Game-Changing: Fast Dynamic Updates in a Flexible Navigation Mesh

Wouter G. van Toll[1], Atlas F. Cook IV[2], and Roland Geraerts[1]

[1]Department of Information and Computing Sciences, Utrecht University
[2]Institute for Computational Engineering and Sciences, University of Texas at Austin

Abstract

Games and simulations frequently model scenarios where obstacles move, appear, and disappear in an environment. A city environment changes as new buildings and roads are constructed, and routes can become partially blocked by small obstacles many times in a typical day. This paper studies the effect of using *local* updates to repair only the affected regions of a navigation mesh in response to a change in the environment. The techniques are inspired by incremental methods for Voronoi diagrams. The main novelty of this paper is that we show how to maintain a 2D or 2.5D *navigation mesh* in an environment that contains dynamic polygonal obstacles. Experiments show that local updates are fast enough to permit real-time updates of the navigation mesh.

Keywords: navigation mesh, dynamic environments, medial axis, Voronoi diagram, virtual worlds, path planning.

This paper has been published previously in the *Computer Animation and Virtual Worlds* journal [32].

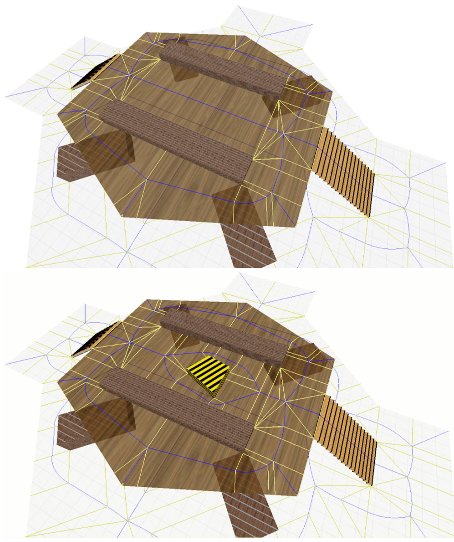


Fig. 1. The *Layers* environment. The obstacle can be locally inserted onto the top floor of a 2.5D multi-layered environment in $1.1ms$. Ten new vertices were added to the navigation mesh when the obstacle was inserted.

1 Introduction

A *navigation mesh* is a data structure that uses a set of two-dimensional regions to represent the walkable space in an environment [29]. These regions are commonly used to plan visually convincing paths through complicated environments [7, 19, 23, 24, 31].

Current environments are largely static because it is a relatively expensive operation to recompute the entire navigation mesh each time the environment changes. The goal of this paper is to show that a navigation mesh that is based on a medial axis can be updated very quickly. We locally repair only the affected regions of the navigation mesh each time the environment changes. Our experiments show that local operations can be performed quickly enough to support dynamic environments where many obstacles are frequently moved, added, and removed.

The navigation mesh that we choose to locally repair is the *Explicit Corridor Map (ECM)* [7, 31]. The ECM was chosen because it can quickly produce smooth and short paths with any feasible amount of clearance to the obstacles. This navigation mesh can be used to plan paths for characters that may have a variety of widths and clearance preferences. The ECM has a small memory footprint and has previously been used to plan visually convincing paths for thousands of virtual characters in real-time [33].

To the best of our knowledge, this paper is the first to describe how to perform dynamic updates in an exact 2.5D navigation mesh. As illustrated in Fig. 1, such a mesh describes the walkable areas for a connected set of 2D floors. Such a multi-layered structure can be used to model buildings with multiple floors.

1.1 Related Work on Static Environments

Most path planners assume that the obstacles in an environment are fixed. This means that the environment is *static*. Graph-based techniques such as probabilistic roadmaps [17], rapidly-exploring random trees [18], waypoint graphs [26], and reactive deformation roadmaps [6] represent the walkable environment (or a high-dimensional configuration space) using a set of *one-dimensional* edges. By contrast, a *navigation mesh* partitions the walkable environment into a set of *two-dimensional* walkable areas. These walkable areas permit virtual characters to control their movements inside each two-dimensional region so that they can more easily avoid other moving characters [1, 12, 16].

There are many techniques to construct navigation meshes. Pettré et al. [24] use a set of overlapping disks to describe the walkable space. Mononen's [19] open-source *Recast Navigation* project discretizes the environment into cubic voxels, extracts the walkable surfaces, and connects all adjacent cells. Hale et al. [4] seed the environment with a series of quads, and each quad grows until it is as large as possible. These techniques approximate the walkable space, so small geometric details might be lost.

Several *exact* approaches exist to construct navigation meshes. Kallmann [13] uses a special *triangulation* to construct walkable areas in $O(n \log n)$ time. The amount of clearance along a path in this triangulation is based on the radius of the largest empty disk along the path. Such a triangulation has linear complexity and encodes clearance information for many points in the environment. This technique currently supports triangulations in 2D environments, but it could be easily extended to support multi-layered environments. Wein et al. [34] combine visibility graph and Voronoi diagram techniques with an $O(n^2 \log n)$ time approach. This powerful technique encodes clearance information for all points in the environment and produces global shortest paths. It is relatively expensive to compute and is intended for static 2D environments.

The *medial axis* is the set of all points in the environment that are equidistant from at least two distinct closest obstacle points in the environment [25]. Geraerts [7] uses an augmented medial axis called the Explicit Corridor Map to partition a two-dimensional environment into a set of walkable areas in $O(n \log n)$ time. This work was extended to deal with multi-layered environments [31]. An advantage of this structure is that it naturally encodes clearance information for all points in the environment. It also allows efficient local updates. A major goal of this paper is to describe how the augmented medial axis of [7] and [31] can be quickly updated each time an obstacle is inserted, deleted, or moved.

