fragment shader to flatten each layers' geometry into a 2D high resolution texture encoding the depth map of each layer. Then each layer is encoded as a single simple polygon with holes which is the required input for the NavMesh generator [1]. The NavMesh generator provides a convex decomposition with a number of cells close to the optimal value, since for most cases it only needs to create one portal per notch of the polygon with holes (see proof in the appendix). The resulting CPGs are further optimized with a novel convexity relaxation technique which further reduces the number of cells. Finally all the layers' CPG are automatically linked together to obtain the final CPG of the entire scene. Figure 1 shows the flow of the algorithm.

To the best of our knowledge this is the first fully automatic architecture that can provide an accurate navigation mesh, with an almost optimal number of cells, ready for path planning from just a polygon soup (with degeneracies such as intersections and holes). The presented system is efficient enough to allow the scene modeler to make changes and observe the impact on the final navigation mesh at interactive rates and is of high importance to the fields of video games, robotics, movies and character simulation.

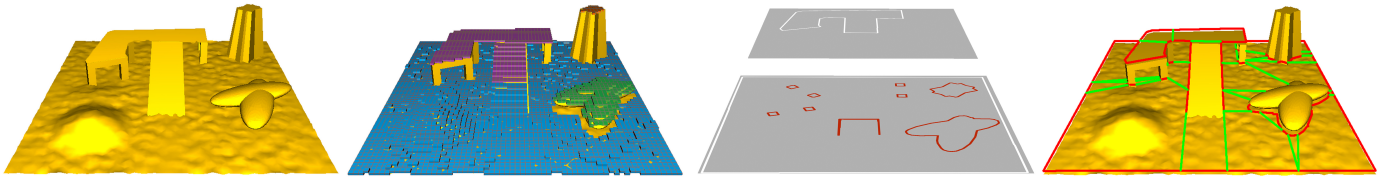

Figure 1: Near optimal navigation mesh construction for a simple example of a 3D environment with 2 layers. From left to right we can see the original scene, the result of the layer extraction step after the coarse voxelization, the 2D floor plan of each layer, and finally the near optimal navigation mesh.

This paper is organized as follows: In the next section we discuss previous work on Navigation Meshes. The third section describes the details of our method, explaining step by step the algorithm to go from the initial polygon soup to the CPG representing the navigation mesh of the entire scene. Next, results are shown followed by conclusions and future work.

2. Related Work

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

The Roadmap approach [2, 3, 4, 5] captures the connectivity of the free space by using a network of standardized paths (lines, curves). The main limitation of this representation is that it only contains information about which locations of the scene are directly connected, but it does not describes the geometry of the scene, nor where the obstacles are. Consequently, avoidance of dynamic obstacles is usually a hard task and not always possible, as exposed in [4].

The Cell Decomposition approach consists of the partition of the navigable geometry of the scene into convex regions, guaranteeing that a character can move from two points on the same cell following a straight line, without getting stuck in local minima. This particular decomposition is usually known as Navigation Mesh (NavMesh) [6], and a CPG can be obtained to compute paths free of obstacles. Collisions with movable obstacles such as other agents are solved by using a local movement algorithm [7, 8], or by dynamically modifying the NavMesh [9].

Local movement algorithms are generally driven by setting way points within the portals of the NavMesh that work as attractors to steer the agents in the right direction. An improvement to traditional way points was introduced by Curtis et. al. [10] by using way portals where the whole length of the portal can be used to attract the local movement of the agents, thus resulting in more natural looking paths.

Uniform grids have been proposed as a cell decomposition of the environment [11, 12]. It is an easy and fast solution to obtain a convex decomposition, but the main problem of this technique is that a high-density CPG is generated, increasing the time of the search algorithm. Another limitation is the poor adjustment to the shape of the obstacles, thus losing an important part of the walkable space. Valve's Game Engine follows a similar approach [13]. It subdivides the virtual map by Axis Aligned (AA) quads of arbitrary size. The resulting partition contains fewer cells than in the case of uniform grids, but is still far from the optimal as it is a partition restricted by AA Quads. For the same reason, the adjustment to the obstacles remains poor. Hale et. al. [14], presented an automatic NavMesh generator method that consists of spreading a certain number of unitary quad seeds in the scene. Those quads are expanded as much as possible, adjusting to the contour of the obstacles and generating new seeds to completely cover the walkable space. Although the adjustment to the obstacles is perfect, the partition obtained contains many narrow cells and T-Joints, which could introduce artifacts on the movement of the characters when applying a local-movement method. In addition, the method only works if every obstacle is convex, so a preliminary step to decompose the obstacles into convex parts is required. A volumetric version of this algorithm was proposed in [15], but it has the same limitations as the 2D version.

Triangular Meshes have also been used to represent a Navigation Mesh. In [9, 16], a dynamic Constrained Delaunay Triangulation (CDT), having as constraints the edges of the obstacles, is used to represent the walkable area of a scene. During run-time obstacles are allowed to be inserted, removed or displaced and the CDT is able to dynamically take into account these changes. However, the results show that the performance of the application greatly depends on the complexity of the CDT, as well as on the complexity and number of constraints being moved. The size of the CDT is linear to the input vertices, whereas the convex partition obtained by our approach is linear to the concave vertices and hence, our method scales better. In [17], Kallmann introduced a new type of triangulation called Local Clearance Triangulation (LCT) that allows computing paths free of obstacles with arbitrary clearance. Such triangulation is obtained by a process that iteratively refines the starting CDT of the scene.

