Automatic Generation of Suboptimal NavMeshes

Ramon Oliva, Nuria Pelechano

Universitat Politècnica de Catalunya, Barcelona, Spain
ramon.oliva.martinez@gmail.com, npelechano@lsi.upc.edu

Abstract. Most current games perform navigation in virtual environments through A* for path finding combined with a local movement algorithm. Navigation Meshes are the most popular approach to combine path finding with local movement. This paper presents a new Automatic Navigation Mesh Generator (ANavMG) that subdivides any polygon representing the environment, with or without holes, into a suboptimal number of convex cells where local movement algorithms can be applied without deadlocks. We introduce the concept of convex relaxation to further reduce the number of cells depending on the flexibility of the local movement algorithm. Finally we show results of the ANavMG and its application to a multi player game.

Keywords: Navigation meshes. Convex decomposition. Crowd navigation.

1 Introduction

Navigation meshes (NavMeshes) are commonly used to represent the walkable geometry within a virtual environment. Path finding can then be performed on a graph which abstracts away the geometry details, by representing each walkable area as a cell and the crossing segments between cells, as the portals of the graph.

When creating these NavMeshes we have two main restrictions: the first one given by the path finding algorithm which implies reducing the number of cells so that the path finding algorithm will find a suitable route as fast as possible, while the second one is given by the local movement algorithm and it implies having convex cells so that agents can move in a straight line between any two points of the cell.

In many cases, game designers need to create these navigation meshes by hand, which is extremely time consuming and can introduce errors which may either leave areas of the walkable space not accessible to the Non Player Character (NPC), get the NPCs stuck in concave regions, or create paths that do not look natural.

In this paper we introduce a new approach to automatically obtain NavMeshes for a given environment. For clarity, we will explain the algorithm for the case of 2D environments, where the input polygon representing the environment can be seen as the floor plan, and the holes represent static obstacles. This implies that we are not currently handling environments consisting of several levels with staircases, ramps, etc., but we will explain how the method could be easily extended to 3D.

The main contribution of this paper is an algorithm that takes as an input any 2D simple polygon (i.e. no self-intersections) with or without holes, and automatically splits it into a suboptimal subdivision of convex polygons that are highly suited to path finding by avoiding the presence of both degenerated polygons and almost all ill-conditioned polygons. We also introduce the concept of convex relaxation to achieve smaller navigation meshes based on the flexibility of the local movement algorithm being used for obstacle avoidance.

There is a trade-off when generating the NavMesh between having the minimum number of cells possible, so that path finding can run as fast as possible, and having portals that best guarantee natural traversals when applying local movement. Therefore the overall goals of our navigation mesh generator are:

1) To achieve as few cells as possible
2) To achieve portals as short as possible (since it introduces less inaccuracies when setting attractors to drive the natural movement of the agents).
3) To avoid cells with interior angles close to zero, since it complicates the local movements and leads to agents being physically in more than two cells simultaneously. (For the rest of the paper, we shall define an ill-conditioned polygon as a polygon with interior angles close to zero.)

In this paper we are not concerned about the time complexity of our algorithm, since it can be executed during pre-processing, and given that it only deals with static geometry, no further changes need to be made at run time. Dynamic obstacles and other agents are avoided through local movement techniques based on Reynolds' steering behaviors [13].

Once the subdivision is created, we automatically generate the cell and portal graph (CPG) representing the environment, where cells are the convex polygons resulting from the subdivision, and portals are the segments created to subdivide the original polygon into convex cells.

We finally present an example of a multi player game where path finding is carried out through A* over the generated NavMesh and movement within cells and dynamic obstacle avoidance are performed through steering behaviors. The physical library *Bullet* [2] has been integrated for several purposes including: speed up of the local movement simulation, guarantee non overlapping between agents, and keeping track of agents' within each cell to quickly update their mental maps in cases where agents are accidentally pushed through portals. Section 4 shows results of our ANavMG as well as multi agent navigation in a game application.