1.2 Related Work on Dynamic Environments

Some data structures can handle obstacles that change over time. The *adaptive roadmaps* of Sud et al. [30] contain elastic edges that can change along with the environment. Roos and Noltemeier [27] have augmented a structure with time information to enable continuous updates in an environment with moving points. Kallmann and Matarić [15] describe *dynamic roadmaps* that keep track of the obstacles in the environment and constantly update a graph.

Green and Sibson [9] show how to locally update a two-dimensional Voronoi diagram each time a *point obstacle* is inserted. Devillers [3], Mostafavi et al. [20], and Gowda et al. [8] all consider how to delete *point obstacles* from a Voronoi diagram. Held and Huber [10] show how to insert polygons and circular arcs into a Voronoi diagram. Kallmann et al. [14] describe how to insert or remove obstacles from a triangular mesh. Concurrently with our work, de Pinto Moura and Dal Sasso Freitas [21] have implemented insertions and deletions for Voronoi diagrams of complex sites. Our results are similar, but more application-oriented.

Since there has not been much previous work on maintaining a multi-layered navigation mesh in a dynamic setting, the focus of this paper is to describe how to locally repair an augmented medial axis each time an obstacle is inserted, deleted, or moved. Our experiments show that these local operations take only a few milliseconds to perform in practice. Hence, dynamic obstacles can be used in real-time applications.

1.3 Overview

This paper is organized as follows. Section 2 reviews fundamental data structures such as the medial axis. The medial axis is useful because it can easily be annotated with nearest obstacle information. Such an augmented medial axis defines a navigation mesh called the Explicit Corridor Map [7]. Section 3 shows how to locally *insert* a point, line segment, or polygonal obstacle into a 2D augmented medial axis. Section 4 describes how to locally *delete* any polygonal obstacle from a 2D augmented medial axis. Section 5 shows how to insert and delete obstacles into a *2.5D* multi-layered augmented medial axis. The experimental results in Section 6 show that an augmented medial axis can be locally repaired in just a few milliseconds each time an obstacle is inserted, deleted, or moved.

2 Preliminaries

Throughout this paper, we assume that all polygonal obstacles in the environment have been partitioned into convex parts.

Consider a set of m (convex) polygonal obstacles $\{p_1, ..., p_m\}$ in the plane. Let n be the total number of vertices defined by these obstacles. The *Voronoi diagram* of these obstacles is a partition of the plane into a set of two-dimensional *cells* $\{C_1, ..., C_m\}$. The cells are constructed such that each point in the cell C_i is nearer to the obstacle p_i than to any other obstacle $p_{j\neq i}$. The boundary of a cell C_i is denoted by ∂C_i. As shown in Fig. 2a, a cell in the Voronoi diagram can have disconnected components.

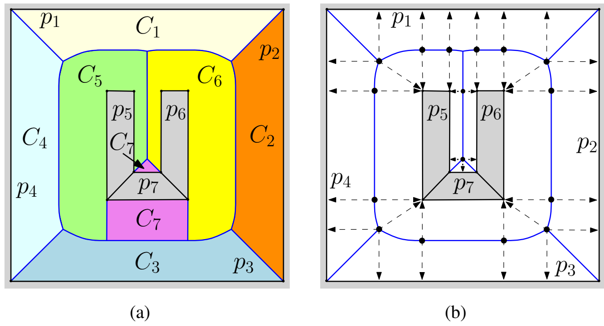


Fig. 2. (a) The Voronoi diagram partitions the environment into a collection of two-dimensional cells $C_1, ..., C_7$. All points in C_i are closer to the polygonal obstacle p_i than to any other obstacle $p_{j\neq i}$. Obstacles are shown in gray. (b) The medial axis of a two-dimensional environment is shown in blue. The dashed line segments are closest point edges that make it easy to determine the nearest obstacle.

Each edge in the Voronoi diagram is a *bisector* that is equidistant from at least two closest obstacles. In a polygonal environment, such a bisector is composed of one or more line segments and parabolic arcs [2, 22].

A concept that is closely related to the Voronoi diagram is the *medial axis*. The medial axis of a set of (convex) polygonal obstacles $\{p_1, ..., p_m\}$ is the set of all bisectors that are equidistant from at least two *distinct* closest obstacle points in the environment [25]. Fig. 2b illustrates the medial axis of a two-dimensional environment that contains four line segment obstacles $p_1, ..., p_4$ and three convex polygonal obstacles p_5, p_6, p_7. Notice that the edges of the medial axis are a subset of the edges of the Voronoi diagram.

Since the medial axis partitions the environment into a set of two-dimensional walkable areas, it is possible to transform the medial axis into a navigation mesh by adding $O(n)$ *closest point edges*. A closest point edge is a line segment that connects the medial axis to a closest obstacle point. As in Fig. 2, closest point edges are created at all medial axis vertices that do not intersect an obstacle. These edges make it easy to determine the nearest obstacle point to any query point. The resulting augmented medial axis is a navigation mesh. It is well-suited for computing minimum clearance paths that stay as far away from obstacles as possible [7].

A variety of exact approaches exist to construct the medial axis. Green and Sibson [9] build the medial axis for a set of point obstacles in the plane by incrementally adding one obstacle at a time and updating the data structure at each step. Shamos and Hoey [28] describe a divide-and-conquer technique that recursively splits the obstacles into two groups, computes the medial axis for each group, and merges these two groups together. Fortune's [5] *sweep line* algorithm sweeps a line over the plane and maintains the medial axis behind this sweep line. Hoff et al. [11] show how to *approximate* the medial axis by using graphics hardware to project distance functions onto an orthogonal plane.

Most previous work describes how to maintain a *2D medial axis* in environments with dynamic *point* obstacles. The main novelty of this paper is that we show how to maintain a 2D or 2.5D *navigation mesh* in an environment that also contains dynamic *polygonal* obstacles.

