CEIG - Spanish Computer Graphics Conference (2012)
Isabel Navazo and Gustavo Patow (Editors)

A GPU Based Method for the Automatic Generation of Near-Optimal Navigation Meshes

R. Oliva[1] and N. Pelechano[1]

[1]Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract

In this paper we present a novel, robust and efficient GPU based technique to automatically generate a Navigation Mesh for complex 3D scenes. Our method consists of two steps: firstly, starting with a 3D scene representing a complex environment of one floor with slopes, steps, and other obstacles, it automatically generates a 2D representation based on a single polygon (floor) with holes (obstacles).This step can handle degeneracies of the starting 3D scene model, such as interpenetrating geometry. Secondly, a novel method that exploits the GPU efficiency is used to automatically generate a near-optimal convex decomposition which will represent the cell and portal graph of the environment. Such convex decomposition is a 2D representation of the walkable areas of the environment with portals indicating the crossing borders. The results show that the presented technique not only is more robust than previous CPU methods, but also for the tested environments with up to 1000 vertices, it performs five times faster.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computational Geometry and Object Modeling]: Geometric algorithms, languages and systems

1. Introduction

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

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

In [OP11], we presented a method to automatically generate near-optimal *NavMeshes* in the CPU. The method, entitled ANavMG, calculates for every notch (concave vertex) of the scene, the closest element that lies on the area formed by the prolongation of the edges incident to the notch, and creates a portal between the notch and the closest element, which can be a vertex, an edge or a previously created portal. The main target of this algorithm is to create a near-optimal convex decomposition of the scene, and the brute force version of the algorithm solves the problem in quadratic time.

In this paper we present an efficient and robust GPU approach to speed up the step of searching for the closest element, based on performing renders of the scene for each notch with the camera parameters defined by characteristics of the notch. The new method not only has higher performance values which allow the user to have very large environments, but it is also more robust since it overcomes several problems that ANavMG presented. Unlike our previous work which created *NavMeshes* for 2D floor plans, this new method can also deal with complex one-level 3D environments.

The main contribution of this paper is a novel GPU based method to generate a *NavMesh* for a given 3D scene, representing a complex environment of a single floor. Our method has two main steps: firstly it abstracts away the information of the 3D model that represents the scene (with its slopes, steps and other obstacles) to automatically convert it into a 2D representation based on a single simple polygon (floor) that can contain holes (obstacles). Secondly, it automatically generates a near-optimal convex decomposition of this 2D representation. Our method is robust against degeneracies of the starting 3D model, such as interpenetrating geometry.

© The Eurographics Association 2012.

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG12/049-056

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

2. Related Work

The concept of *Navigation Mesh* first appeared in [Sno00] described as a triangular decomposition of the walkable area. Some notions to construct an acceptable *NavMesh* are given, such as trying to create as few cells as possible and avoid to have overlapping cells. It does not provide an automatic method, so the *NavMesh* needs to be manually constructed.

Triangular Meshes are commonly used to represent a *Navigation Mesh*. In [KBT03, Kal05], a dynamic Constrained Delaunay Triangulation (CDT) is used to represent the walkable area of a scene. Triangular *NavMeshes* are a good first approach because it guarantees that every cell created is convex and in the case of Delaunay triangulation, it also guarantees that it generates the lowest possible number of ill-conditioned triangles. The method proposed also allows the incremental insertion, move and removal of obstacles, adapting the *Navigation Mesh* in consequence. The main drawback is that many unnecessary cells are created, increasing the time for calculating a path between two given cells, which can be specially problematic in applications such as videogames, where a real-time response is required. In [DB06] the CDT technique is compared against grid-based maps of real commercial videogames. The results show that the use of a CDT to represent the walkable space dramatically reduces the computation time to find a path between two points, compared to the grid representation of the same map. In [Kal10], more uses of the CDT are explored, such as the automatic placement of agents in the free space and path planning with clearance. In a recent publication [QCT12], a method for computing the CDT using the GPU has been presented. The implementation is done using the CUDA programming model [cud] on NVIDIA GPUs and the results show that it runs several times faster than any CPU method.

Lerner et al. [LCC06] presented a method to automatically generate a *Cell-and-Portal Graph* that worked both for interior and outdoor scenarios. The goal of their algorithm was to solve visibility problems, so the cells are not guaranteed to be convex. However, this algorithm could be easily adapted to create a *Navigation Mesh* using a post-processing step to convert the resulting cells into convex, for example, using the *Hertel-and-Mehlhorn method* [HM83] that is used to decompose a simple polygon without holes into convex regions.