TopoPlan [18] is an application that automatically generates a CPG for a given virtual environment defined as a mesh of triangles that can contain multiple layers. The layers are extracted using a prism subdivision, and the CDT of each layer is computed. Although the description of the walkable space is perfect, it is very costly in time. Results show that for a scenario of just 120k triangles, Topoplan needs more than 15 minutes to compute the NavMesh. This work was then extended to identify

outdoor, indoor and covered areas for spacial reasoning [19]. The main advantage of using a partition based on triangles is that geometric operations with triangles are very efficient, the convexity of the partition is guaranteed and it contains the least possible number of ill-conditioned cells. However, the partition is far from optimal as it is restricted by triangles.

Another representation consists of partitioning the space by using Ngons [20, 21, 22, 1]. An example of such an application is the Recast toolkit [22], a popular tool used in complex applications such as videogames and other virtual simulations. The method used by Recast is inspired by the work of Haumont et al. [23]. Firstly, it computes a voxelization of the scene that makes the method robust against degeneracies of the input model (such as interpenetrating geometry, cracks or holes) as well as simplifies the furniture. Then, a partition of the scene is obtained by applying the Watershed Transform (WST) on the Distance Map Field of the voxelized scene. This partition is further refined as the WST does not guarantee convexity of the generated cells. Recast is a robust application, but the description of the walkable space is not totally exact as the adjustment to the contours is not precise. However the most important drawback is the over-segmentation of the scene and the generation of walkable regions where a character cannot access, (which may not be desirable and thus require manual adjustment, or post-processing).

Toll et al. [24] presented a NavMesh generation method for a multi-layered environment, such as an airport or a multi-story car-park, where the different layers of the scene are connected by elements such as stairs or ramps. However, they do not provide an automatic method to extract such layers. The implementation of this NavMesh generation method, restricted to one single layer, can be found in [25].

The work presented by Oliva and Pelechano [1] provided an efficient technique to calculate a convex decomposition with a number of cells close to the optimal convex decomposition. However, it only worked for simple polygons with holes, so in order to use it for a real scene, the polygons would have to be introduced manually, and the CPGs merged together manually.

3. Automatic NavMesh Generator

NEOGEN takes as an input a polygon soup describing a multi-layer 3D environment and provides as an output a CPG that can be directly used for character navigation without any manual work required by the user. The user only needs to specify four input parameters:

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

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

In Figure 1 we can see an example with some intermediate results provided by NEOGEN, and the final CPG. Figure 2 shows the NEOGEN framework to generate a navigation mesh from a 3D multi-layer environment.

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

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

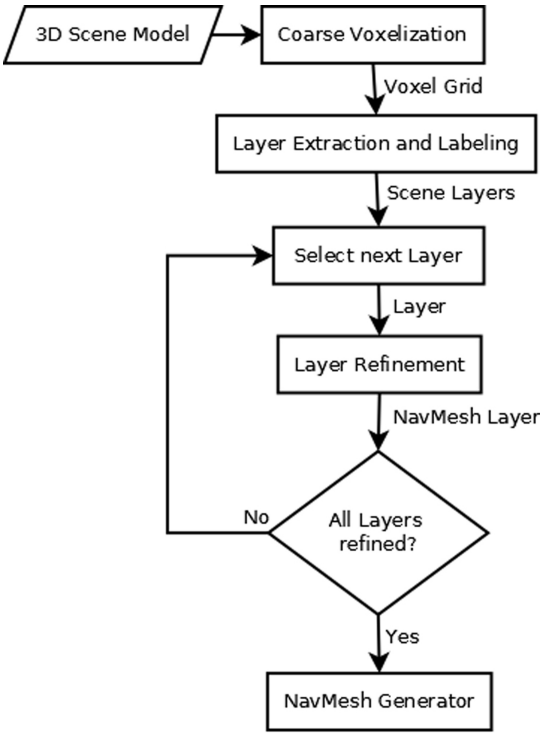

Figure 2: Multilayer framework for the automatic navigation mesh generator.

3.1. GPU coarse voxelization

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

The voxelization method is an extension of the GPU method described in [26], where they perform in just one rendering pass a voxelization based on a slicing method. A grid is defined by placing a camera above the scene and adjusting its view frustum to enclose the area to be voxelized. This camera has an associated viewport with (w, h) dimensions. The scene is then rendered, constructing the voxelization in the frame buffer. A pixel (i, j) represents a column in the grid and each voxel within this column is binary encoded using the k^{th} bit of the RGBA value of the pixel. Therefore, the corresponding image represents a $w \times h \times 32$ grid with one bit of information per voxel. This bit indicates whether a primitive passes through a cell or not. The union of voxels corresponding to the k^{th} bit for all pixels defines a slice. Consequently, the image/texture encoding the grid is called a *slicemap*. When a primitive is rasterized, a set of fragments are obtained. A fragment shader is used in order to determine the position of the fragment in the column based on its depth. The result is then $OR - ed$ with the current value of the frame buffer.