3 Inserting an Obstacle into a 2D Navigation Mesh

This section describes how to efficiently insert a point, line segment, or convex polygonal obstacle into a two-dimensional navigation mesh. We first describe how to insert *point obstacles* into a navigation mesh that only contains point obstacles. Next, we give a detailed insertion algorithm for inserting points into a navigation mesh that contains polygonal obstacles. This algorithm is then refined so that *line segments* and *convex polygonal obstacles* can also be inserted.

These algorithms all work by updating the medial axis of the environment. Each *edge* in the medial axis is associated with a nearest obstacle. Each *vertex* in the medial axis stores a set of closest point edges. As shown in Fig. 2, these closest point edges connect each vertex in the medial axis to all of its nearest obstacle points. Note that the closest point edges for any vertex can be computed by using the medial axis edges to determine the nearest line segment obstacles to this vertex. Given these line segment obstacles, we can then easily determine the closest point on each of these line segments to the current medial axis vertex.

3.1 Inserting a Point into an Environment with Point Obstacles

Intuitively, we can insert a point p into a navigation mesh that contains only point obstacles as follows. We construct a cell for this new point, insert this cell into the underlying medial axis, and update the closest obstacle information in the navigation mesh. This approach has previously been used to incrementally construct a Voronoi diagram of *points* [9, 22]. The following steps describe this approach in more detail:

(1) Find a cell C_j that intersects the new point p. This cell identifies a nearest obstacle p_j to p. Please refer to Fig. 3a.

(2) Calculate the bisector of p and p_j. Let i_1 and i_2 be the two intersection points of this bisector with the boundary of the cell C_j.

(3) The bisector from i_1 to i_2 is the first edge of the new cell C_p for p. At i_2, the bisector runs into an adjacent cell, say C_k. This cell identifies a nearest obstacle p_k to the point i_2. Calculate the bisector of p and p_k, and let the intersections of this bisector with the boundary of C_k be i_2 and i_3 (see Fig. 3b). Repeat this process to determine points i_3, i_4, and so on, until a bisector endpoint is found that returns to i_1. The resulting closed loop of bisectors will define the boundary of the new cell C_p (see Fig. 3c).

(4) Deleting all vertices and edges in C_p will finalize the insertion of p into the medial axis (see Fig. 3d).

It takes $O(x+\log n)$ time to insert a single point. Here, $x \in O(n)$ is the number of edges that are updated during the algorithm, and n is the total number of obstacle vertices in the environment. The $O(\log n)$ term is required to find a cell that intersects the new point p in step 1.

3.2 Inserting a Point into a Polygonal Environment

We are now ready to describe a detailed algorithm that updates a navigation mesh each time an obstacle is inserted into a *polygonal* environment. We start by describing how to insert a point obstacle into a polygonal environment. We then describe how to insert a line segment obstacle and a polygonal obstacle into a polygonal environment.

Algorithm 1 describes how to add a point obstacle to a polygonal environment. To compute the new cell for the point that is being inserted, we will iteratively construct a sequence of bisectors around the newly added point. Algorithm 2 contains a subroutine named GETBISECTORARC that calculates the next bisector arc that

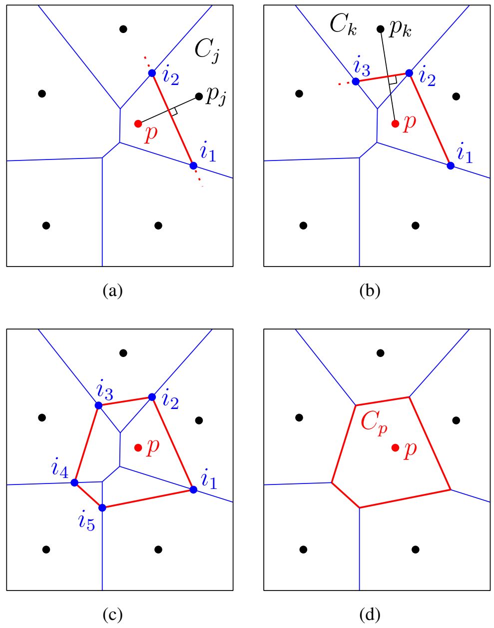


Fig. 3. A new point obstacle p is inserted by computing a sequence of bisectors that form a closed loop. (a) The bisector of p and p_j intersects the boundary of C_j at two points i_1 and i_2. (b) The bisector of p and p_k intersects the boundary of C_k at two points i_2 and i_3. (c) Eventually, the bisectors form a closed loop around the new obstacle p. (d) The new cell C_p is the region inside this closed loop.

is needed. Algorithm 3 contains a subroutine named TRACEBISECTOR that calculates the first intersection of a directed sequence of bisectors with the boundary of an existing cell.

These algorithms assume that a point location function named CLOSESTOBSTACLE is available. Given any point in the environment, CLOSESTOBSTACLE returns the closest convex polygonal obstacle, the closest line segment on this obstacle, and the closest point on this line segment. Note that when the bisector crosses an edge of the medial axis, new closest-obstacle information can be derived from the data structure in constant time. Hence, only the first call to CLOSESTOBSTACLE takes $O(\log n)$ time. For further information on the basic concept of point location, we refer the interested reader to a popular book [2].