In [HYD08], an automatic *NavMesh* generator method is described, that consists in spreading a certain number of unitary quad seeds on the scene. Those quads are expanded as much as possible, adjusting to the contour of the obstacles even if they are not Axis-Aligned. When the algorithm ends, a merging process is applied to reduce the number of resulting cells. The problem is that it is restricted by quads, so depending on the complexity of the scene, many of them need to be created to completely full-fill the walkable area. Another issue is that there can be intersection of portals which could be problematic when applying a local-movement method, leading to unnatural movement of the characters. The merging process helps to reduce the final number of cells, but the result is far from the optimal subdivision. In addition to these problems, the method only works if every obstacle is convex, so a previous step to decompose the obstacles into convex parts is required. A 3D version of this algorithm was proposed in [HY09], but it has the same limitations as the 2D version.

Toll et al. [TCG11] presented an automatic *NavMesh* generator for a multi-layered environment, such as an airport or a multi-story car-park, where the different layers of the scene are connected by elements such as stairs or ramps. Each layer is represented as a set of 2D polygons that lies in the same plane, and the medial axis set for the layer is computed. The connections between layers are used to iteratively merge the different sets of medial axis and create a single data structure. Then, they extend this structure by adding segments with the closest obstacle to create a convex partition of the scene. The main problem is that the use of the medial axis seems to be inadequate for the computation of the *NavMesh* because a great number of unnecessary cells are created. Also, this technique creates many degenerated cells, which can introduce artifacts on the movement of the virtual characters. An approximation of the medial axis set can be computed using the GPU, as described in [HCK*99].The implementation of this *NavMesh* generation method, restricted to one single layer, can be found in [ecm]. It requires to manually creating a file that describes the contour of the obstacles, so the process is not fully automatic.

Even though *Navigation Mesh* is the most commonly used solution to solve the navigation problem in videogames, just a few Game Engines and third party applications offers the possibility of creating this *NavMeshes* automatically. Valve's Game Engine has an automatic *NavMesh* generator method based on subdividing the virtual map by axis-aligned quads [val]. As any method based exclusively on quads, it is not really extensible to maps with arbitrary geometry. In addition, if the environment contains very steep stairs, ramps or hills, the generator system makes errors, resulting in a *NavMesh* that does not cover the entire map. So it is necessary to manually complete the *NavMesh*.

Unreal Engine [unr] has also its own *NavMesh* generator. Firstly, a high-density grid that covers all the walkable area is automatically generated. To adjust to the obstacle contours, the size of the cell used is modified. Secondly, this grid is simplified, merging all quads into concave slabs separated only by differences in slope. Finally, those concave slabs are decomposed into convex shapes.

Recast [rec] is an automatic open-source *NavMesh* generator broadly used in popular videogames and other complex virtual applications. The method used by Recast is based on the work by Haumont et al. [HDS03] that consists on a voxelization of the scene, followed by the generation of the cells through a watershed algorithm applied to the distance map of the scene. The cells generated by this method are not necessarily convex, so Recast applies a final step of convexalization to the resulting cells to obtain a convex partition. The main problem of using watershed to generate the cell-and-portal graph is that every local-minimum generates a cell, leading to a non-optimal partition. The voxelization step makes the method robust against degeneracies (such as cracks, holes and intersecting geometry) as well as reduces the number of local minima,

© The Eurographics Association 2012.

© The Eurographics Association 2012.

but even with this improvement, the number of generated cells is far from the optimum.

Regarding the use of shaders for visibility, in [MMG*09], a 3D urban visualization and navigation application is presented. Their approach renders the environment geometry in a cube map, using a shader that calculates the distance to the viewer for each fragment. One of the main uses of this representation is collision avoidance: If a fragment is inside the bounding radius of the viewer, a force is applied to avoid the obstacle. This method also allows them to automatically find a path outside of a bounding geometry (such as a building), if the exit point is visible from the viewer position, i.e., it is mapped on the cube map.

3. Converting a 3D World into 2D Polygons

As most *NavMesh* generation methods, the *Navigation Mesh* is constructed in 2D. However, especially in the case of videogames, the virtual world is typically generated using a 3D software modeler. Since we want the method to be fully automatic, the first step of our method transforms the 3D input data into a 2D representation. In particular, the input required by the *Navigation Mesh* Generator consists on a single polygon defying the floor, with the vertices given in counter-clockwise order, and holes representing the static obstacles with the vertices given in clockwise order. The 2D Abstraction step is subdivided in several stages, as can be seen in figure 1.

