In Proceedings of the Third International Conference on Motion in Games (MIG), 2010

Navigation Queries from Triangular Meshes

Marcelo Kallmann

University of California, Merced

Abstract. Navigation meshes are commonly employed as a practical representation for path planning and other navigation queries in animated virtual environments and computer games. This paper explores the use of triangulations as a navigation mesh, and discusses several useful triangulation–based algorithms and operations: environment modeling and validity, automatic agent placement, tracking moving obstacles, ray–obstacle intersection queries, path planning with arbitrary clearance, determination of corridors, etc. While several of the addressed queries and operations can be applied to generic triangular meshes, the efficient computation of paths with arbitrary clearance requires a new type of triangular mesh, called a Local Clearance Triangulation, which enables the efficient and correct determination if a disc of arbitrary size can pass through any narrow passages of the mesh. This paper shows that triangular meshes can support the efficient computation of several navigation procedures and an implementation of the presented methods is available.

Keywords: path planning, reactive behaviors, navigation, crowd simulation.

1 Introduction

Navigation meshes are commonly used as a representation for computing navigation procedures for autonomous characters. The term navigation mesh however has been broadly used to refer to any type of polygonal mesh able to represent walkable areas in a given environment, and no specific attention has been given to establishing properties and algorithms for specific types of navigation meshes. This paper explores the use of triangulated meshes and summarizes several operations, queries and properties which are useful for the implementation of navigation strategies for autonomous agents.

One main advantage of relying on triangulations is that the obtained triangular cell decomposition of the environment has $O(n)$ cells, where n is the number of segments used to describe the obstacles in the environment. As a result, spatial processing algorithms will depend on n, which is related to the complexity of the environment (the number of edges to describe obstacles) and not to the size (or extent) of the environment. Therefore triangulated meshes are in particular advantageous for representing large environments where uniform grid–based methods become significantly slower.

The algorithms described in this paper target several navigation queries needed in applications with many agents in complex and large environments. There are two main types of obstacles which have to be addressed: 1) static objects are those describing the environment, they typically do not move over time but it is acceptable that they change their position from time to time (like a door which can be open or closed), and

2) dynamic objects are those which are continuously in movement, as for example the agents themselves.

Following the most typical approach taken with navigation meshes, the presented triangulations are considered to only represent static objects. Even if dynamic updates of obstacles are possible, specific approaches (based on triangulations or not) for handling dynamic objects will usually be more efficient. This paper addresses the use of triangulations for handling these issues and also for computing several additional navigation queries, such as ray–obstacle intersections and path planning with clearance (see Figure 1). The discussed data structures and algorithms have been implemented and are available from the author's web site[1].

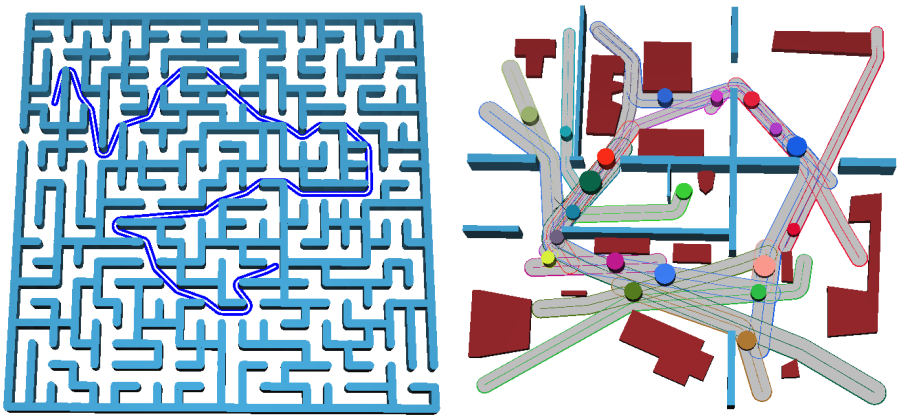

Fig. 1. Examples of several paths computed with arbitrary clearances in different environments.

2 Related Work

Triangulations are powerful representation structures which have been used in different ways for the purpose of computing navigation queries. Triangulations have in particular been used for extracting adjacency graphs for path planning from given environments. A variety of approaches have been devised, as for example to automatically extract roadmaps considering several layers (or floors) [16], and to hierarchically represent environments with semantic information and agents of different capabilities [19].