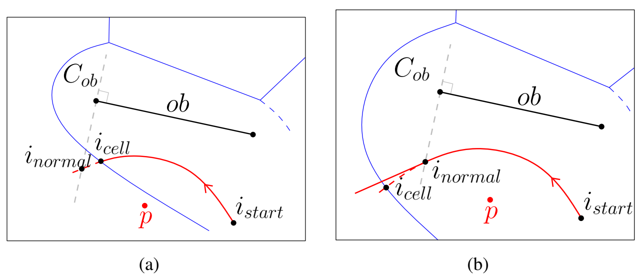


Fig. 4. A red bisector is computed for a point p and an obstacle ob. Edges of the medial axis – before inserting p – are shown in blue. (a) If the point i_{cell} precedes i_{normal} on the directed bisector, then we have a *bisector vertex* at i_{cell}. This follows because there are at least three obstacles (including p) that are equidistant to i_{cell}. (b) If the point i_{normal} precedes i_{cell} on the bisector, then we have a bisector vertex at i_{normal}. This follows because a new vertex or line segment of ob can begin inducing the bisector at some point in the interior of C_{ob}.

The subroutine TRACEBISECTOR iteratively computes the directed sequence of arcs for the bisector $B_{p,ob}$ in *one* direction until the bisector intersects the cell boundary ∂C_{ob}. Note that the bisector $B_{p,ob}$ of a point p and a convex polygonal obstacle ob is a sequence of line segments and parabolic arcs. As shown in Fig. 4a, a *bisector vertex* can appear on $B_{p,ob}$ at a point i_{cell} where the bisector

Algorithm 1 ADDPOINT(p)
Input: A new point obstacle p that should be inserted into a polygonal environment.
Output: The updated navigation mesh after inserting p.

{The bisector $B_{p,ob}$ of a point p and a convex polygonal obstacle ob is a sequence of line segments and parabolic arcs. We first compute the intersection of the cell boundary ∂C_{ob} with this bisector.}

$(ob, obSeg, obPt) \leftarrow$ CLOSESTOBSTACLE(p)
$b \leftarrow$ GETBISECTORARC(p, ob, $obSeg$, $obPt$)
$(i_1, i_2) \leftarrow$ the two intersection points in $b \cap \partial C_{ob}$
$m \leftarrow$ the midpoint of the bisector arc from i_1 to i_2
$arcsR \leftarrow$TRACEBISECTOR(p,ob,$obSeg$,$obPt$,b,m,i_1)
$arcsL \leftarrow$TRACEBISECTOR(p,ob,$obSeg$,$obPt$,b,m,i_2)
Reverse the sequence of bisector arcs in $arcsR$.
$e \leftarrow$ the sequence of bisector arcs in $arcsR$ and $arcsL$
$(i_1, i_2) \leftarrow$ the endpoints of e

{The bisector $B_{p,ob}$ enters a new cell at i_2. We now repeatedly compute the intersection of the current cell with the current bisector.}

$i_{prev} \leftarrow i_1$; $i_{curr} \leftarrow i_2$; $i_{next} \leftarrow$ NIL
while $i_{next} \neq i_1$ **do**
  $(ob, obSeg, obPt) \leftarrow$ CLOSESTOBSTACLE(i_{curr})
  $b \leftarrow$ GETBISECTORARC(p, ob, $obSeg$,$obPt$)
  $i_{next} \leftarrow$ the endpoint of $b \cap \partial C_{ob}$ that is not i_{curr}
  $arcsL \leftarrow$ TRACEBISECTOR(p,ob,$obSeg$,$obPt$,b,i_{prev},i_{next})
  $e \leftarrow$ the sequence of bisector arcs in $arcsL$
  $i_{next} \leftarrow$ the endpoint of e that does not equal i_{curr}
  $i_{prev} \leftarrow i_{curr}$; $i_{curr} \leftarrow i_{next}$

{The boundary of the new cell for p is now complete.}
Remove the mesh edges that lie inside this new cell.
Update closest point information for all modified cells.

Algorithm 2 GETBISECTORARC(p, ob, $obSeg$,$obPt$)
Input: A new point p that should be inserted into a polygonal environment, plus the closest convex obstacle ob to p, the closest line segment obstacle $obSeg$ to p, and the closest point obstacle $obPt$ on ob to p.
Output: Returns the directed bisector arc for p and the current closest obstacle.

if ob is a point **then**
  return the line segment bisector of p and $obPt$
else
  if $obPt$ is an endpoint of $obSeg$ **then**
    return the line segment bisector of p and $obPt$
  else
    return the parabolic bisector of p and $obSeg$

intersects the *boundary* of C_{ob}. Note that the point i_{cell} is a bisector vertex because i_{cell} has at least three closest obstacles (including the new obstacle p).

As shown in Fig. 4b, a bisector vertex can also appear at a point i_{normal} because a new vertex or line segment of the polygon ob starts inducing the bisector $B_{p,ob}$ at some point in the *interior* of C_{ob}. The position of i_{normal} is simply the first intersection of the current directed bisector arc with the surface normals through the endpoints of the closest line segment of ob, i.e. $obSeg$.

To determine the next bisector vertex, TRACEBISECTOR repeatedly determines the positions of both i_{cell} and i_{normal}. The intersection that occurs first along the directed bisector arc becomes the endpoint of the current arc, until an instance of i_{cell} is chosen and the bisector is completed. Algorithms 1–3 are used to insert a point obstacle into a polygonal environment.

3.3 *Inserting a Line Segment or Polygon into a Polygonal Environment*

The primary difference between adding a *point* to an environment and adding a *line segment* to an environment is that the normals through the endpoints of the line segment that is being inserted can generate bisector vertices. This means that the TRACEBISECTOR subroutine should consider *three* candidate bisector endpoints at each step. As before, a bisector vertex can occur when a bisector

Algorithm 3 TRACEBISECTOR (p, ob, $obSeg$, $obPt$, b, i_{start}, i_{cell})
Input: A new point obstacle p, closest obstacle information for p, the directed bisector arc b to trace, the start point i_{start} of the bisector arc, and the first intersection point i_{cell} of the directed bisector b with ∂C_{ob}.
Output: A sequence of bisector arcs from i_{start} to ∂C_{ob}.