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

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

For each drawn fragment we classify the voxel as:

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

where α_{max} is the maximum walkable angle, $\vec{n}$ is the normal of the polygon corresponding to that fragment and $\vec{UP}$ is the vector $(0, 1, 0)$.

The main problem is that *positive* and *negative* surfaces can fall in the same voxel. Since in our application we are only interested in detecting the voxels with walkable surfaces, we will classify those voxels also as *positive*. The refinement step performed later on will solve the ambiguity. Figure 3 shows the decomposition into positive and negative voxels of a simple scene.

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

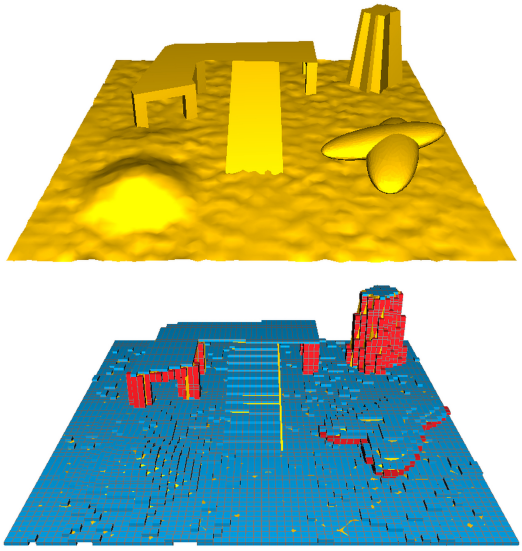

Figure 3: Original scene and its corresponding GPU coarse voxelization. Red indicates negative voxels and blue positive voxels (i.e. where the slope is within the walkable capabilities of the character).

3.2. Layer extraction and labeling

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

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

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

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

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

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

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

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

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

The flooding algorithm proceeds iteratively from bottom to top. Figure 4 shows the result of the layer ex5traction step.

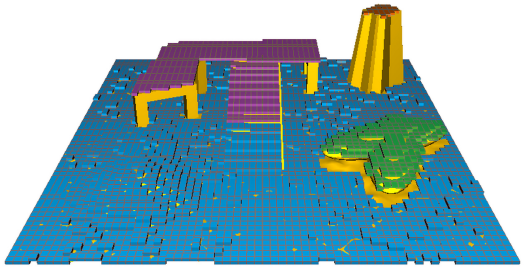

Figure 4: Result of the layer extraction step. Each color indicates the set of voxels belonging to the same layer.

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

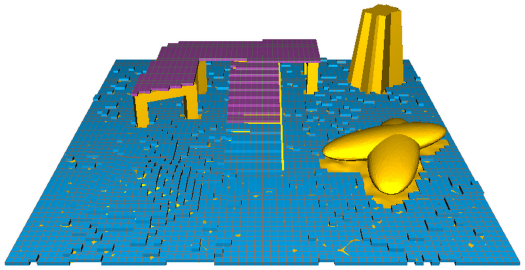

Figure 5: After the layer connectivity step, unreachable layers are eliminated (for this example, the orange and green layers are eliminated.

3.3. Layer refinement

At this stage, we have subdivided the real walkable space into Layers. The Navigation Mesh could be computed from this representation, but since we want to obtain a fine adjustment to the obstacles, we need to further increase the resolution. The goal is to obtain a 2D high resolution floor plan of each layer. This is a key step of our algorithm, since it will solve any ambiguities contained in the voxels due to the coarse voxelization step, and provide as an output an accurate high resolution description of the original geometry. In order to do this we have implemented a fragment shader that for each layer will only render the geometry that corresponds to such layer. Once the fine floor plan is rendered we can calculate the polygon bordering the layer, as well as the obstacles. We will now explain in detail each of the sub-steps followed by the Layer Refinement process as shown in Figure 6.

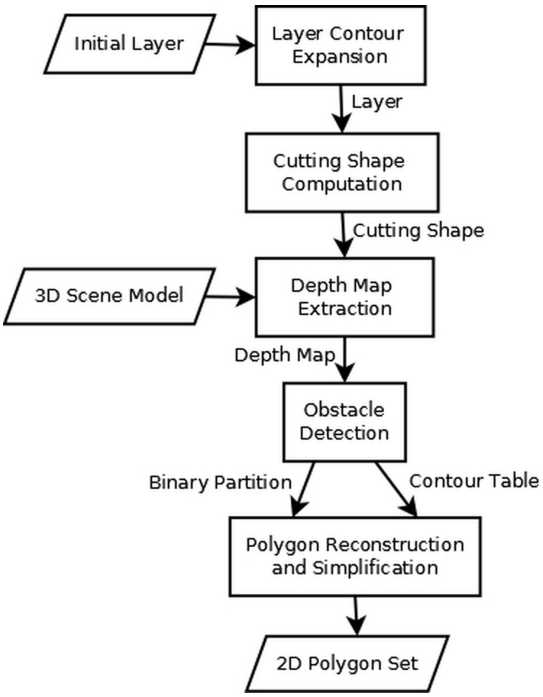

Figure 6: Layer refinement diagram.

Layer contour expansion

For each layer obtained from the coarse voxelization we have two types of voxels: *accessible* and *obstacle*. It is possible though that *obstacle* voxels neighbours of *accessible* voxels partly contain walkable geometry. Contour expansion is thus computed in order to consider these voxels for the cutting shape calculation. Figure 7 shows the result of this step around obstacles or floating geometry.