Most of the previous work on the area has however focused on specific application goals, and not on the underlying algorithms and representations. For instance, one main drawback of reducing the path planning problem to a search in a roadmap graph is that the obtained paths will still need to be smoothed. In addition, it also becomes difficult to compute paths with other useful properties, such as being optimal in length and having a given clearance from obstacles.

[1] `http://graphics.ucmerced.edu/software.html`

The most popular approach for computing the geometric shortest path among polygonal obstacles defined by n segments is to build and search the *visibility graph* [3,17] of the obstacles, what can be achieved in $O(n^2)$ time [21, 24]. The shortest path problem is however $O(n \log n)$ and optimal [11] and near-optimal [20] algorithms are available following the *continuous Dijkstra* paradigm. However, in particular when considering arbitrary clearances from obstacles, it is difficult to achieve efficient algorithms which are suitable for practical implementations. The *visibility–voronoi complex* [27] is able to compute paths in $O(n^2)$ time and is probably the most efficient implemented approach to compute paths addressing both global optimality and arbitrary clearance. It is possible to note that the use of dedicated structures is important and planar meshes have not been useful for computing globally-optimal geometric shortest paths.

Nevertheless, navigation meshes remain a popular representation in practice, and recent works have started to address properties and algorithms specifically for them. My previous work of 2003 [14] addressed the insertion and removal of constraints in a Constrained Delaunay Triangulation (CDT), and showed that environments can be well represented by CDTs for the purpose of path planning. The implementation developed in [14] has been used by other researchers [4] and significant improvements in performance were reported in comparison to grid–based methods. Extensions to the original method for handling clearance have also been reported [4], however without correctly solving the problem. In a recent publication [13], I have showed how arbitrary clearances can be properly addressed with the introduction of a new triangulation called a *Local Clearance Triangulation* (LCT), which after a precomputation of $O(n^2)$, is able to compute *locally shortest* paths in $O(n \log n)$, and even in $O(n)$ time, achieving high quality paths very efficiently. A summary of this approach is given in Section 7.

Other efficient geometric approaches are also available. In particular the approach based on corridor maps [7,8] is also able to efficiently achieve paths with clearance. One fundamental advantage of using triangulated meshes is that the environment is already triangulated, and thus channels or corridors do not need to be triangulated at every path query according to given clearances.

Path planning is not the only navigation query needed for the simulation of locomotion in complex environments. Handling dynamic agents during path execution is also important and several approaches have been proposed: elastic roadmaps [6], multi agent navigation graphs [26], etc. Avoidance of dynamic obstacles has also been solved in a reactive way, for instance with the use of *velocity obstacles* [2, 5]. Hardware acceleration has also been extensively applied [12] for improving the computation times in diverse algorithms. Although these methods are most suitable for grid–based approaches, methods for efficiently computing Delaunay triangulations using GPUs have also been developed [22].

Among the several approaches for computing navigation queries, this paper focuses on summarizing techniques which are only based on triangulations.

3 The Symedge Data Structure for Mesh Representation

The algorithms presented in this paper require a data structure able to represent planar meshes and to encode all adjacency relations between the mesh elements in constant

time. The data structure used here follows the adjacency encoding strategy of the quad–edge structure [9] and integrates adjacency operators and attachment of information per element similarly to the half–edge structure [18]. The obtained data structure is called *Symedge*, and its main element represents an oriented edge which is always symmetrical to the other oriented edge adjacent to the same edge. Oriented edges in this representation are hence called symedges, and each one will always be adjacent to only one vertex, one edge, and one face.

Each symedge keeps a pointer to the next symedge adjacent to the same face, and another pointer to the next symedge adjacent to the same vertex. The first pointer is accessed with the $nxt()$ operator and the second with the $rot()$ operator, since it has the effect of rotating around the adjacent vertex. These operators rely on a consistent counter-clockwise orientation encoding. In addition, three optional pointers are stored in each symedge for quick access to the adjacent vertex, edge and face elements, which are used to store user–defined data as needed. These pointers are accessed with operators $vtx()$, $edg()$, and $fac()$, respectively. Figure 2-left illustrates these operators.