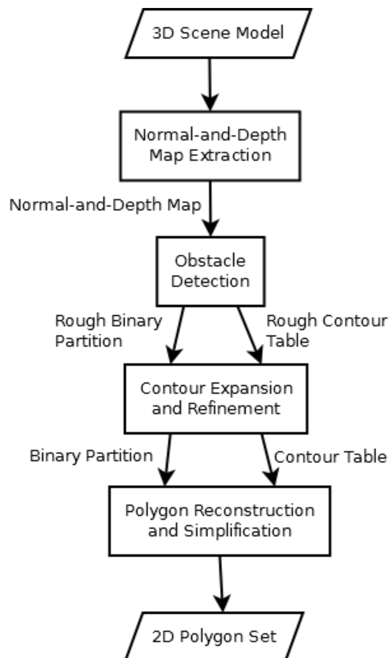

Figure 1: *This figure describes the data flow of the pipeline of the 2D Abstraction step to convert from the 3D world to the 2D representation.*

3.1 Normal and Depth Map Extraction

The first stage of the pipeline takes the 3D model of the scene as input and performs a render of the model from a top view, using an orthographic camera. A texture is created using the fragment shader, that stores the normal per fragment (red, green and blue channels) and its normalized depth (alpha channel). Figure 2 shows the resulting Normal-and-Depth Map for a given scene.

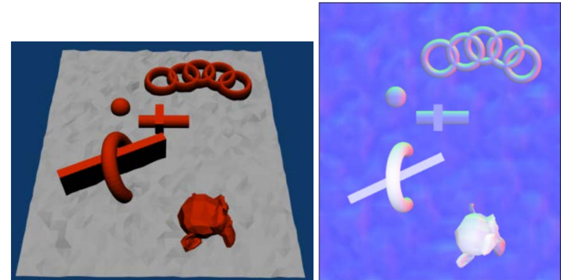

Figure 2: *The initial 3D scene (Left) and its Normal-and-Depth Map generated with the shader (Right).*

3.2 Obstacle Detection

The target of this stage is to identify walkable space (floor) vs. non-walkable (obstacles). The obstacle detection is solved using a flood fill algorithm, where the seed is introduced by the user over a walkable area (notice that this is the only input required by the user). The Normal-and-Depth map is used to determine if a neighboring fragment is similar to our current fragment. Two adjacent fragments are similar if the character can overcome the angle formed by their normals and the difference of depth. These parameters are configured through the application and depend on the walking abilities of the characters. If the neighbor fragment is reachable from the current one, then it belongs to the walkable area; otherwise it belongs to the frontier of an obstacle (contour).

The output of this stage are a Rough Binary Partition (RBP) and a Rough Contour Table (RCT). The former is a binary image representing the walkable areas (white pixels) and the obstacles (black pixels), and the latter is a table containing those pixels marked as contour (black pixels in the RBP that have at least one white neighbor). In Figure 3 we can see the binary partition with the walkable areas and the obstacles. Notice that the torus is treated as a solid obstacle seen from above and thus the floor underneath it will not be treated as walkable.

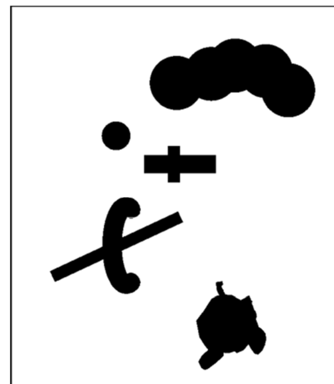

Figure 3: *The Rough Binary Partition resulting of the Obstacle Detection stage.*

© The Eurographics Association 2012.

© The Eurographics Association 2012.

3.3 Contour Expansion and Refinement

In order to ensure a one pixel wide continuous contour with an area greater than zero (i.e. no obstacles of size one pixel or line obstacles) the RBP and RCT need to be further refined.

This stage is subdivided into two sub-steps. Firstly, the contour is expanded by iterating over all the pixels in the Contour Table marking as contour those adjacent pixels that in the binary partition belong to the floor, i.e. white pixels. The target of this sub-step is to avoid future degeneracies such as having obstacles mapped into a single vertex.

The Contour Refinement step removes those contour pixels that have end up completely surrounded by black pixels i.e. pixels of an obstacle, and hence, they do not belongs to the frontier of an obstacle anymore. Figure 4 shows these two steps over a given obstacle.

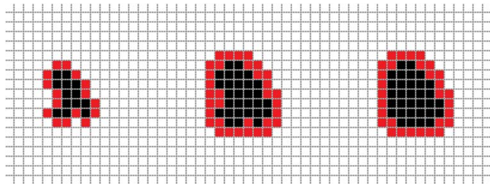

Figure 4: *The initial contour of an obstacle (Left); the expanded contour (Center); the refined contour (Right).*

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

3.4 Polygon Reconstruction and Simplification

This step will generate the 2D model representing the floor and obstacles to feed the *NavMesh* generator.

Firstly, the pixels on the Contour Table are sorted by its x coordinate, i.e., they are sorted from left to right. If the x coordinate of two contour pixels is the same, they are sorted by the y coordinate from top to bottom. Each contour pixel is considered a vertex of a polygon and then a simplification method is used to reduce the final number of vertices. Initially all contour pixels are marked as not-visited.