Cutting Shape

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

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

- Channel R: the type of voxel (1 for accessible voxel, 0.5 for voxels which contain a portal between layers, and 0 for non-accessible voxel).

- Channels GBA: the depths of the cutting plane for each column of the voxelization grid.

Depth map extraction

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

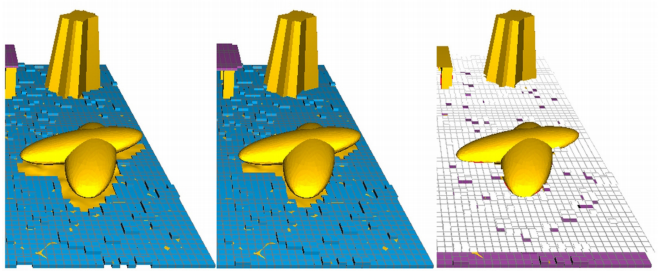

Figure 7: From left to right we can see a close up of the scene. On the left, the output of the layer extraction, in the center the result of the voxel expansion step to better capture the original geometry around obstacles, and on the right the calculated *Cutting Shape* (in white).

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

In any other case, the fragment is passed to the next step of the graphics pipeline. Since the CS can intersect with vertical walls, and those walls will not produce any fragment on the rasterization process, we need to deal with near-perpendicular polygons separately in order not to lose any details of the original geometry. This is done by storing those polygons and then for each layer rendering polygons that intersect with *accessible* voxels directly onto the depth map of the corresponding layer. Figure 8 shows the result of this process for the bottom and top layer of the example scene. Notice that black areas indicate non-walkable space (obstacles, or empty) and in grey scale we can see the depth of the walkable areas. Notice how the ramp has been split between the two layers.

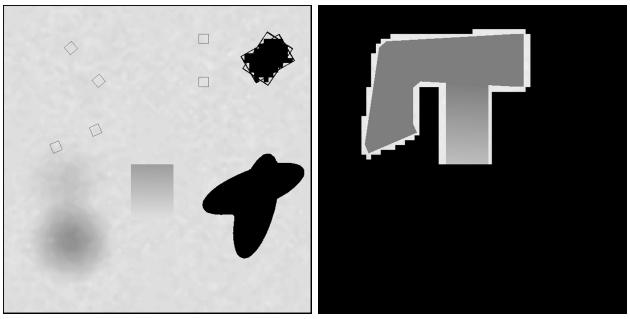

Figure 8: Depth maps for the bottom and top layers of the example scene.

Obstacle detection and polygon reconstruction

To identify floor vs. obstacles, a flood fill algorithm is performed over the depth map. Obstacles are detected when the difference in height between neighbouring pixels is bigger than the character's step height, h_s. The output of this flooding is a binary file where 1 means floor, and 0 means obstacle. Pixels belonging to the contour of an obstacle are considered vertices of the obstacle shape. To reduce the final number of vertices, we apply the Ramer-Douglas-Peucker algorithm [27].

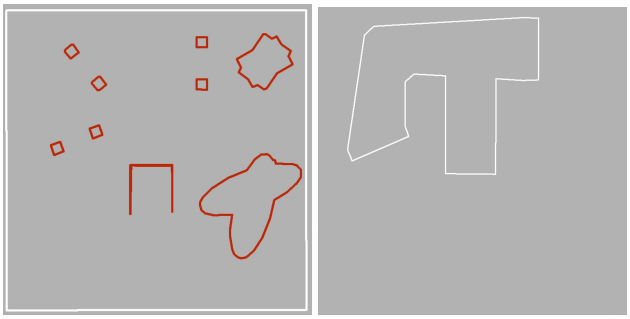

Figure 9: Floor plans of bottom and top layers. White polygons provide the shape of each layer, and red polygons represent obstacles.

3.4. NavMesh Generation

The final step of the algorithm consists of generating a navigation mesh of the scene. The floor plans generated previously are in the input format required by the navigation mesh generator ANavMG [1]: a single polygon with holes representing floor vs. obstacles. The resulting CPG is a near optimal convex subdivision of the space, since the results obtained show that the final number of cells is close to the minimum value of the range where the optimal solution lies (as proven in [1] and in the Appendix). ANavMG calculates for every notch (concave vertex) its *Area of Interest* (delimited by the prolongation of the edges incident to the notch). Then the closest element to the notch inside this *Area of Interest* is calculated and a portal is created to convert the notch into a convex vertex. The closest element to the notch can be another vertex, an edge of the geometry, or a previously created portal. Figure 10 shows the output provided by ANavMG for a simple polygon with holes.

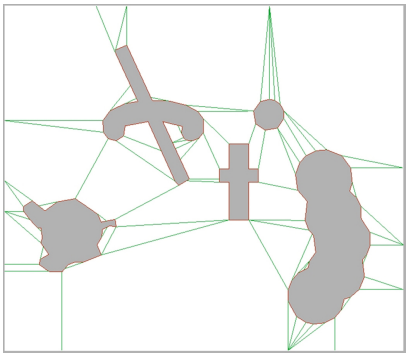

Figure 10: Cell-and-Portal Graph obtained with ANavMG (69 cells).