Note that the two described primitive adjacency operators are enough for retrieving all adjacent elements of a given symedge in constant time and additional operators can be defined by composition. For instance operator $sym()$ is defined as $sym() = nxt() \rightarrow rot()$ for accessing the symmetrical symedge of a given symedge s. Inverse operators are also defined: $pri() = nxt()^{-1} = rot() \rightarrow sym()$, and $ret() = rot()^{-1} = sym() \rightarrow nxt()$. Also note that $sym()^{-1} = sym()$. Figure 2 illustrates all the mentioned element retrieval and adjacency operators.

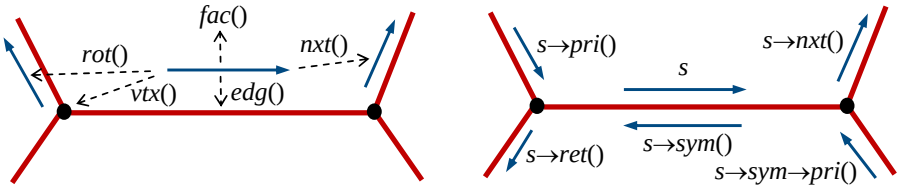

Fig. 2. Pointers are stored per symedge for fast retrieval of adjacent information (left), and several adjacency operators are defined for accessing any adjacent symedge in constant time (right).

The described symedge structure includes many additional utilities useful for the construction of generic meshes. In particular, construction operators are also included as a safe interface to manipulate the structure and Mäntylä's Euler operators [18] are implemented as the lowest–level interface. In a previous work [15], the mentioned *simplified quad–edge* structure is equivalent to the symedge structure described here. The benchmark performed in this previous work indicates that the symedge structure is among the fastest ones for describing general meshes. Although the algorithms discussed here mainly rely on triangulated meshes, using a generic structure has several advantages, in particular for the correct description of intermediate meshes during operations, and for the correct description of generic outer borders. The algorithms de-

scribed in this paper have been implemented using the described symedge data structure, which is therefore also included in the available implementation.

4 Mesh Construction and Maintenance

Let $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ be a set of n input segments describing the polygonal obstacles in a given planar environment. Segments in $\mathcal{S}$ may be isolated or may share endpoints forming closed or open polygons. The input segments are also called constraints, and the set of all their endpoints is denoted as $\mathcal{P}$.

A suitable navigation mesh can be obtained with the Constrained Delaunay Triangulation (CDT) of the input segments. Let T be a triangulation of $\mathcal{P}$, and consider two arbitrary vertices of T to be visible to each other only if the segment connecting them does not intercept the interior of any constraint.

Triangulation T will be the CDT of $\mathcal{S}$ if 1) it enforces the constraints, i.e., all segments of $\mathcal{S}$ are also edges in T, and 2) it respects the Delaunay Criterion, i.e., the circumcircle of every triangle t of T contains no vertex in its interior which is visible from all three nodes of t.

The first step to build a CDT therefore consists of identifying the segments delimiting obstacles in a given environment. Sometimes these segments will already be available, but very often designers will specify them by hand. One main difficulty in the process is that most CDT implementations will require a clean input segment set. Instead, the implemented solution chooses to handle self–intersections, overlaps, and duplicated vertices automatically as constraints are inserted in the triangulation. For example, Figure 3 shows the segments modeled by a designer to represent the walls of an apartment. The segments were intuitively organized in rectangles but with several intersections and overlapping parts. Nevertheless a correct CDT can still be obtained.

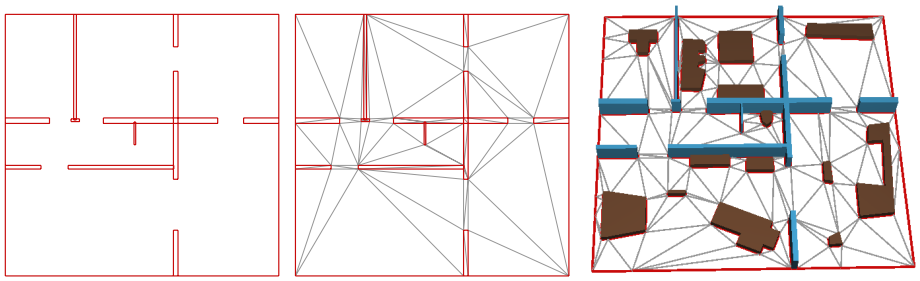

Fig. 3. Given the input set $\mathcal{S}$ of segments delimiting a given environment (left), $CDT(\mathcal{S})$ provides a suitable navigation mesh for several queries (center). Note that included validity tests [14] are able to automatically handle overlapping and intersecting constraints. Additional obstacles in the environment can also be incrementally inserted in the CDT as needed (right).