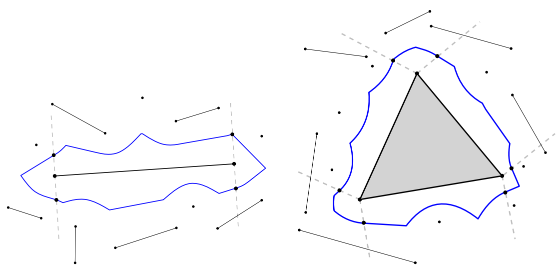
$result \leftarrow$ empty list that will store bisector arcs
$finished \leftarrow$ **false**
while $finished =$ **false do**
  Ensure that the first endpoint of b is i_{start} and the second endpoint of b is i_{cell}.
  if b intersects the normals through the endpoints of $obSeg$ **then**
    $i_{normal} \leftarrow$ the first intersection of b with the normals through the endpoints of $obSeg$
  else
    $i_{normal} \leftarrow$NIL
  if i_{cell} precedes i_{normal} along b **then**
    {The bisector intersects the boundary of the cell.}
    $result$.add(i_{cell})
    $finished \leftarrow$ **true**
  else {The current arc has an endpoint inside the cell.}
    $result$.add(i_{normal})
    $(obSeg, obPt) \leftarrow$ CLOSESTOBSTACLE(i_{normal}, ob)
    $i_{start} \leftarrow i_{normal}$
    $b \leftarrow$ GETBISECTORARC(p, ob, $obSeg$, $obPt$)
    $i_{cell} \leftarrow$ the first intersection of b with ∂C_{ob}
return $result$

arc intersects the boundary of a cell or when a bisector arc intersects a surface normal through an endpoint of the current closest obstacle. The new scenario is that a bisector vertex can now also occur when a bisector arc intersects a surface normal through a vertex of the line segment that is being inserted. Please refer to Fig. 5a.

The GETBISECTORARC subroutine should consider the 'active part' of the line segment that is being inserted. For example, when drawing the portion of the bisector in between the surface normals through the new segment's endpoints, we need to return the bisector of a *line segment* and the closest obstacle. By contrast, when drawing the remainder of the bisector, we need to return the bisector between an *endpoint* and the closest obstacle.

Note that a line segment that passes through existing obstacles can be added by partitioning the line segment into pieces that do not intersect any obstacle. Each of these pieces can easily be added to the environment.

A convex polygon can be added by sequentially inserting each of its line segments into the environment (and removing the medial axis inside this polygon). The 'active part' of the new polygon that is currently generating the bisector changes whenever a bisector arc crosses one of the surface normals that passes through a vertex of the polygon. See Fig. 5b.

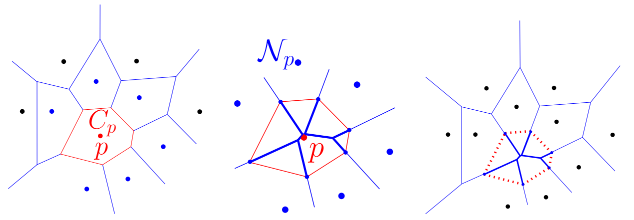


(a) (b)

Fig. 5. A line segment obstacle or a polygonal obstacle can be inserted into the medial axis. The gray dashed-lines are the normals that pass through the vertices of the obstacle that is being inserted.

4 Deletions in a 2D Mesh

To delete an obstacle p from a two-dimensional navigation mesh, we only need to update the mesh inside the cell C_p that contains p. This follows because (a) medial axis edges are defined exclusively by bisectors between pairs of obstacles and (b) only bisectors induced by p are affected by the deletion operation.



(a) (b) (c)

Fig. 6. (a) To delete an obstacle p, identify the cell C_p that contains p, and determine the set $\mathcal{N}_p$ of (blue) neighbor obstacles whose cells are adjacent to C_p. (b) Compute the medial axis of the neighbor obstacles in $\mathcal{N}_p$, and keep just the thickened edges inside the old cell C_p. (c) Add these thickened edges to the medial axis, and delete the edges that bound the old cell C_p.

As depicted in Fig. 6, let $\mathcal{N}_p$ be the set of all *neighbor obstacles* whose cells are adjacent to C_p. The obstacle p can be deleted by computing the medial axis of the neighbor obstacles in $\mathcal{N}_p$ and intersecting this augmented medial axis with the old cell C_p. The edges in this intersection (including their associated closest obstacle information) should be added to the medial axis, and the old boundary edges of C_p should be removed from the medial axis.

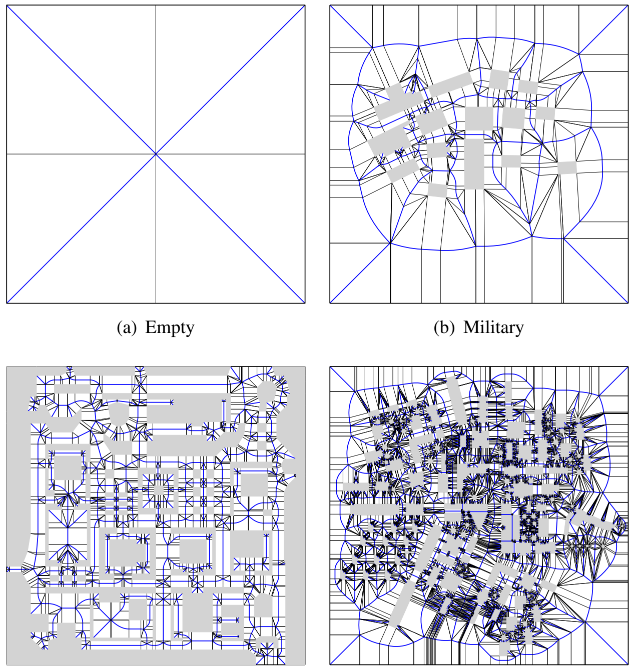
Notice the strictly local nature of this approach. If the number of total vertices defined by the neighbor obstacles in $\mathcal{N}_p$ is $x \in O(n)$, then a deletion takes $O(x \log x)$ time when using an exact construction algorithm.