The algorithm proceeds by iterating over all the pixels on the Contour Table, until it finds the first not-visited contour pixel. The order of the Contour Table guarantees that this pixel is the most left one of a polygon on the Binary Partition. When reconstructing the floor, the vertices have to be given in counter-clockwise order, so for the most left contour pixel, we have to start moving to the S, SE or E neighbor pixel that is contour. If we are reconstructing an obstacle, the vertices have to be given in clockwise order, so we have to start moving to the N, NE or E. Figure 5 exemplifies the process of reconstructing an obstacle from its most left contour pixel C. In this case, the vertices have to be given in clock-wise order, so the neighbor chosen is the one marked with E.

Once the first neighbor has been decided, the process continues by selecting and setting as visited at each iteration the contour pixel that is closest to the current one, that has not been visited yet. In this case, all the adjacent neighbors of the current pixel are checked. The Contour Expansion and Refinement stage ensures that we always have a unequivocal neighbor contour pixel to choose as next. It also ensures that every reconstructed polygon has an area greater than 0, and that we do not have degeneracies such as obstacles reconstructed as a single point. The process of reconstructing a polygon ends when the start pixel is reached and the process of reconstructing all the polygons finishes when all the pixels on the Contour Table have been marked as visited.

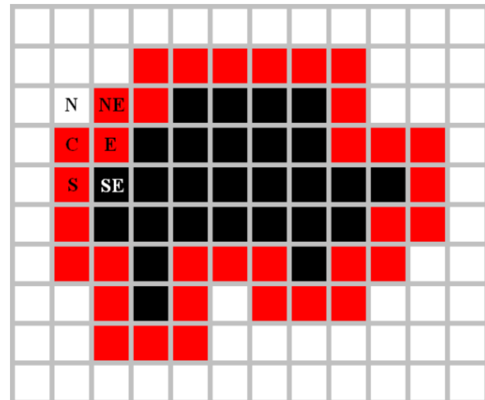

Figure 5: *The most left contour pixel C of an obstacle and the potential neighbors that can be chosen as next.*

To reduce the total number of vertices per polygon, the first straight forward simplification consists on eliminating all vertices that belong to segments aligned horizontally, vertically or with 45º angle and are not end points. This pre-simplification step is done during the reconstruction process. Next, the Ramer-Douglas-Peucker Algorithm [Ram72, DP73] is applied to further simplify the polygon (figure 6).

Figure 6: *A polygon on the Binary Partition (Left); A high density pre-simplified polygon (Center); The final simplified polygon (Right).*

4. Automatically Generating NavMeshes

The algorithm to generate *NavMeshes* requires as an input the 2D simple polygon with holes described in this paper. Such polygon can either be the output of the algorithm described in section 3, or it could be a 2D input given by the user. The algorithm presented in our previous work [OP11], which was fully implemented on the CPU, consisted on identifying the notches (vertices with an angle greater than π) and transforming them into convex vertices. Notches are vertices that cause concavities in the geometry, and thus transforming them into convex vertices provides a convex subdivision of the space which was proved to be a near-optimal partition of the polygon. The transformation

© The Eurographics Association 2012.

© The Eurographics Association 2012.

from concave to convex is performed by creating a portal between each notch and the closest element (vertex, edge or portal) lying inside the Area of Interest of the notch. The Area of Interest is determined by the area formed by prolonging the incident edges of the notch. The method ensures that in the cases where the closest element is a vertex or an edge, only one portal needs to be created per notch. In the cases when the closest element is a portal with neither endpoints lying inside the Area of Interest, it is required to create two portals to transform the notch into a convex vertex. Hence, this algorithm leads to a near-optimal convex partition of the space.

In this paper we present an improved version of the CPU solution introduced in [OP11] as well as a novel GPU solution which is not only more efficient but also more robust. We encountered a problem in the previous work which appears when the closest element of a notch is a previously created portal. In this case a portal is created between the notch and one (or both) of the endpoints of the portal, but without checking whether the endpoint is visible from the notch, which in some cases can lead to intersecting geometry. On section [4.2] a solution to this problem is presented.

4.1 The GPU based version

The CPU solution presented in [OP11] has a cost of $O(n{\cdot}r)$, where *n=number of vertices* and *r=number of notches*. So if *r* is similar to *n*, the algorithm to generate *NavMeshes* has a $O(n^2)$ cost to solve the problem. This is not an important handicap a priori, since the *NavMesh* construction is normally an offline process, but it becomes an issue when dealing with dynamic environments that require continuous updates of the *NavMesh*, as it happens in videogames.