2 Related work

The concept of *Navigation Mesh* was introduced by Snook in his paper *Simplified 3D Movement and Pathfinding Using Navigation Meshes* [14]. He also proposed some ideas to acquire a good *NavMesh* based on polygon triangulation, but the method does not consider the creation of ill-conditioned cells that could introduce problems when local movement methods are applied.

In most games navigation meshes are used to perform path finding. The navigation mesh is then represented through a Cell and Portal Graph (CPG), where the cells correspond to convex walkable areas, and the portals correspond to the segments that are shared by adjacent cells and that can be used for crossing between those cells [11].

Navigation has also been performed through roadmaps [7], Voronoi diagrams [15] or hierarchical representation of informed environments [10]. Many techniques have been introduced for local movement within convex cells [1][11][13]. Lerner et. al. [8] presented an algorithm to automatically generate a CPG for visibility that worked both for interiors and exterior scenarios but, since the goal of their algorithm was visibility, their method generates cells that are not guaranteed to be convex. Haumount, et. al. [1] introduced an algorithm for generating CPG of interiors based on a voxelization followed by a watershed.

Hertel and Mehlhorn [5] presented a non optimal partition by diagonals, which works only for polygons without holes. The algorithm first triangulates the polygon and then removes inessential diagonals. Partitions based on diagonals are commonly used in location problems where they need to keep the total number of vertices of the polygon. But when doing a partition for navigation, it is not strictly necessary to maintain the number of vertices, and thus new vertices can be created if they result in a better partition (meaning less cells or portals that lead to more natural looking movement of the agents).

Kallman proposes an automatic triangular *NavMesh* generation method [1] based on Delaunay's triangulation, so it generates the lowest possible number of degenerated triangles. Using a triangular *NavMesh* is a good first approach because it guarantees that every cell created is convex, so a character can move in a straight line from any pair of points inside the cell. In addition, geometric operations over triangles are very efficient. 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 videogames where we need a real-time response.

Although in many cases the *NavMesh* is created by hand, some Game Engines and third parties programs offer the functionality of an automatically generated navigation mesh for a given virtual map, but they generate a great number of ill-conditioned cells or are map-type-specific. For example, Valve [17] uses a *NavMesh* generator based on subdividing the virtual map by quadrilaters. This method creates a non-optimal convex decomposition and is not really extensible to maps with arbitrary geometry.

Unreal Engine [16] has its own *NavMesh* generator, but it generates a great number of ill-conditioned convex polygons that can affect the application of local movement methods and the quality of generated paths. Recast [12] is actually the open-source *NavMesh* generator most used on popular games such as Bulletstorm, but we have detected that unnecessary cells are created that could easily be merged together, decreasing the final number of cells.

3 The Automatic Navigation Mesh Generator (ANavMG)

There are two possibilities when subdividing a polygon into convex cells. The first one consists of subdividing by adding diagonals, which are edges between pairs of vertices in the original geometry. The second one consists of using segments which are edges between a vertex of the geometry and a new point created on the boundary of the original geometry. The algorithm presented in this paper carries out a partition based on segments, and thus we are not limited by the position of the vertices in the original geometry.

3.1 Previous Concepts

The approach followed by our algorithm consists of subdividing the input polygon by first detecting which are the notches (concave vertices, i.e. interior angle larger than π) that appear in the polygon and then splitting them by creating portals so that for each original notch in the geometry, we will split it into two new angles that are both convex (i.e. interior angle smaller than π). In this way, we guarantee that if all the notches in the original polygon are split into convex angles we will obtain a partition consisting only of convex cells.

In order to ensure that we only require one new segment in the geometry to split the notch into two convex angles, we define the area or interest, I_i of a notch v_i given by two edges of the geometry, $e_{i-1,i}$ and $e_{i,i+1}$ as the resulting interior area of prolonging $e_{i-1,i}$ and $e_{i,i+1}$ as we indicate in figure 1 (left), where $e_{i-1,i}$ is the edge that joins v_{i-1} with v_i.