Convexity Relaxation

The purpose of Navigation Meshes is to have a decomposition of space into walkable cells with portals joining those cells. Path planning algorithms are used to find the path between two

cells of the navigation mesh, and a local movement algorithm usually deals with the character's displacement within a cell. Local movement algorithms, use different techniques to avoid obstacles that can also be used against small concavities in walls, therefore we can further reduce the number of cells in the final navigation mesh if we take this into consideration and relax the notion of convexity. Oliva and Pelechano [1] introduced this idea, by allowing to slightly increase the internal angle of the *Area of Interest*. However their solution does not always give good results since it does not guarantee to reduce the number of concave vertices for all scenarios, and it may lead to even more ill-conditioned polygons. Therefore in this work we present a different approach that matches better the local movement abilities of characters. Depending on the local movement algorithm being used for collision detection and avoidance, the user can decide empirically the convexity relaxation threshold, τ_{cr}. Our approach has some similarities with the Ramer-Douglas-Peucker algorithm [27], which is commonly used to reduce the number of points in a curve that is approximated by a series of points. Our method focuses exclusively on notches in the floor plan, since our goal is to determine which notches can be ignored when creating portals. So the algorithm is run for every sequence of notches found between two convex vertices (by sequence we mean 1 or more). Given the input threshold τ_{cr}, the algorithm recursively finds the notches that need to be kept in order to create portals. So at each step it finds the center notch of the sequence and calculates the distance between the center notch and the line segment joining the first and last vertex. If the distance is bigger than τ_{cr} then that notch needs to be kept and the algorithm calls itself recursively for the sequence of notches before and after the center notch. The recursivity stops when the distance is smaller than τ_{cr}, and all notches not marked as kept are ignored. Note that we are not eliminating these notches, we are simply not creating portals to split these notches into convex vertices.

Figure 11 shows the navigation mesh after applying our convexity relaxation method to the example shown in Figure 10 . This solution allows us to keep a detailed geometry for local movement purposes, yet reduces the final number of cells to speed up path planning.

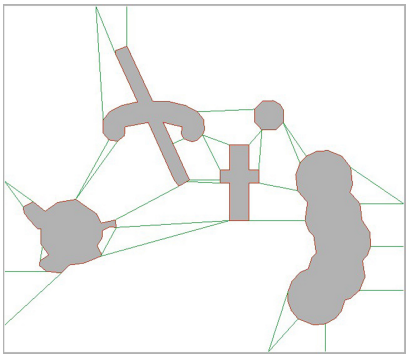

Figure 11: CPG of the example shown in Figure 10 after applying convexity relaxation (τ_{cr} = 0.5). The final number of cells is 29.

Merging layers

During the layer extraction step, we mentioned that channel R of the cutting shape texture stores the value 0.5 when a pixel could belong to boundary between layers. This information is used to determine the portals that join a layer with its neighbouring layers. During this step, the algorithm iterates through these possible portal edges to join them together. After the merging process is finished, the navigation mesh could end up with some non-essential portals. A non-essential portal is defined as a portal that if removed will not leave a notch in the navigation mesh. For example in Figure 12 we see that a portal has been created in the center of the ramp to join the bottom and top layers (black dotted line) that could be removed merging the cells of the ramp. Therefore the final step of the algorithm consists of searching for non-essential portals around the merged areas.

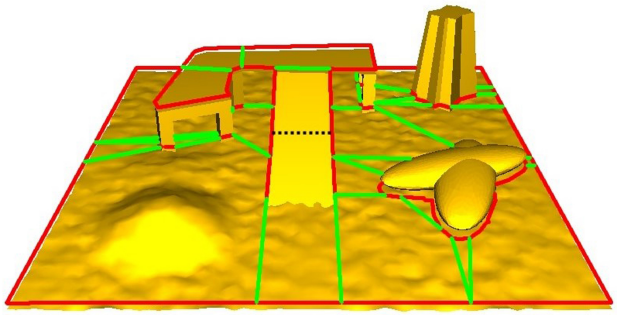

Figure 12: The merging step will join the top and bottom Navigation Meshes into one, and will also eliminate the non-essential portals around merged areas (black dotted line).

4. Results

We have tested our algorithm in several scenarios of increasing complexity and number of vertices. NEOGEN has successfully generated the Navigation Mesh for all the multi-layered 3D environments tested. In Table 1 we have a summary of some different scenarios in which we have tested our algorithm. Figures 15 and 16 show the visual results obtained. It is important to mention that our algorithm is robust against intersecting geometry, cracks and holes (which would be treated as obstacles). This is a very important advantage, since it makes easier the task of designing scenarios.

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

Table 1: Summary of the scenes tested (with references to the Figure number in the paper) with the number of triangles and layers for each scene and the final number of cells generated by NEOGEN for the given convexity relaxation threshold, τ_{cr}.

In Figure 13 we show the time taken by NEOGEN to output the NavMesh for each scene (tested with Intel Core 2 Quad

Q9300 @ 2.50GHz, 4GB of RAM and a GeForce 460 GTX). Even though the execution time of the algorithm is not a major goal, it is important that the process run as fast as possible, to make it easier for the designer to make changes to the geometry and see the impact on the resulting navigation mesh at interactive rates.