The new method based on GPU, starts by assigning a unique color identifier to each edge of the scene, which will be used for rendering and identification purposes. Then, the 2D scene is rendered from the point of view of every notch, with the parameters of the camera set based on the characteristics of the notch. The position of the camera is given by the position of the notch, the FOV of the camera is equal to the angle formed by the prolongation of the edges that define the Area of Interest of the notch and the forward direction of the camera is defined as the sum of the unitary vectors that define the Area of Interest. Once the camera has been configured, the scene is rendered and the result is stored on a one-dimensional texture that contains those elements visible from the point of view of the notch, as can be seen in figure 7.

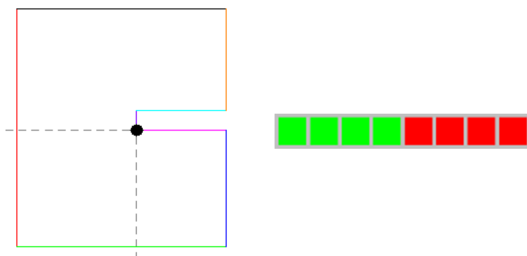

Figure 7: *A simple scene with all edges drawn with a unique color (Left) and the texture generated from the point of view of the notch (Right).*

© The Eurographics Association 2012.

To recover the edges visible from the notch, we check every pixel of the texture. The color of such pixel identifies the edge. Then, we determine which of those edges is the closest one to the notch and we create a portal with its best candidate. We cannot simply read the depth of each pixel, because we need Euclidean distances to the notch.

A critical parameter that affects directly the performance of the algorithm is the *zFar* of the camera. To avoid rendering an unnecessary number of elements that are occluding each other, the *zFar* is dynamically updated. A variable *zFarScene* contains the average of the distances to the closest element of the already visited notches. Initially, *zFarScene* is set to $1/10^{th}$ of the diagonal of the bounding box of the scene. Then for each notch, a render is performed with *zFar* set to *zFarScene*. If no element has been rendered with such *zFar*, the *zNear* is set to the current *zFar* and the *zFar* is doubled in order to carry out a new render. This process continues until at least one element has been found that lies in the Area of Interest of the notch. Once the closest element to the notch is found, *zFarScene* is updated accordingly. Notice that this process implies several renders for some notches, but we have found empirically that the *zFarScene* converges towards an optimal value that results in the most efficient render for a large number of the notches in the given scene. The entire process is described on figure 8.

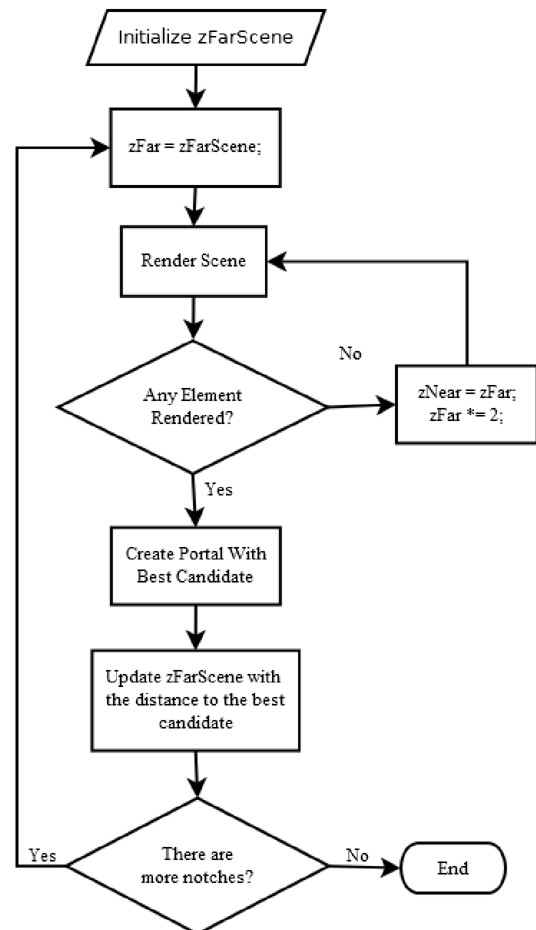

Figure 8: *Diagram describing the dynamic update of the zFar.*

© The Eurographics Association 2012.

4.2 The Portal Vertex-Portal case

The most complicated case that our algorithm must handle is when the closest element to the notch is a previously created portal. In order to avoid intersections between portals, the algorithm presented in [OP11] proposes creating a portal between the notch and one of the endpoints of the portal that is inside the Area of Interest of the notch. If neither endpoint lies within the Area of Interest, then two portals are created to join the notch with each of the endpoints of the previous portal. However, that approach can cause intersection problems, when the endpoints are not actually visible from the notch (see figure 9).

To solve this problem and carry out a fair comparison between the CPU solution, and the new approached based on GPU, we have modified the previous algorithm so when a portal is created with another portal, we check for intersections between the segment formed by the notch and the endpoint of the portal and the rest of edges in the scene. If there are no intersections, then the portal can be created; otherwise, the portal is created with the new found closest edge.