5 Dynamic Multi-Layered Mesh

This section describes how to insert and delete obstacles in a multi-layered environment. Such an environment consists of a set of *layers* (2D environments) plus a set of *connections* between the layers. An algorithm has been previously developed that builds the medial axis and a navigation mesh of such an environment [31]. Fig. 1 illustrates the result of inserting a polygonal obstacle into a multi-layered navigation mesh.

To insert an obstacle into a multi-layered environment, we need to create a new cell for this obstacle. The main difference from the 2D setting is that the new cell can now exist in multiple layers. Consequently, the insertion algorithm must detect when the current bisector arc crosses a connection.

To delete a polygonal obstacle p from a multi-layered environment, we identify the cell C_p that contains p and compute the set $\mathcal{N}_p$ of all neighbor obstacles whose cells are adjacent to C_p. Although the neighbor obstacles may lie in different layers, a 2D algorithm can be used to approximate the medial axis of the neighbor obstacles. As in [31], we project all neighbor obstacles onto a plane that contains p and compute the medial axis (including the closest point edges) of these projected neighbor obstacles. We then project the (multi-layered) cell C_p onto this same plane and keep only the edges that lie inside the projected cell C_p. These edges are added to the multi-layered medial axis. The boundary of the old cell C_p is then pruned from the medial axis.



(a) Empty (b) Military

(c) Zelda (d) City

Fig. 7. Each environment has a physical size in meters, a rendering resolution of 1000x1000 pixels, and a number of convex primitives (after partitioning all non-convex objects into convex components). The blue medial axis and the black closest point edges together define a navigation mesh for each environment.

6 Experimental Results

Five environments are used in our experiments. As illustrated in Fig. 7, and Table 1, the *Empty* environment represents the simplest type of scene because it is simply a bounding box that contains no additional obstacles. The *Military* environment represents the McKenna military training site at Fort Benning, Georgia, USA. The *Zelda* environment is from a popular video game, and the *City* environment represents a large and complicated scene with many obstacles. The *Layers* environment in Fig. 1 depicts part of a multi-storey building that contains two layers connected by two staircases modelled as two more layers. Our experiments measure the effect of inserting, deleting, and moving polygonal obstacles in these environments.

Table 1. Five Experimental Environments

	Environment		Navigation Mesh		
Name	Size(m)	vertices	vertices	Edges	Time(ms)
Empty	100x100	8	5	4	10
Military	200x200	104	56	70	23
Zelda	100x100	560	287	342	35
City	500x500	2638	1447	1639	78
Layers	100x100	213	43	51	15

The experiments were performed in Visual C++ on an NVIDIA GT 240 graphics card and an Intel Core2 Duo CPU (3.0 GHz) with 4 GB memory. Only one core was used.

6.1 Inserting Points into the Empty Scene

We iteratively insert 150 random point obstacles into the *Empty* scene. Each time a point is inserted, we update the navigation mesh by either using our ADDPOINT method to locally repair the mesh, or by rebuilding the navigation mesh from scratch using the graphics processor [11]. Fig. 8 illustrates the results of locally repairing the navigation mesh each time a point obstacle is inserted. All insertion times have been averaged over ten separate runs of the experiment. Note that each experiment used different randomly generated points. The horizontal axis of this diagram denotes the i-th point that is being inserted. Notice that the average time to locally insert a single point obstacle using ADDPOINT varies between $0.2ms$ and $0.6ms$.

Fig. 8 also shows the average time to reconstruct the entire scene using the graphics card each time a point is added. For a resolution of 1000x1000 pixels, complete reconstruction times vary between $9ms$ and $22ms$. This means that locally repairing the navigation mesh using the CPU is much cheaper than completely reconstructing the mesh using the graphics card.

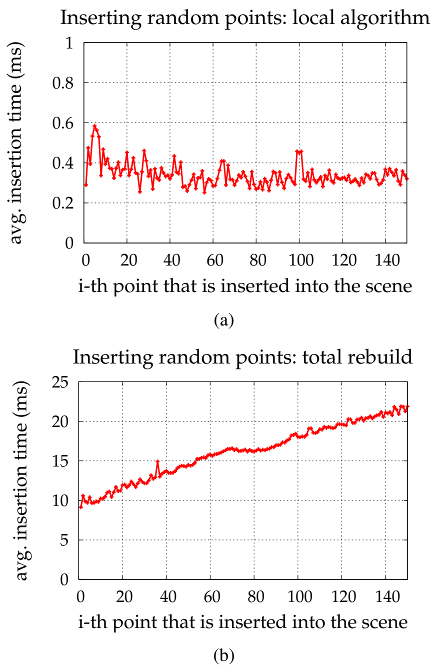


Fig. 8. We incrementally insert 150 random points into the *Empty* scene. The insertion times have been averaged over ten separate runs of the experiment. (a) If we locally update the navigation mesh after each insertion, then each insertion takes between $0.2ms$ and $0.6ms$. (b) By contrast, if the graphics card is used to rebuild the entire scene from scratch each time a point is inserted, then each insertion takes between $9ms$ and $22ms$.