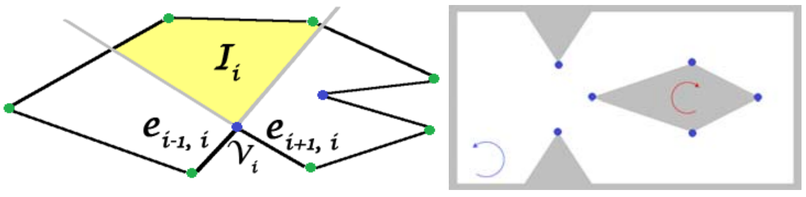


Fig. 1. On the left, we show the interest area, I_i of a notch v_i. Green vertices are convex, and blue vertices are notches that need to be split. On the right, we show a simple example of an input given to the algorithm, with the order of the vertices implying polygon (*in white*) or holes (*in grey*).

The floor plan of the virtual environment where we want our characters to navigate is given as a simple polygon, and the vertices are given in counter-clockwise order. Any obstacle within the virtual environment will be given as a polygon with its vertices in clockwise order. Obstacles can be seen as holes in the main polygon that represents the entire map (figure 1, right).

The input geometry consists of a polygon P enclosing other polygons $H_1, \ldots, H_m$, where all holes are simple empty polygons. Let δP be the boundary of the polygon P, and δH_i the boundary hole δH_i. We assume that the following conditions apply:

$$\begin{aligned} &1)\quad \delta P \cap \delta H_i = \varnothing, \quad \forall i = 1,\ldots,h \\ &2)\quad H_i \cap H_j = \varnothing, \quad \forall i \neq i \end{aligned} \qquad \textbf{(1)}$$

The first step of the algorithm consists of determining which vertices are notches. This step is performed through an orientation test based on calculating the signed area of the triangle defined by three consecutive vertices, v_i, v_{i+1}, v_{1+2}:

$$A(v_i v_{i+1} v_{i+2}) = \frac{1}{2}\begin{vmatrix} \overline{v_i v_{i+1}}_{,x} & \overline{v_{i+1} v_{i+2}}_{,x} \\ \overline{v_i v_{i+1}}_{,y} & \overline{v_{i+1} v_{i+2}}_{,y} \end{vmatrix} \tag{2}$$

If the area $A(v_i, v_{i+1}, v_{1+2})$ is positive, it means that vertex v_{i+2} is on the left hand side of edge $e_{i,i+1}$ given by the previous vertices v_i, and v_{i+1}. If it is negative, it means that v_{i+2} is on the right hand side of edge $e_{i,i+1}$. So for the main polygon which is given in counter-clockwise order, if the area is negative it means that v_{i+1} is a notch and thus needs to be split, whereas for the holes, given in clockwise order, we will also find a notch when the area is negative. We will introduce all notches of the geometry in a vertex list $\mathcal{V}$ to be treated in order. This step has cost $O(n)$ where n is the total number of vertices of the geometry.

3.2 Creating portals

For each v_i in $\mathcal{V}$, the algorithm looks for the closest element in the geometry that falls within its area of interest $\mathcal{I}_i$ to create a portal with it. This has cost $O(n{\cdot}r)$, where *n=number of vertices*, and *r=number of notches*. Elements can be other vertices, edges of the original geometry, or portals. Depending on the element being selected, we classify three types of portals: vertex-vertex, vertex-edge, vertex-portal. Each of these cases needs to be treated differently.

3.2.1 Vertex-Vertex portals.

When the closest element to v_i is another vertex v_j of the geometry, the algorithm simply needs to create a portal p_i between v_i and v_j. As can be seen in figure 2, the portal created guarantees that v_i gets split in two convex regions, and thus no further processing of v_i is necessary to subdivide the original polygon into convex cells. If the other vertex v_j was also contained in $\mathcal{V}$ (which means that it is also a notch), then the algorithm also checks whether by creating portal p_i, v_j gets split in two convex angles. This will happen exclusively when v_i falls within $\mathcal{I}_j$ as we can see in the example shown in figure 2.