In GPU mode this problem is solved by using one extra render step. The new Area of Interest is defined as the one delimited by the segments that join the notch with the endpoints of the previous portal as can be seen in figure 9.

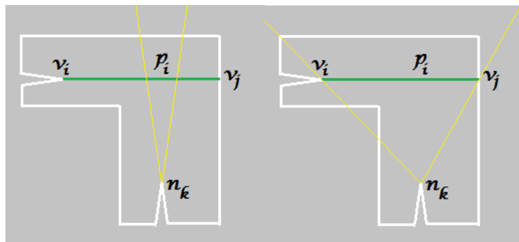

Figure 9: *The original Area of Interest of a notch n_k (Left). When the closest element is a portal p_i, the new Area of Interest is defined by the notch and the endpoints of the portal (Right).*

The camera parameters are thus updated accordingly and a new render of the scene is performed. Then the algorithm checks for intersections between the segments joining the notch with the endpoints and the edges that appear on the new render. Notice that the GPU version needs to check against a reduced number of edges unlike the CPU version that checks against all edges in the scene.

5. Results

In this paper we have presented a framework to obtain *Navigation Meshes* from 3D complex environments consisting of one layer where we could carry out navigation for characters. Firstly a method has been described to abstract the 3D geometry into a 2D simple polygon with holes. The results shown in figures 2,3,13 demonstrate the robustness of the method to deal with complicated environments with ramps, steps, holes, and so on. The method provides the flexibility of being adjusted to the walking abilities of the characters, to determine the height of the steps, and the angle of a ramp that a character can easily overcome. Secondly we described a novel GPU based approach to speed up the search for closest element to a notch.

The experimental results have been obtained on a NVIDIA GeForce 8800 GTX and an Intel Core 2Quad Q6700 at 2.66GHz with 8 GB of RAM. We have tested the new GPU based algorithm presented in this paper to generate automatically *NavMeshes* against an optimized version of the work presented in [OP11] with the extension of checking for visibility in the case of creating portals between a notch and a previous portal.

To test the overall performance of the algorithm, we created 10 scenarios of increasing complexity ranging from 23 vertices to 965.The algorithm applied dynamic *zFar* calculation. Figure 10 shows the time taken by both CPU and GPU implementations. As we can see, the time taken by the CPU version to solve the problem increases quadratically, whereas the GPU version increases nearly linearly with the number of vertices of the environment. Notice that the GPU algorithm can be quadratic in the worst case, but in practical scenarios it performs with nearly linear time, since it applies geometry culling using an octree to render the 1D texture for each notch.

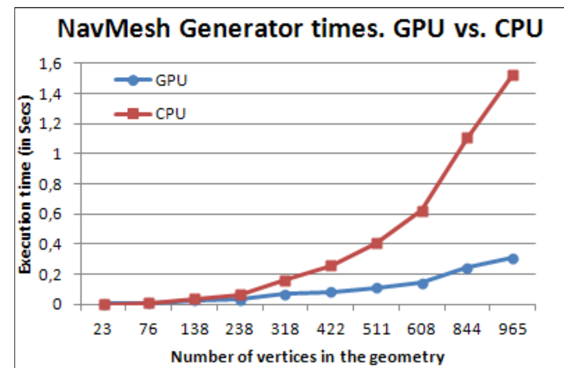

Figure 10: *Time comparison between the CPU and GPU versions of 10 scenarios with different complexity.*

The experimental results were obtained performing renders over a viewport of 1x32 pixels that were then mapped onto a texture. The size of the texture (and thus the viewport) used is important for the overall performance of the algorithm, as can be seen in figure 11.

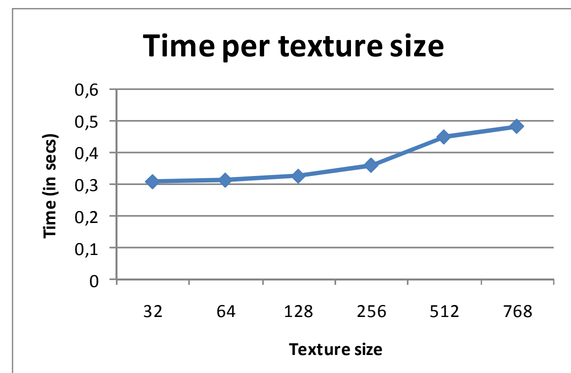

Figure 11: *Time spend to solve the most complex test environment, for different sizes of texture.*

© The Eurographics Association 2012.

© The Eurographics Association 2012.

We have found empirically that a size of 32 pixels for the texture is adequate to correctly identify the closest element. This comparison table was obtained with the largest scene of 965 vertices, since for smaller scenes the difference is less significant.