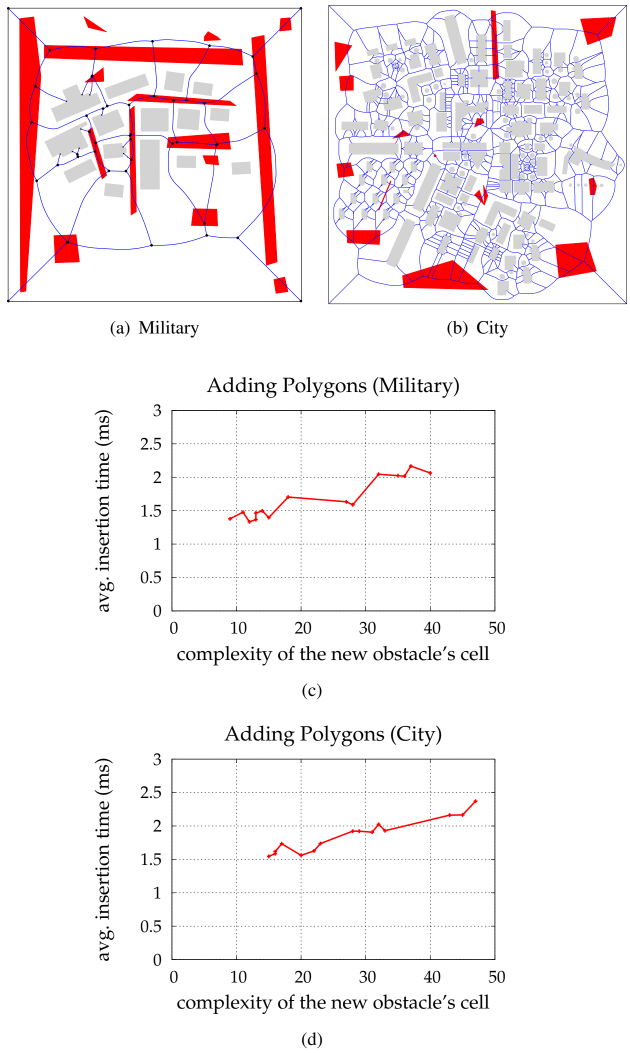


Fig. 9. A set of polygons is inserted into the *Military* scene (a) or the *City* scene (b). Obstacles in the original scene are shown in light gray, and their medial axis is shown in blue. Each red inserted polygon was inserted 10 times, and the average insertion time is displayed in the vertical axis of the graphs for *Military* (c) and *City* (d). To insert an obstacle, we build the new cell for this obstacle by computing a sequence of bisectors. The number of vertices defining this new cell is shown on the horizontal axis.

6.2 Inserting Polygons into the Military and City Scenes

We will now use our local algorithm to iteratively insert *convex polygons* into the *Military* and *City* scenes. As illustrated in Fig. 9, we choose 15 different polygons to insert into the *Military* scene. Each time an obstacle is inserted, it defines a new cell in the navigation mesh. In our experiments, the complexity of each new obstacle's cell varies between 9 and 40 bisector vertices, and the time to perform each insertion varies between $1.3ms$ and $2.2ms$. In the *City* scene, we also choose 15 different polygons to insert. The complexity of each new obstacle's cell varies between 15 and 47 bisector vertices, and the time to perform each insertion varies between $1.5ms$ and $2.4ms$. This means that polygons can be inserted quickly enough to be useful in real-time applications.

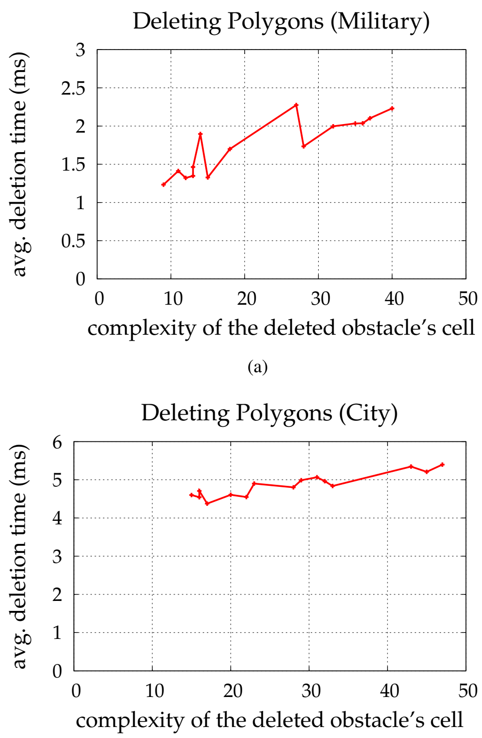
6.3 Deleting Polygons from a Scene

We will now use our local algorithm to iteratively delete each of the red polygons shown in Fig. 9. As shown in Fig. 10, the complexity of each obstacle's cell in the *Military* scene varies between 9 and 40 bisector vertices, and the time to perform each deletion varies between $1.2ms$ and $2.3ms$. In the *City* scene, the complexity of each obstacle's cell varies between 15 and 47 bisector vertices, and the time to perform each deletion varies between $4.3ms$ and $5.4ms$. Hence, deletions take significantly longer than insertions. This follows because an insertion simply needs to build the new cell for the inserted obstacle. By comparison, a deletion must construct the medial axis of all neighbouring obstacles of the obstacle that is being deleted.

6.4 Moving a Convex Polygon in a Scene

We move a polygonal obstacle with 6 vertices through the environments. Since deletions are more expensive than insertions, we move an obstacle by first storing the navigation mesh without the moving obstacle. Each frame, we can then insert the obstacle into a static scene.

In each scene, we moved the polygon until 1000 distinct insertions of the polygon had been performed. The average insertion times were $0.29ms$ in *Empty*, $0.78ms$ in *Military*, $0.65ms$ in *Zelda*, $1.09ms$ in *City*, and $0.55ms$ in *Layers*. The speed of these operations means that the navigation mesh can be maintained in real-time as multiple obstacles are moved.