The employed corrective incremental insertion of constraints is described in a previous work [14]. Note that the alignment problem is in particular important. For instance

if two adjacent walls do not precisely share common vertices, a non–existing gap will be formed. In order to automatically detect and fix such possible gaps, the whole CDT construction uses a user–provided ϵ value and performs two specific corrective tests for each new segment inserted in the CDT: 1) if the distance between a new vertex and an existing one is smaller than ϵ, the new vertex is not inserted and instead the existing one becomes part of the input segment currently being inserted, and 2) if the distance between a new vertex and an already inserted segment (a constrained edge in the current CDT) is smaller than ϵ, the vertex is projected to the segment and its insertion precisely subdivides the segment in two collinear sub–segments. This ϵ–based approach for cleaning the input data also represents a way to improve the robustness of the geometric algorithms involved during the CDT construction. However robustness cannot be guaranteed for all types of input sets only using these two tests. Still, it seems to be possible to extend the approach to handle any possible situation. Note that this corrective approach for robustness is fundamentally different than addressing robustness only in the involved geometric computations, which is the usual approach in CDT implementations targeting mesh generation for finite element applications [23].

Given that the input segment set $\mathcal{S}$ can be correctly handled, the mesh obtained with $CDT(\mathcal{S})$ will always well conform to the obstacles. If only closed obstacles are represented, each triangle of the mesh will be either inside or outside each obstacle. Note that it is also possible to represent open polygons or simple segments, and therefore the representation is flexible to be used in diverse situations (see Figure 7). The obtained $CDT(\mathcal{S})$ is suitable for all operations described in this paper, except for the determination of paths with clearance, which will require additional properties leading to the introduction of the Local Clearance Triangulation $LCT(\mathcal{S})$, as discussed in Section 7.

Obstacles can also at any point be removed and re–inserted in $CDT(\mathcal{S})$. This is possible by associating with each inserted segment an id which can be later used to identify segments to be removed. The correct management of ids is described in detail in [14]. Removal of segments only involves local operations, however if the segment is long and connects to most of other segments in the triangulation, the removal may be equivalent to a full re–triangulation of the environment. In any case, the ability to efficiently update the position of small obstacles is often important. For example, Figure 3-right shows the apartment environment with additional obstacles representing some furniture. The ability to update the position of furniture or doors as the simulation progresses is important in many situations.

5 Agent Placement and Avoidance

Once the mesh representation of the environment is available, one particular problem which often appears is to efficiently place several agents in valid initial locations. Usual approaches will often make use of spatial subdivisions to keep track of the agents already inserted in different parts of the environment, in order to locally verify the validity of new agents being inserted. An efficient approach only based on the maintenance of a Delaunay triangulation of the agents as they are inserted is also possible.

Let r be the radius of the circle representing the agent location and let T be a Delaunay triangulation (of points) being built. First, T is initialized with (usually four)

points delimiting a region containing the entire environment, with a margin of $2r$ space. Candidate locations are then sampled at random or following any scripted distribution strategy. Each candidate location p is only inserted in T if no vertices of T are closer to p than $2r$. The triangle t containing p is then determined by efficient point location routines [14,23], and all edges around t are recursively visited for testing if their vertices respect the distance of $2r$ from p. Adjacency operators are used to recurse from the seed edges (the edges of t) to their neighbor edges, and marking of visited vertices will avoid overlaps. When all edges closer to p than $2r$ are visited with no illegal vertices found, then p is inserted as a new vertex of the triangulation and a new agent of radius r can be safely inserted at p without intersection with other agents. If the environment also has obstacles, an additional similar recursive test is performed to check if the circle centered at p with radius r does not intersect any constrained edge represented in the navigation mesh of the environment. See Figure 4 for examples.

Agents of different sizes can also be handled by storing the size of each agent in the vertices of the triangulation and performing vertex–specific distance tests during the recursive procedures described above. Alternatively, the recursive test can be avoided by first inserting each candidate point as a new vertex v in T, and then if an adjacent vertex v is too close, v is removed from T and a new candidate location is processed.