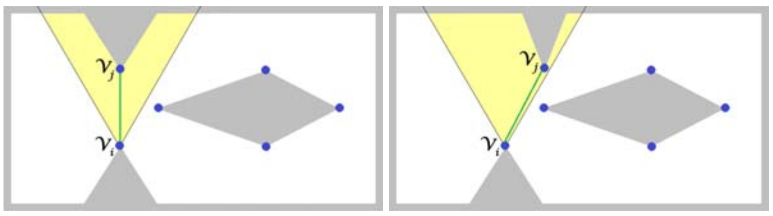


Fig. 2. Vertex-Vertex portal creation. On left, v_i also falls within $\mathcal{I}_j$, so it can be removed from $\mathcal{V}$, on the right, v_i does not fall within $\mathcal{I}_j$ and since it is a notch it still needs to be split.

3.2.2 Vertex-Edge portals.

When the closest element to v_i is an edge $e_{j,j+1}$ of the geometry, the algorithm needs to create a portal p_i between v_i and a point q in the segment $e_{j,j+1}$. Since we want portals to be as short as possible, we first consider the closest point within the segment, which is calculated as the projection of v_i over $e_{j,j+1}$, so in this case $q=(proj_e\ v_i)$.

If q falls within I_i then a new portals is created and the algorithm proceeds with the next notch in $\mathcal{V}$ (see figure 3, left), but it could be possible that even though the edge $e_{j,j+1}$ is the closest element to v_i, we could have its projection falling outside $e_{j,j+1}$ or outside the interest area, I_i, and thus the portal between those two points would not be enough to split v_i in two convex angles (see figure 3 center and right).

Therefore if the projection is not a good candidate to create a unique portal, the algorithm considers four new candidates:

- the two end vertices of $e_{j,j+1}$, v_j and v_{j+1} (see figure 3, center).
- the two intersection points q_l and q_r (if they exist) where q_l is the intersection between the closest edge $e_{j,j+1}$, and the result of extending the segment $e_{i-1,i}$ (segment on the left of v_i), and q_r is the intersection between $e_{j,j+1}$, and the result of extending the segment $e_{i,i+1}$ (segment on the right of v_i) (see figure 3, right). There is a chance that depending on the orientation of each segment, none of those intersections exist, and therefore only the ends of segment $e_{j,j+1}$ are considered.

Among the four possible vertices mentioned above, the algorithm selects the closest one that falls within I_i and a new portal is created between v_i and the selected vertex.

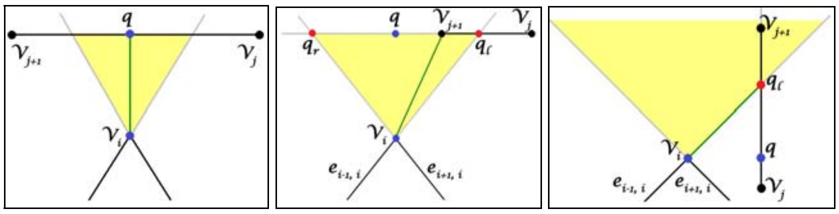


Fig. 3. Vertex-Edge portal. Candidate point q being the projection (*left*), candidates being the end points of segment v_{j+1} *(center)*, and candidate points being the intersections *(right)*.

In figure 3 we can see the three different types of portals created in the category of vertex-edge portal. As we can see the cases on the left (projection) and the right (intersection points) are the only cases where new vertices are added into the geometry. These vertices can never be notches, therefore the algorithm does not need to do any further processing with them.

3.2.3 Vertex-Portal portals.

In the case where the closest element to v_i in the geometry is a previously created portal, the treatment when creating portals differs from the vertex-edge portal since we do not want to have intersecting portals (or *T*-shapes), which would be the case for calculating a projection or intersection over an existing portal.