(a)

(b)

Fig. 10. The red polygons from Fig. 9 are iteratively deleted from the *Military* scene (a) or the *City* scene (b). Each polygon was deleted 10 times, and the average deletion time is displayed in the graph.

7 Conclusion

Gaming and simulation applications typically contain events that lead to small changes in the environment. We have described algorithms that locally update a navigation mesh each time a point, line segment, or polygon is added to or removed from either a 2D environment or a 2.5D multi-layered environment. Our experiments show that local routines are much faster than completely reconstructing the navigation mesh. The quickness of the local routines makes a dynamic navigation mesh suitable for real-time settings. A movie highlighting the effectiveness and robustness of these techniques can be found online[1]. In the future, we will augment the navigation mesh with terrain slopes and types.

Acknowledgements

This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO), and by the COMMIT project (`http://www.commit-nl.nl/`).

References

[1] J.P. van den Berg, M.C. Lin, and D. Manocha. Reciprocal Velocity Obstacles for real-time multi-agent navigation. *IEEE International Conference on Robotics and Automation*, pages 1928–1935, 2008.

[2] M. de Berg, O. Cheong, M. van Kreveld, and M.H. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[3] O. Devillers. On deletion in Delaunay triangulations. *15th Symposium on Computational Geometry*, pages 181–188, 1999.

[4] P.N. Dixit D.H. Hale, G.M. Youngblood. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. *4th AAAI AI Interactive Digital Entertainment Conference*, pages 173–178, 2008.

[5] S. Fortune. A sweepline algorithm for Voronoi diagrams. *2nd Symposium on Computational Geometry*, pages 313–322, 1986.

[6] R. Gayle, A. Sud, M.C. Lin, and D. Manocha. Reactive deformation roadmaps: motion planning of multiple robots in dynamic environments. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3777–3783, 2007.

[7] R. Geraerts. Planning short paths with clearance using Explicit Corridors. *IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.

[8] I. Gowda, D. Kirkpatrick, D. Lee, and A. Naamad. Dynamic Voronoi diagrams. *IEEE Transactions on Information Theory*, 29(5):724–731, 1983.

[9] P.J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978.

[10] M. Held and S. Huber. Topology-oriented incremental computation of Voronoi diagrams of circular arcs and straight-line segments. *Computer-Aided Design*, 41(5):327–338, 2009.

[11] K.E. Hoff III, T. Culver, J. Keyser, M.C. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. *International Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[12] N.S. Jaklin, A.F. Cook IV, and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds*, 24(3):285–295, 2013.

[13] M. Kallmann. Shortest paths with arbitrary clearance from navigation meshes. *Eurographics / SIGGRAPH Symposium on Computer Animation*, 2010.

[14] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained Delaunay triangulations. *Geometric Modeling for Scientific Visualization*, pages 241–257, 2003.

[15] M. Kallmann and M. Matarić. Motion planning using dynamic roadmaps. *IEEE International Conference on Robotics and Automation*, pages 4399–4404, 2004.

[16] I. Karamouzas, R. Geraerts, and M.H. Overmars. Indicative routes for path planning and crowd simulation. *4th International Conference on Foundations of Digital Games*, pages 113–120, 2009.

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

[18] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. *IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[19] M. Mononen. Recast navigation. *Google Project: http://code.google.com/p/recastnavigation*, 2011.

[20] M.A. Mostafavi, C. Gold, and M. Dakowicz. Delete and insert operations in Voronoi/Delaunay methods and applications. *Computers and Geosciences*, 29(4):523–530, 2003.

[21] F. de Moura Pinto and C.M. Dal Sasso Freitas. Dynamic Voronoi diagram of complex sites. *The Visual Computer: International Journal of Computer Graphics*, 27(6–8):463–472, 2011.

[22] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. John Wiley and Sons, Ltd., 2nd edition, 2000.

[23] R. Olivia and N. Pelechano. Automatic generation of suboptimal navmeshes. *Motion in Games*, pages 328–339, 2011.

[24] J. Pettré, J.P. Laumond, and D. Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. *1st International Workshop on Crowd Simulation*, pages 1–9, 2005.

[25] F. Preparata. The medial axis of a simple polygon. In *Mathematical Foundations of Computer Science*, volume 53, pages 443–450. Springer, 1977.

[26] S. Rabin. AI game programming wisdom 2. *Charles River Media Inc., Hingham*, 2004.

[27] T. Roos and H. Noltemeier. Dynamic Voronoi diagrams in motion planning. *System Modelling and Optimization*, 180:102–111, 1992.

[28] M.I. Shamos and D. Hoey. Closest-point problems. *IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[29] G. Snook. Simplified 3D movement and pathfinding using navigation meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[30] A. Sud, R. Gayle, E. Andersen, S. Guy, M.C. Lin, and D. Manocha. Real-time navigation of independent agents using adaptive roadmaps. *ACM Symposium on Virtual Reality Software and Technology*, pages 99–106, 2007.

[31] W.G. van Toll, A.F. Cook IV, and R. Geraerts. Navigation meshes for realistic multi-layered environments. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3526–3532, 2011.

[32] W.G. van Toll, A.F. Cook IV, and R. Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012.

[33] W.G. van Toll, A.F. Cook IV, and R. Geraerts. Real-time density-based crowd simulation. *Computer Animation and Virtual Worlds*, 23(1):59–69, 2012.

[34] R. Wein, J.P. van den Berg, and D. Halperin. The Visibility-Voronoi Complex and its applications. *Computational Geometry: Theory and Applications*, 36(1):66–78, 2007.

[1] `http://www.youtube.com/watch?v=tH9dNESH4ic`. The video also refers to a webpage with more information.