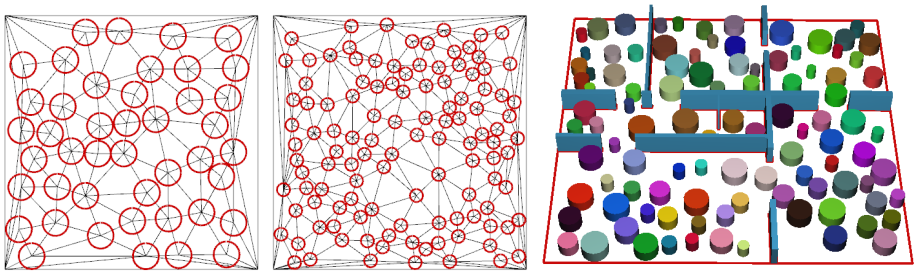

Fig. 4. The placement of non-overlapping agents with a given radius can be efficiently performed with a Delaunay triangulation tracking the inserted locations. Each location is also tested against the obstacles represented in the navigation mesh in order to achieve valid placements in the given environment.

After agents are correctly placed in valid locations navigation modules can then take control of the agents. The described Delaunay placement strategy can also be extended to efficiently perform collision avoidance strategies between the agents. One typical scenario is when agents are following given free paths in respect to the static environment while reactively avoiding the other agents on the way. For that, each agent needs to know the location of the closest agents around it at all times. The vertices of the initial Delaunay triangulation used to place the agents can then be updated as the agents move, such that each agent can quickly query the location of all agents around it. The key element of this strategy is to efficiently update the vertices of the triangulation. Fortunately there are several known algorithms able to track the position of moving vertices and only perform topological changes (of $O(n \log n)$ cost) when needed [1].

6 Ray–Obstacle Intersection Queries

Another important class of navigation queries is related to the simulation of sensors. Sensors are useful in a number of situations: for simulating laser sensors attached to robotic agents, for obtaining a simplified synthetic vision module, for querying visibility length along given directions, for aiming and shooting actions, etc. Figure 5 shows the example of a generic ray–obstacle intersection query. In this example a ray direction is given, and the ray query can be computed as follows. First, the edge e_0 first crossing the ray is determined by testing among the three edges of the triangle containing the ray source point. Then, the other edges on the next triangle adjacent to e_0 are tested for intersection and the next intersection edge e_1 is determined. The process continues until a given number of constrained edges are crossed or until a given ray length is reached. In most cases only the first crossing is important, but the algorithm can compute any number of crossings, as showed in Figure 5. Several extensions can be easily designed, for example for covering a cone sector, or a full circular region around the agent.

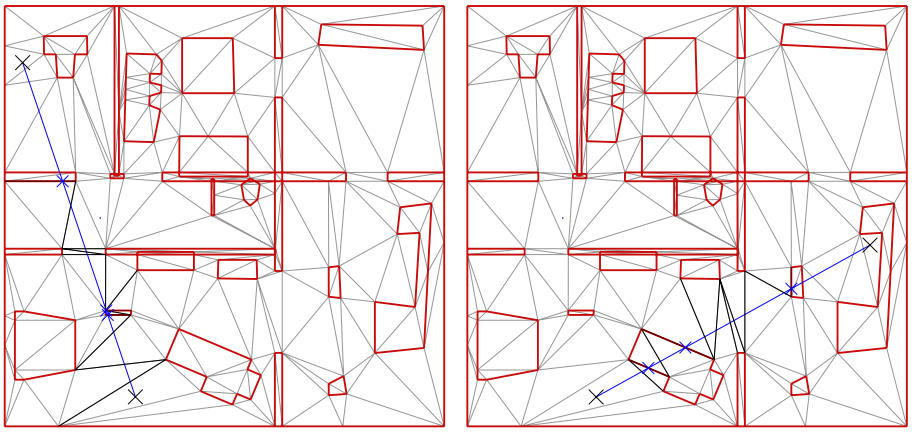

Fig. 5. In both examples, the illustrated ray–obstacle intersection query starts at the marked bottom location and identifies the first three obstacle intersections. All traversed edges are marked in black and the final top location represents the length of the query.