The value of the *zFar* chosen for performing the render from each notch has also an impact on the performance, since it determines how many segments get discarded at an earlier stage of the graphics pipeline. A large *zFar* will guarantee that all segments visible from the given notch are rendered, but with a high cost, whereas a small *zFar* will result in faster renders but may not render segments that are visible and relevant for creating the *NavMesh*. The optimal *zFar*, is thus the shortest one that allows the closest element to be rendered without rendering many additional segments that are far away and thus either not visible or simply not relevant.

In order to calculate the optimal *zFar*, we have carried out an experiment with the largest scenario. Empirically we found that for the given scenario, the optimal *zFar* was 2. Figure 12 shows the time results of generating the *NavMesh* with increasing *zFar* starting with the optimal value 2 (any value under 2 would not guarantee that the closest element is found for all notches).

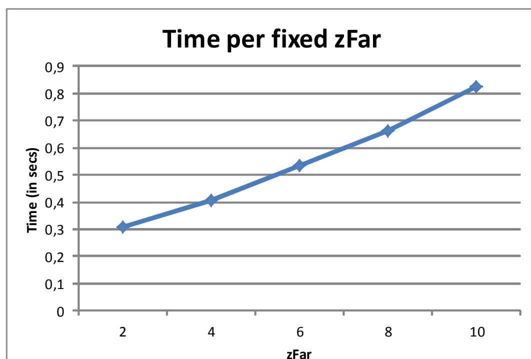

Figure 12: *Time spend to solve the most complex test environment, for different values of fixed zFar.*

Our goal with this experiment was to test whether the automatic method for calculating the *zFar* dynamically would solve the problem in similar times. Therefore we then tested that same scenario with the method presented in section 4.1. The resulting time was 0,312 seconds, with an average *zFar* of 1,25 which is automatically calculated and changes dynamically when necessary. This shows that our automatic method achieves time results similar to the optimal *zFar* calculated manually.

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

6. Conclusions and Future Work

We have presented a novel GPU based method to automatically compute a *Navigation Mesh* for a complex 3D scene, representing a single floor plant. Our method has two main steps: first a 2D abstraction is constructed from the 3D model. Then the *NavMesh* is computed using this 2D abstraction.

The results show that the GPU based version is more efficient and scalable than the CPU version. The GPU version is also more robust than the previous CPU version since it solves efficiently visibility issues that could lead to intersecting geometry.

Currently, the presented method works for complex 3D scenes representing a single layer (with ramps, steps, holes, etc.). If the original scene consisted of more levels, the user would need to manually subdivide it, treat each level independently and connect its *Navigation Meshes*. In the future we would like to extend our work to deal with several floors automatically to handle also multilayered scenes.

Finally, our current method only takes into account the static geometry which is enough for most applications. However, it is common in applications such as videogames to have worlds that are constantly changing (for example, an explosion that creates a crack on the floor; a tree that falls and blocks a path; a door that blocks or makes accessible a region of the scene, etc.). In those situations, the *NavMesh* needs to be modified. We would like to further improve our application to also handle such dynamic events in real time and modify the *NavMesh* in consequence.

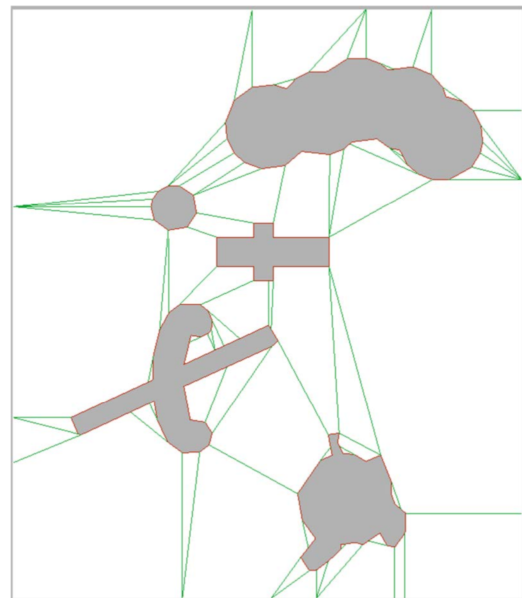

Figure 13: *The resulting NavMesh of the scene described in figure 2.*

© The Eurographics Association 2012.

© The Eurographics Association 2012.

7. References

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

[DB06] DEMYEN D, BURO, M.: Efficient triangulation-based pathfinding. In *Proc. AAAI '06* (2006), vol. 1, pp. 942-947.

[DP73] DOUGLAS D. H., PEUCKER T. K.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization 10, 2 (December 1973), 112-122.

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

[HCK*99] HOLF III K. E., CULVER T., KEYSER J., LIN M. C., MANOCHA D.: Fast computation of generalized Voronoi diagrams using graphics hardware. In Proc. SIGGRAPH '99 (1999), pp. 277-286.