Therefore we assume that when the closest element is a portal p_k, we will need to create a new portal p_i with either end vertex of segment p_k. The algorithm selects the closest vertex that falls within I_i (figure 4, left and center). But since only vertices that fall within I_i can guarantee that v_i will get split into two convex areas, if none of the end vertices of p_k satisfy that requirement (figure 4, right), then the algorithm needs to create two portals instead of one. The new portals will be p_i which joins v_i with the left end of p_k and p_{i+1} which joins v_i with the right end of p_k. Notice that given the type of geometry we are dealing with, the interior angle between p_i and p_{i+1} will always be smaller than π, and therefore we guarantee that when adding these two portals, v_i will get split in three convex regions.

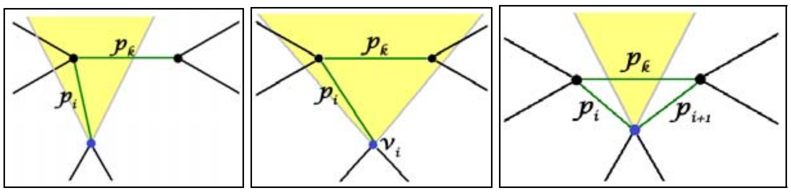


Fig. 4. Vertex-Portal portal. Only one end of p_k falls within I_i *(left)*, both ends of p_k fall within I_i *(center)*, and none of the ends of p_k fall within I_i *(right)*.

This case of portal creation is the only one that may require two new portals instead of just one per notch, but in most cases when creating these portals we will be able to remove the original portal p_k, and thus we are on average creating one portal per notch.

Removing Previously Created Portals:

When a vertex-portal portal, $p_{i,}$ is created between a vertex v_j and a previously created portal p_k, we will have at least one vertex, v_i, where both portals meet, since portals always meet at their ends which are located over existing vertices. In order to determine whether we could merge the two cells divided by portal p_k, the algorihtm checks whether p_k is still a necessary portal, since it is possible that by adding p_i to vertex v_j, this vertex already gets split in two convex regions, and thus there is no need to have two portals splitting one vertex.

To be able to remove portal $p_{k,}$ it is necessary to check whether both the left and right vertices of the portal need $p_{k,}$ not to be a notch. This step is performed by calculating the interior angle between the two neighouring segments of portal $p_{k,}$ at each end vertex (which can be edges of the geometry or other portals) and testing for convexity. If they both pass the convexity test, then we can remove p_k, and thus merge two convex cells into one larger convex cell (see figure 5).

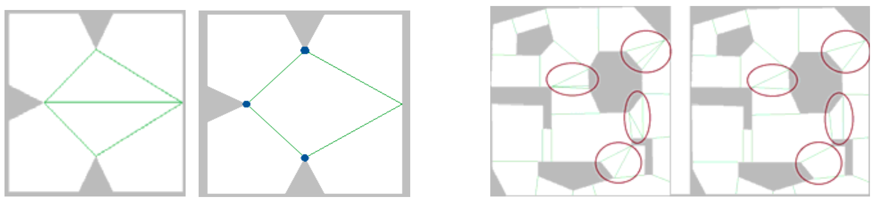


Fig. 5. Removal of previously created portal, p_k when creating a new set of portals p_i and p_{i+1} (*left*), or several cases of just one new portal being created (*right*)

3.3 Convexity Relaxation.

A vertex is defined to be convex, if its internal angle is smaller or equal than π, otherwise it is a notch. This is the mathematical definition of convexity but, in some applications such as the creation of navigation meshes, it may not be necessary to consider such a strict definition. As described previously in this paper, most navigation meshes consist of a set of convex cells with portals representing the traversal segments between them. Movement within a convex cell and between cells is driven by some local movement algorithm which can usually deal relatively easily with small concavities through obstacle avoidance behavior which also applies to static geometry (such as walls). Therefore, depending on the local movement algorithm being implemented, we can relax the definition of convexity by allowing a certain threshold τ. Relaxing the definition of convexity results in a smaller number of portals since more cells can be merged together into τ-convex cells, where τ is the threshold given by the local movement algorithm. We provide the following definitions:

Def: a vertex v_i is said to have τ-convexity if its internal angle is smaller than $\pi+\tau$.

Relaxing convexity affects not only the classification of vertices into notches, but also the definition of the area of interest of a node.