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

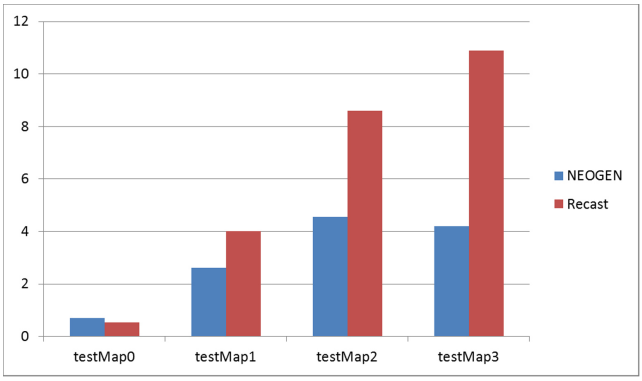

Figure 13: Computation time (in seconds) taken for each of the tested scenes.

Another advantage of our method is that it creates cells that adjust tightly to the geometry. Figure 14 compares the adjustment to the contour offered by NEOGEN and Recast. As we can observe, NEOGEN provides a better adjustment to the contour of the obstacles, and hence, our description of the walkable space is more accurate as can be seen in the contour around the columns (notice that the small visual offset between the cell limits and the contours in our method is due to the cell edges being rendered slightly above the terrain). The voxelization used in Recast provides a coarse approximation of the geometry that can be refined by reducing the size of the voxels, but at the cost of generating significantly more cells (this issue arises regardless of the character's size used to calculate clearance through Minkowski sum). NEOGEN offers a tight fit to the geometry without adding unnecessary cells.

Finally, in Figure 15 we can see one of the limitations of Recast being that it creates walkable regions where a character cannot access. This occurs as Recast only takes into account whether the geometry enclosed by a voxel is traversable by the character or not, but it does not take into account any type of connectivity. Some of those regions can be discarded if they are small enough compared to the real walkable region, but if not the resulting CPG will have many unnecessary cells. While this limitation may not be a problem in situations where connectivity is assumed (for example, where characters can jump or teleport) we believe that accessibility (or the lack thereof) should be dictated by a set of parameters that describe the abilities of the character.

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

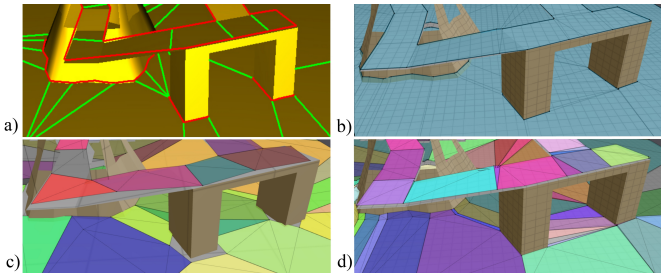

Figure 14: Adjustment to the geometry of the NavMesh in our method (a) and Recast (b,c,d). Recast results are calculated with agents radius being 0 (to eliminate the object enlargement carried out by Recast to account for clearance). In (b) we can observe a good adjustment to the geometry when the original cell size for the voxelization is 0.1, which has the drawback of creating too many cells (see (d) where cells are shown with different colors), and taking too long to compute. When the cell size is big (0.9), Recast can obtain a small number of cells (c), but with very bad adjustment to the geometry and even intersections (notice the bottom of the columns). In our result (a) we combine small number of cells with tight adjustment (cell borders drawn slightly above the geometry for clarity).

5. Conclusions

We have presented a novel fully automatic system entitled NEOGEN, to compute a Near Optimal convex decomposition from a multi-layered complex 3D environment given as a polygon mesh.

Our method can be divided into the following steps: firstly, a coarse voxelization of the scene is done in order to obtain a first approximation of the walkable area. Secondly, the potentially walkable area is subdivided into layers, using an ordered flooding process, and the layers that are not connected with the user seed, s_w, are discarded. Then, each layer is refined by using the fragment shader at higher resolution and the NavMesh is computed. Finally, all those individual NavMeshes are merged into a single one, that represents the walkable space of the entire scene.

The results show that convexity relaxation is a powerful tool to reduce the final number of cells, especially when the scenario contains many rounded objects, and hence, the resulting CPG fits well with the requirements of an application that needs a real-time response. To the best of our knowledge, NEOGEN is the first NavMesh generator for complex 3D scenes that applies this concept successfully to reduce the size of the graph.

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

As future work, we would like to add support for dynamic events. Our current method only takes into account the static geometry, which is enough for most applications, as collisions against dynamic obstacles such as other characters is normally solved by applying a local movement algorithm. However, it is common in applications such as video games to have worlds that are constantly changing (for example, an explosion that creates a crack in the floor, a tree that falls and blocks a path, a door that blocks or makes accessible a region of the scene, etc.). In those situations, the NavMesh needs to be modified on the fly.

We would also like to extend our method to work with arbitrarily large environments. The size of the environment that we can currently handle is restricted by the resolution of the voxelization step. Hence, the main idea is to subdivide the scene into regions small enough to fit with the resolution of the voxelization, treat each of these regions as an individual multi-layered map applying the method described in this paper, and finally join the resulting NavMeshes into a single one.

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

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2010-20590-C01-01.

References

[1] Oliva R, Pelechano N. Automatic generation of suboptimal navmeshes. In: Proceedings of the 4th international conference on Motion in Games. MIG'11; Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-642-25089-7; 2011, p. 328–39.