[HDS03] HAUMONT D., DEBEIR O., SILLION B.: Volumetric cell-and-portal generation. Computer Graphics Forum 3, 22 (2003), 303-312.

[HM83] HERTEL S., MEHLHORN K.: Fast triangulation of simple polygons. In Proc. FCT '83 (1983), vol. 158, pp. 207-218.

[HY09] HALE D. H., YOUNGBLOOD G, M.: Full 3D Decomposition for the Generation of Navigation Meshes. In Proc. AAAI '09 (2009), pp. 142-147.

[HYD08] HALE D. H., YOUNGBLOOD G, M., DIXIT P, N.: Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds. In Proc. AAAI '08 (2008), pp. 173-178.

[Kal05] KALLMANN M.: Path Planning in Triangulations. In Proc. IJCAI '05 (2005), pp. 49-54.

[Kal10] KALLMANN M.: Navigation Queries from Triangular Meshes. In Proc. MIG '10 (2010), pp. 230-241.

[KBT03] KALLMANN M., BIERI H., THALMANN D.: Fully Dynamic Constrained Delaunay Triangulations. Geometric Modeling for Scientific Visualization (2003), 241-257.

[LCC06] LERNER A., CHRYSANTHOU Y., COHEN-ORD.: Efficient Cells-and-portals Partitioning. Computer Animation & Virtual Worlds 17, 1 (February 2006), 21-40.

[MMG*09] MCCRAE J., MORDATCH I., GLUECK M., KHAN, A.: Multiscale 3D navigation. In Proc. I3D '09 (2009), pp. 7-14.

[OP11] OLIVA R., PELECHANO N.: Automatic Generation of Suboptimal NavMeshes. In Proc. MIG '11 (2011), pp. 328-339.

[QCT12] QIM, CAO, T-T., TAN, T-S.: Computing 2D constrained Delaunay triangulation using the GPU. In Proc. I3D '12 (2012), pp. 39-46.

[Ram72] RAMER U.: An iterative procedure for the polygonnal approximation of plane curves. Computer Graphics and Image Processing 1, 3 (November 1972), 244-256.

[rec] Recast Navigation Toolkit. http://code.google.com/p/recastnavigation/

[Sno00] SNOOK G.: Simplified 3D movement and pathfinding using navigation meshes. Game Programming Gems (February 2000), 288-304.

[TCG11] TOLL W., COOK IVA. F., GERAERTS R.: Navigation Meshes for Realistic Multi-Layered Environments. In Proc. IROS '11 (2011), pp. 3526-3532.

[unr] Unreal Engine's *NavMesh* Generation Method. http://udn.epicgames.com/Three/NavigationMeshReference.html

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

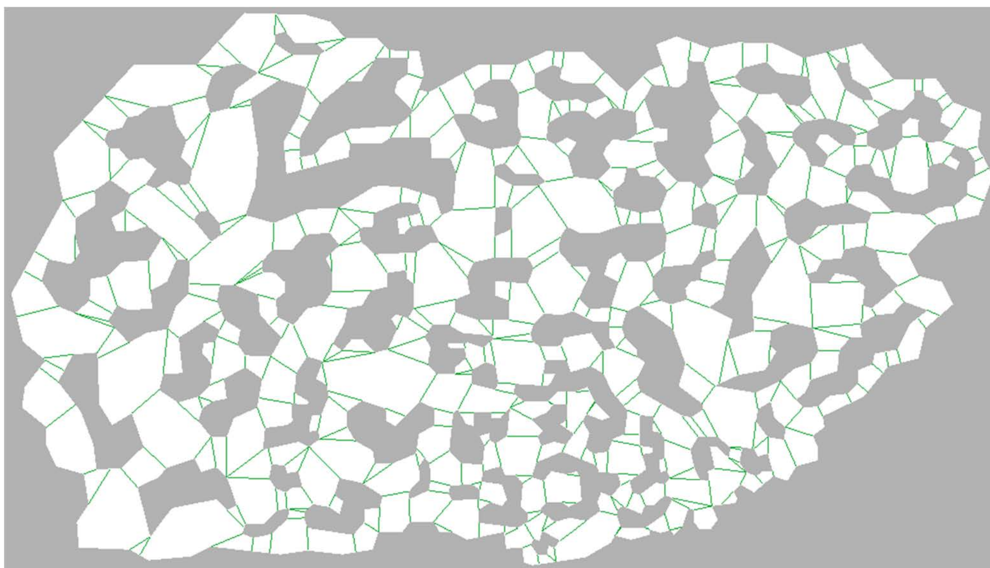

Figure 14: *The resulting Navigation Mesh for the scene containing 965 vertices and 650 notches.*

© The Eurographics Association 2012.