7 Path Planning and Paths with Clearance

Although $CDT(\mathcal{S})$ is already able to well represent environments, an additional property is required for enabling the efficient computation of paths with arbitrary clearance. This property is called the *local clearance property* [13] and will guarantee that only local clearance tests are required during the search for paths with clearance. Its construction starts with the $CDT(\mathcal{S})$, and then refinement operations are performed until

the local clearance property is enforced for all triangle traversals in the mesh. The obtained mesh is a *Local Clearance Triangulation* (LCT) of the input segments.

Once $T = LCT(\mathcal{S})$ is computed, T can be efficiently used for computing free paths of arbitrary clearance. Let $\mathbf{p}$ and $\mathbf{q}$ be two points in $\mathbb{R}^2$. A non–trivial free path between $\mathbf{p}$ and $\mathbf{q}$ will cross several triangles sharing unconstrained edges, and the union of all traversed triangles is called a *channel*. A path of r clearance is called *locally optimal* if 1) it remains of distance r from all constrained edges in T and 2) it cannot be reduced to a shorter path of clearance r on the same channel. Such a path is denoted π_r, and its channel C_r. Note that a given path π_r joining two points may or not be the globally shortest path. If no shorter path of clearance r can be found among all possible channels connecting the two endpoints, the path is then the *globally optimal* one.

The key issue for finding a path π_r is to search for a channel C_r which guarantees that there is enough clearance in all traversed triangles. A graph search over the adjacency graph of the triangulation is then performed, starting from the initial triangle containing $\mathbf{p}$, and until reaching $\mathbf{q}$. For each triangle traversed during this search, a precomputed clearance value will determine if that single triangle traversal is guaranteed to have clearance r. The refinement operations performed to build a LCT will guarantee that each traversal can be locally tested for clearance and thus enabling the precomputation of two clearance values per edge for testing the clearance of all possible triangle traversals. Figure 6 shows a typical problem which can occur in $CDTs$ but which will not occur in $LCTs$. Note that if a channel C_r is not found, the goal is not reachable.

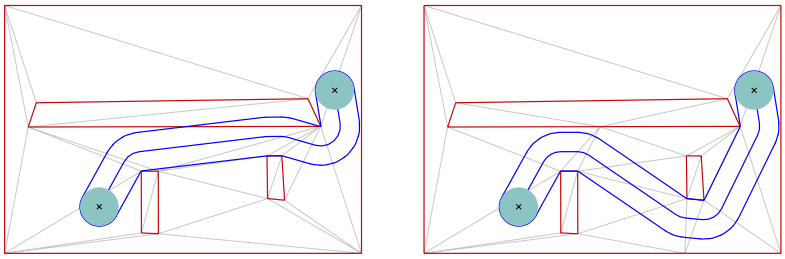

Fig. 6. Local clearance tests in CDTs cannot guarantee the correct clearance determination of paths (left). The corresponding LCT (right) will always lead to free paths with correct clearances.

Once a channel C_r of arbitrary clearance is found, its locally optimal path π_r can be computed in linear time in respect to the number of triangles in the channel. This is achieved with an extended *funnel algorithm* [10] handling clearances, which is detailed in [13].

The result is a flexible and efficient approach for path planning. The LCT can be precomputed in $O(n^2)$, and then paths of arbitrary clearance can be retrieved in $O(n \log n)$ by using a standard A* search (as implemented in [13]), or even in $O(n)$ time as the generated structure is suitable for the application of linear time planar search algorithms [25]. Figures 1 and 7 show several examples.