Def: an area of interest I_i is said to have τ-convexity if its internal angle is $\alpha_i+\tau$, where α_i is the original internal angle of I_i before applying convexity relaxation.

In order to avoid obtaining degenerate or non-simple polygons, it is necessary to refine the definition of the threshold per vertex, so that we ensure that the best candidate for any given vertex v_i, will never be laying over the same edge as v_i, or causing an intersection with the boundary of the original polygon. This is achieved by limiting $\alpha+\tau$ to always be smaller than π. And this leads us to the next definition:

Def: a polygon P is said to have been split into *τ-convexity* cells, when all its vertices have at most *τ-convexity*.

The effect of increasing the internal angle of the I_i with the threshold τ, implies a larger area to look for candidates, which not only reduces the total number of cells, but also implies a reduction in the number of ill-conditioned polygons (see figure 6).

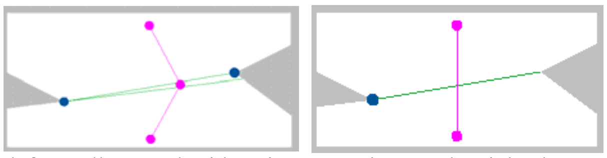


Fig. 6. On the left a cell created with strict convexity, on the right the same scenario but applying the concept of convexity relaxation.

3.4 Discussion on the Navigation Mesh Created.

Considering a polygon *P* with *r* notches, let the optimal number of convex cells, *O* be within the following bounds:

$$\left\lceil \frac{r}{2} \right\rceil + 1 \le O \le 2r + 1 \tag{3}$$

Since at most two diagonals are essential for any notch [5] then any subdivision without inessential diagonals is within four times of the minimum subdivision (which gives us the upper bound). This holds even for polygons with holes, as proved in [4]. The lower bound is given by the best case scenario where we only need one portal to join pairs of notches.

In the case of subdivision based on diagonals, there may be more than one edge created per concave vertex, since it is possible that no vertex of the geometry fall within the area of interest. If we create an edge outside this area, we will split the current concave vertex into two regions, one convex and one concave, therefore, we will still need additional edges to guarantee that all the final regions are convex.

Since our algorithm is not limited to diagonals, it only needs one new edge per concave vertex, which indicates that our number of cells will be of *O(r)*. This indicates that our method is always going to give a subdivision with fewer cells than any solution based exclusively on diagonals. In fact as we will see in the results section, our algorithm provides a subdivision of less than *r* cells.

4 Results and Future Work

The method presented in this paper generates navigation meshes that can successfully be used for path finding and driving local movement between cells. The subdivision meshes generated in all cases contain a number of cells lower than the number of notches in the geometry. Since calculating the optimal subdivision mesh for a convex polygon with holes is NP-hard, we consider that any algorithm that can guarantee a maximum number of cells equal to the number of notches can be considered a good suboptimal subdivision.

Another advantage of our method is that we obtain NavMeshes without degenerated polygons, and almost no ill-conditioned polygons. Therefore our NavMeshes can be used with any local movement technique guaranteeing natural looking movement of the characters.

Even though the time complexity of our method was not the main concern, the ANavMG can generate NavMeshes with a temporal cost of *O(r·n)*, where *r* is the number of notches, and *n* the number of vertices.

We have tested scenarios with increasing numbers of obstacles (see table 1), and the results show that for the first version of the algorithm (strict convexity rule and no elimination of old portals as new ones are generated) we achieve a ratio on average of 0.8 cells per notch. When applying the optimization of previous portal removal explained in section 3.2.3, we achieve an improvement of 10%, with the new ratio being 0.71 cells per notch. Finally the contribution based on convexity relaxation (section 3.3), achieves a ratio of 0.67 cells per notch on average, which implies an improvement of around 20% with τ-convexity=5º.

Table 1. Results from 8 scenarios with increasing number of notches. For each version of the algorithm we have calculated the number of resulting cells, and the ratio cells/notches.