[2] Arikan O, Chenney S, Forsyth DA. Efficient multi-agent path planning. In: Proceedings of the Eurographic workshop on Computer animation and simulation. New York, NY, USA: Springer-Verlag New York, Inc. ISBN 3-211-83711-6; 2001, p. 151–62.

[3] Young T. Expanded geometry for points-of-visibility pathfinding. In: Game Programming Gems 2. Charles River Media; 2001, p. 317–23.

[4] Sud A, Gayle R, Andersen E, Guy S, Lin M, Manocha D. Real-time navigation of independent agents using adaptive roadmaps. In: Proceedings of the 2007 ACM symposium on Virtual reality software and technology. VRST '07; New York, NY, USA: ACM. ISBN 978-1-59593-863-3; 2007, p. 99–106.

[5] Rodriguez S, Amato NM. Roadmap-based level clearing of buildings. In: Proceedings of the 4th international conference on Motion in Games. MIG'11; Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-642-25089-7; 2011, p. 340–52.

[6] Snook G. Simplified 3d movement and. pathfinding using navigation meshes. In: Game Programming Gems. Charles River Media; 2000, p. 288–304.

[7] Reynolds CW. Steering behaviors for autonomous characters. In: Proceedings of Game Developers Conference 1999. GDC '99; San Francisco, California: Miller Freeman Game Group; 1999, p. 763–82.

[8] Pelechano N, Allbeck JM, Badler NI. Controlling individual agents in high-density crowd simulation. In: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation. SCA '07; Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. ISBN 978-1-59593-624-0; 2007, p. 99–108.

[9] Kallmann M, Bieri H, Thalmann D. Fully dynamic constrained delaunay triangulations. In: Brunnett G, Hamann B, Mueller H, Linsen L, editors. Geometric Modeling for Scientific Visualization. Heidelberg, Germany: Springer-Verlag; 2003, p. 241–57. ISBN 3-540-40116-4.

[10] Curtis S, Snape J, Manocha D. Way portals: efficient multi-agent navigation with line-segment goals. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. I3D '12; New York, NY, USA: ACM. ISBN 978-1-4503-1194-6; 2012, p. 15–22.

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

[12] Kuffner Jr. JJ. Goal-directed navigation for animated characters using real-time path planning and control. In: Proceedings of the International Workshop on Modelling and Motion Capture Techniques for Virtual Environments. CAPTECH '98; London, UK, UK: Springer-Verlag. ISBN 3-540-65353-8; 1998, p. 171–86.

[13] Valve . Valve navmesh generator; 2005. http://developer.valvesoftware.com/wiki/Navigation_Meshes.

[14] Hale DH, Youngblood GM, Dixit PN. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. In: AIIDE'08. 2008, p. 173–8.

[15] Hale DH, Youngblood GM. Full 3d spatial decomposition for the generation of navigation meshes. In: Darken C, Youngblood GM, editors. AIIDE. The AAAI Press. ISBN 978-1-57735-431-4; 2009,.

[16] Kallmann M. Path planning in triangulations. In: Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games. Edinburgh, Scotland; 2005,.

[17] Kallmann M. Shortest paths with arbitrary clearance from navigation meshes. In: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. SCA '10; Aire-la-Ville, Switzerland, Switzerland: Eurographics Association; 2010, p. 159–68.

[18] Lamarche F. Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. Comput Graph Forum 2009;28(2):649–58.

[19] Jorgensen CJ, Lamarche F. From geometry to spatial reasoning: automatic structuring of 3d virtual environments. In: Proceedings of the 4th international conference on Motion in Games. MIG'11; Berlin, Heidelberg: Springer-Verlag; 2011, p. 353–64.

[20] Tozour P. Ai game programming wisdom. In: Rabin S, editor. Building a Near-Optimal Navigation Mesh. Charles River Media; 2002, p. 171–85.

[21] UDK . Unreal navmesh generator; . http://udn.epicgames.com/Three/NavigationMeshReference.html.

[22] Mononen M. Recast navigation toolkit; 2009. http://code.google.com/p/recastnavigation/.

[23] Haumont D, Debeir O, Sillion FX. Volumetric cell-and-portal generation. Comput Graph Forum 2003;22(3):303–12.

[24] van Toll W, Cook IV AF, Geraerts R. A navigation mesh for dynamic environments. Journal of Visualization and Computer Animation 2012;23(6):535–46.

[25] Geraerts R. Explicit corridor map; 2010. http://www.staff.science.uu.nl/ gerae101/motion_planning/cm/index.html.

[26] Eisemann E, Décoret X. Fast scene voxelization and applications. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. ACM SIGGRAPH; 2006, p. 71–8.

[27] Douglas DH, TK. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization 1973;10(2):112–22.

[28] Fernandez J, Canovas L, Pelegrin B. Algorithms for the decomposition of a polygon into convex polygons. European Journal of Operational Research 2000;121(2):330–42.