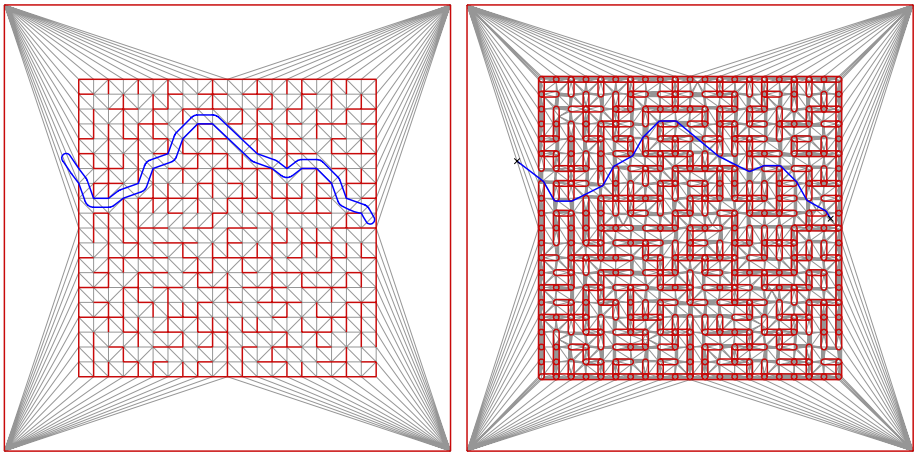

Fig. 7. The LCT of the input segments is required for computing paths with arbitrary clearance (left). Alternatively, if the clearance is constant, the environment can be inflated and paths without clearance can be extracted from the CDT of the inflated input segments (right). Also note that triangulated meshes can well represent environments described by input segments which do not form closed obstacles (left).

8 Determination of Corridors and Extensions

Note that the search for free channels (with or without clearances) during the described path planning procedure automatically determines free corridors around the computed paths. A channel C is the union of all traversed triangles and therefore the boundary of the channel will be a polygon representing a corridor containing the path. See Figure 8 for an example. Figure 8 also illustrates the computation of *extra clearances*, which deform the path in order to achieve higher clearance than the minimum required. Extra clearances can be computed with post-optimization of obtained paths and can model a variable range of locomotion behaviors, from attentive in passages with minimum clearance to safe navigation in higher clearance areas.

Many other extensions can be devised. For instance the corridor search procedure can be optimized (significantly in certain cases) by introducing a smaller connectivity graph which excludes the triangles inside corridors, which are those that have two constrained edges and thus have only one way of being traversed. Hierarchical representations of several levels (common in grid–based approaches) can also be translated to triangle meshes.

Note that efficient path planning queries are also important for decision modules. For example, the ability to query goal reachability with different clearances and to compute lengths of obtained paths may be important for deciding which target locations to visit first in case of several choices. Many other uses of the proposed methods exist. For instance, the handling of intersections and overlaps in the input segment set can be used to perform Boolean operations with obstacles, what is useful for optimizing the navigation mesh. Finally, one important extension in many applications is the ability

to model uneven terrains. Due to their irregular decomposition nature, triangulations are well suited for the representation of terrains, however each geometric test in the described procedures would have to be generalized for handling non-planar surfaces.

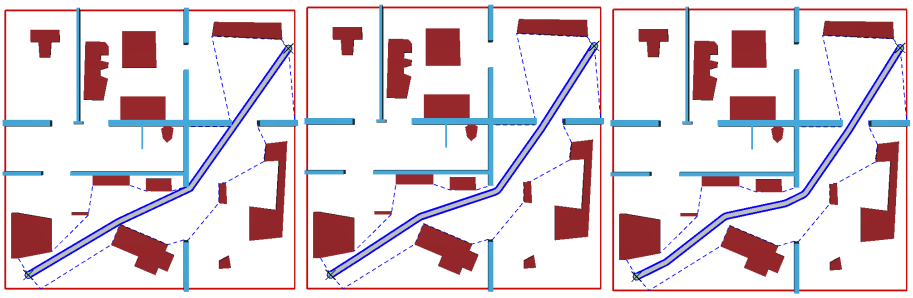

Fig. 8. Three paths with same minimum clearance but with extra clearances of 0, 0.45, and 0.9.

9 Final Remarks

This paper presented several triangulation–based methods for the efficient computation of diverse navigation queries for autonomous agents. With the growing research activity in the area and the appearance of several new development tools, triangulation–based navigation meshes can be expected to become increasingly popular.