Geom.	#Notches	#cells original	ratio c/r	#cells Portal removal	ratio c/r	#cells Conv. relaxation	ratio c/r
1	15	15	1.00	12	0.80	12	0.80
2	22	21	0.95	17	0.77	16	0.73
3	32	27	0.84	22	0.69	21	0.66
4	43	34	0.79	30	0.70	29	0.67
5	55	39	0.71	38	0.69	35	0.64
6	68	46	0.68	46	0.68	42	0.62
7	93	63	0.68	63	0.66	58	0.62
8	106	73	0.69	72	0.68	62	0.58
Average			0.8		0.71		0.67

As the number of vertices in the geometry increases, we observe that the ratio of cells/notches drops, since there are more chances of a vertex-vertex portal splitting simultaneously two notches, and thus reducing the number of portals per notch needed towards 0.5. We have tested scenarios of up to 136 vertices, with 106 being notches, and we believe that the average ratios calculated would be reduced even further as we test larger scenarios.

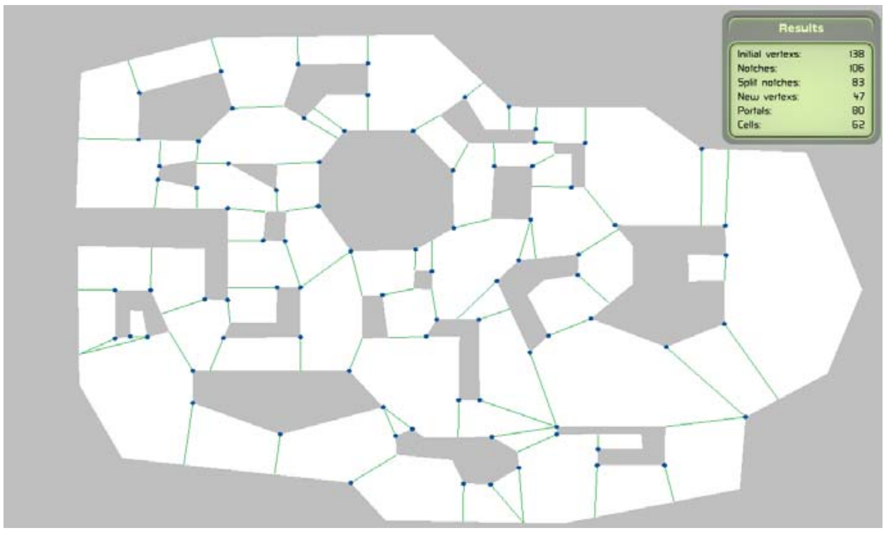
Figure 7 shows an example of a NavMesh obtained with ANavMG. (The following video *http://www.lsi.upc.edu/~npelechano/videos/MIG2011_NavMesh.avi* shows the result generated step by step as well as its associated CPG.

Game Application: Capture the Flag

Our ANavMG generator has been successfully integrated into *Ninja Flag*, a tactical multiplayer online game, inspired by the famous outdoor sport called *Capture The Flag,* that we have developed.

To determine how a character moves from one cell to another and to describe its behavior inside a convex cell, we use several steering behaviors [13]. Attractors are set based on the agents' projection over the portals, as they move within a cell. This avoids agents sharing the same attraction points when crossing and leads to natural looking movement.

Overlapping is solved by integrating the physical library *Bullet* [1]. To keep track of characters within a cell at all times, we employ *Bullet's GhostObjects*, which are special physic bodies that do not interact with the rest of the standard physics bodies of the simulation, but they track an updated list of the objects they are in contact with.



Results

Initial vertexs:	138
Notches:	106
Split notches:	83
New vertexs:	47
Portals:	80
Cells:	62

Fig. 7. Example of a NavMesh with 106 notches and 62 cells. Green lines represent portals.

5 Conclusions and Future Work

The ANavMG provides an automatic convex cells subdivision for any simple polygon with our without holes. The polygons can represent the floor plan of a given environment, with holes representing static objects such as walls. Although the current work has been tested with environments consisting of only one level, it could be expanded by considering each different level individually, and then creating cells for ramps, stairs, etc. connecting two levels following the same idea.