Appendix: Near Optimal Decomposition of a Polygon with Holes

The convex decomposition calculated for each layer (2D simple polygon with holes) is based on creating a portal between each notch and the closest element in the polygon (edge, vertex, or portal). In a few cases, if the closest element is a portal, it may be necessary to create two new portals to avoid T-Joints between portals. Often, when this happens the initial portal can be eliminated because it turns into a non essential portal. As proven by Fernandez et. al. [28], the optimal number of cells, OPT, for a convex decomposition of a polygon with holes is within the bounds:

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

Where r is the number of notches, and h the number of holes. The lower bound corresponds to the ideal case where all portals created join two notches, and the higher bound corresponds to the case where a portal needs to be created for each notch to turn it into a convex vertex. Since the NavMesh generator creates in most cases one portal per notch, our convex decomposition will theoretically provide a result around $r - h$, although in practice as we increase the complexity of the polygons our result tends towards the lower bound .

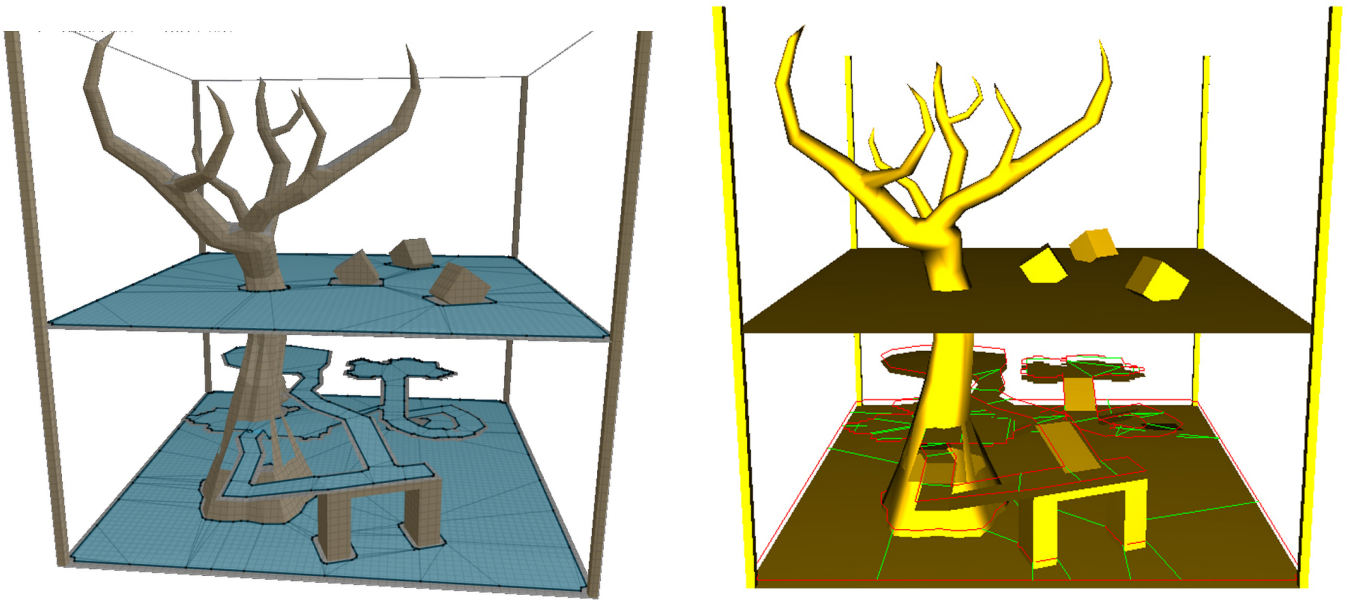

Figure 15: Comparison between the results given by Recast and our method (using map3). As we can see, our method not only successfully eliminates unreachable layers, but also it calculates a navigation mesh with a much lower number of cells.
Note also that the shape of obstacles is perfectly adjusted by our cell decomposition (cells rendered slightly above the geometry for clarity), without increasing unnecessarily the number of cells in the final CPG.

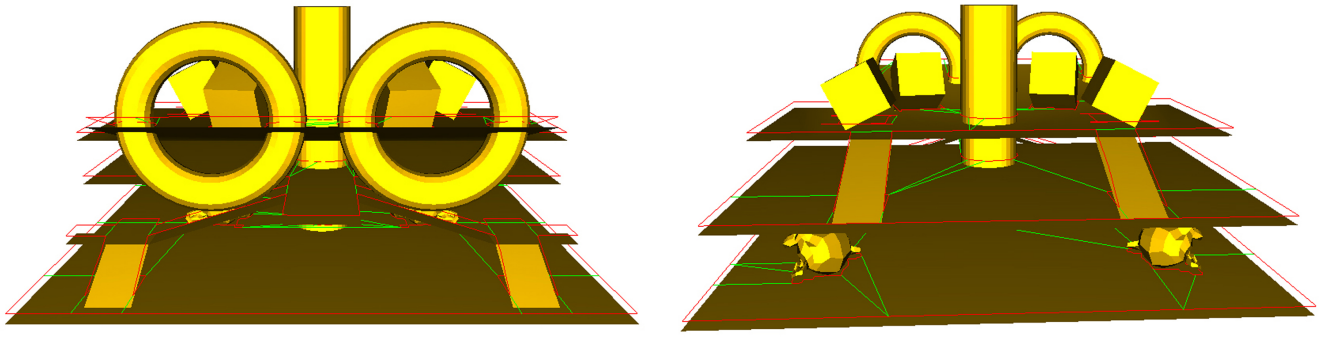

Figure 16: Two views of the Navigation Mesh for the scene map2 (3 layers and 7,308 triangles) NEOGEN calculates 86 cells with $\tau_{cr} = 0.5$