References

1. Albers, G., Mitchell, J.S., Guibas, L.J., Roos, T.: Voronoi diagrams of moving points. Internat. J. Comput. Geom. Appl 8, 365–380
2. 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 (2008)
3. De Berg, M., Cheong, O., van Kreveld, M.: Computational geometry: algorithms and applications. Springer (2008)
4. Demyen, D., Buro, M.: Efficient triangulation-based pathfinding. In: AAAI'06: Proceedings of the 21st national conference on Artificial intelligence. pp. 942–947. AAAI Press (2006)
5. Fiorini, L.P., Shiller, Z.: Motion planning in dynamic environments using velocity obstacles. International Journal of Robotics Research 17(7), 760–772 (1998)
6. Gayle, R., Sud, A., Andersen, E., Guy, S.J., Lin, M.C., Manocha, D.: Interactive navigation of heterogeneous agents using adaptive roadmaps. IEEE Transactions on Visualization and Computer Graphics 15, 34–48 (2009)
7. Geraerts, R.: Planning short paths with clearance using explicit corridors. In: ICRA'10: Proceedings of the IEEE International Conference on Robotics and Automation (2010)
8. Geraerts, R., Overmars, M.H.: The corridor map method: a general framework for real-time high-quality path planning: Research articles. Computer Animation and Virtual Worlds 18(2), 107–119 (2007)

9. Guibas, L., Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of voronoi. ACM Trans. Graph. 4(2), 74–123 (1985)
10. Hershberger, J., Snoeyink, J.: Computing minimum length paths of a given homotopy class. Computational Geometry Theory and Application 4(2), 63–97 (1994)
11. Hershberger, J., Suri, S.: An optimal algorithm for euclidean shortest paths in the plane. SIAM Journal on Computing 28, 2215–2256 (1997)
12. Hoff, III, K.E., Culver, T., Keyser, J., Lin, M., Manocha, D.: Fast computation of generalized voronoi diagrams using graphics hardware. In: Proceedings of the sixteenth annual symposium on computational geometry (2000)
13. Kallmann, M.: Shortest paths with arbitrary clearance from navigation meshes. In: Proceedings of the Eurographics / SIGGRAPH Symposium on Computer Animation (SCA) (2010)
14. Kallmann, M., Bieri, H., Thalmann, D.: Fully dynamic constrained delaunay triangulations. In: Brunnett, G., Hamann, B., Mueller, H., Linsen, L. (eds.) Geometric Modeling for Scientific Visualization, pp. 241–257. Springer-Verlag, Heidelberg, Germany (2003), iSBN 3-540-40116-4
15. Kallmann, M., Thalmann, D.: Star vertices: A compact representation for planar meshes with adjacency information. Journal of Graphics Tools 6(1), 7–18 (2001)
16. Lamarche, F.: TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints. Computer Graphics Forum 28 (03 2009)
17. Lozano-Pérez, T., Wesley, M.A.: An algorithm for planning collision-free paths among polyhedral obstacles. Communications of ACM 22(10), 560–570 (1979)
18. Mäntylä, M.: An introduction to solid modeling. Computer Science Press, Inc., New York, NY, USA (1987)
19. Mekni, M.: Hierarchical path planning for situated agents in informed virtual geographic environments. In: SIMUTools '10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. pp. 1–10 (2010)
20. Mitchell, J.S.B.: Shortest paths among obstacles in the plane. In: SCG '93: Proceedings of the ninth annual symposium on computational geometry. pp. 308–317. ACM, New York, NY, USA (1993)
21. Overmars, M.H., Welzl, E.: New methods for computing visibility graphs. In: SCG '88: Proceedings of the fourth annual symposium on Computational geometry. pp. 164–171. ACM (1988)
22. Rong, G., seng Tan, T., tung Cao, T.: Computing two-dimensional delaunay triangulation using graphics hardware. In: In Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D) (2008)
23. Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin, M.C., Manocha, D. (eds.) Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science, vol. 1148, pp. 203–222. Springer-Verlag (May 1996), from the First ACM Workshop on Applied Computational Geometry
24. Storer, J.A., Reif, J.H.: Shortest paths in the plane with polygonal obstacles. J. ACM 41(5), 982–1012 (1994)
25. Subramanian, S., Klein, P., Klein, P., Rao, S., Rao, S., Rauch, M., Rauch, M.: Faster shortest-path algorithms for planar graphs. In: Journal of Computer and System Sciences. pp. 27–37 (1994)
26. Sud, A., Andersen, E., Curtis, S., Lin, M.C., Manocha, D.: Real-time path planning in dynamic virtual environments using multiagent navigation graphs. IEEE Transactions on Visualization and Computer Graphics 14, 526–538 (2008)
27. Wein, R., van den Berg, J.P., Halperin, D.: The visibility–voronoi complex and its applications. In: SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry. pp. 63–72. ACM, New York, NY, USA (2005)