We have introduced a novel algorithm which focuses on the idea of sequentially splitting notches into convex areas instead of being limited to some preliminary triangle subdivision. Since our approach is based on subdividing the original polygons with segments, instead of diagonals, we achieve on average a smaller number of convex cells in the environment than previous work in the literature based on diagonals.

In this paper we have also included the concept of convexity relaxation, based on the fact that small concavities in the environment can be easily overcome with most local movement algorithms, and thus we state that for navigation meshes it is not strictly necessary to be limited to interior vertices smaller than π. We have experimentally observed that a τ-convexity of 5^o is a conservative value which works well in all the tested scenarios, however in the future we would like to further explore this idea, so that the τ-convexity can be automatically calculated, and moreover, dependent not only on the interior angle of the vertex, but also on other factors such as the length of the adjacent edges, or the type of vertices at either end of the adjacent edges. Adjusting the τ-convexity per vertex, will allow us to have even larger values for the threshold τ which will lead to an even smaller number of cells generated.

Acknowledgments. We would like to acknowledge the Spanish Ministry grant TIN2010-20590-C02-01.

References

1. Berg, J., Lin, M., Manocha, D.: Reciprocal velocity obstacles for real-time multi-agent navigation. In: ICRA'08: Proceedings of the International Conference on Robotics and Automation, pp: 1928--1935 (2008)
2. Bullet Physics Library, http://bulletphysics.org
3. Haumont, D., Debeir, O., Sillion, F.: Volumetric cell-and-portal generation. Computer Graphics Forum, vol. 22(3). pp. 303--312. (2003)
4. Fernandez, J., Toth, B., Canovas, L., Pelegrin, B.: A practical algorithm for decomposing polygonal domains into convex polygons by diagonals. TOP, vol. 16. pp. 367--387. Springer (2008)
5. Hertel, S., Mehlhorn, K.: Fast triangulation of simple polygons. In: Proc 4th International conference on Foundations of Computation Theory. Lecture LNCS, vol 158. Springer, New York, pp 207—218. (1983)
6. Kallman, M.: Navigation Queries from Triangular Meshes, In proceedings of the Third International Conference on Motion in Games (MIG), (2010)
7. Lamarche, F.: TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints. Computer Graphics Forum. vol. 28 (2), pp. 649--658. (2009)
8. Lerner, A., Chrysanthou, Y., Cohen-Or, D.: Efficient Cells-and-portals Partitioning. Computer Animation and Virtual Worlds, vol. 17(1), pp. 21--40, (2006)
9. Lien, J.-M., Amato, N.M.: Approximate convex decomposition of polygons. In: Computational Geometry, vol. 35(1-2), ACM Symposium on Computational Geometry, pp. 100--123, (2006)
10. Mekni, M. : Hierarchical path planning for situated agents in informed virtual geographic environments. In: Proc. of the 3rd International ICST Conference on Simulation Tools and Techniques. pp. 1-10, (2010)
11. Pelechano, N. and Allbeck, J. M. and Badler, N. I.: Controlling individual agents in high-density crowd simulation, In: Proc. of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 99--108 (2007)
12. Recast Toolkit, http://code.google.com/p/recastnavigation
13. Reynolds, C. W.: Steering Behaviors For Autonomous Characters. In: Game Developers Conference. San Jose. pp. 763--782. (1999)
14. Snook, G.: Simplified 3D Movement and Pathfinding Using Navigation Meshes, Game Programming Gems, Ed. Mark DeLoura, Charles River Media, (2000)
15. Sud, A., Andersen, E., Curtis, S., Lin, M.C., Manocha, D.: Real-time path planning in dynamic virtual environments using multiagent navigation graphs. In: IEEE Transactions on Visualization and Computer Graphics vol. 14, pp 526--538 (2008)
16. Unreal Engine's NavMesh Generation Method. http://udn.epicgames.com/Three/NavigationMeshReference.html
17. Valve's NavMesh Generation Method http://developer.valvesoftware.com/wiki/Navigation_Meshes