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].

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. Petté 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 tech-
niques 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
called the Explicit Corridor Map to partition a two-dimensional en-
vironment into a set of walkable areas in $O(n \log n)$ time. This work
was extended to deal with multi-layered environments [31]. An ad-

c-\vantage of this structure is that it naturally encodes clearance infor-
mation 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 ob-

stacle 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 contin-
uous updates in an environment with moving points. Kallmann and
Matarí [15] describe dynamic roadmaps that keep track of the ob-
stacles 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 con-
side 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 experi-
iments 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 informa-
 tion. 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 me-
dial 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 aug-
mented 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 $\{p_1, \ldots, p_m\}$ 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 $\{C_1, \ldots, C_m\}$. 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 $p_j$, $j$. The boundary of a cell $C_i$
is denoted by $\partial C_i$. As shown in Fig. 2a, a cell in the Voronoi
diagram can have disconnected components.
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 \( C_i \) 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 \( i_3 \) and \( i_4 \) (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 \( \text{GETBISECTORARC} \) that calculates the next bisector arc that 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 \( \text{CLOSESTOBSTACLE} \) is available. Given any point in the environment, \( \text{CLOSESTOBSTACLE} \) 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 \( \text{CLOSESTOBSTACLE} \) 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].
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 bisection $B_{p,ob}$ 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 $\partial C_{ob}$ with this bisection.

\[(ob, obSeg, obPt) \rightarrow \text{CLOSESTOBSTACLE}(p)\]

\[(b, \text{GETBISSECTORARC}(p, ob, obSeg, obPt))\]

\[(t_1, t_2) \leftarrow \text{the two intersection points in } b \cap \partial C_{ob}\]

\[m \leftarrow \text{the midpoint of the bisection arc from } t_1 \text{ to } t_2\]

\[\text{arcsR} \leftarrow \text{TRACEBISECTOR}(p, ob, obSeg, obPt, b, m, t_1)\]

\[\text{arcsL} \leftarrow \text{TRACEBISECTOR}(p, ob, obSeg, obPt, b, m, t_2)\]

Reverse the sequence of bisection arcs in $\text{arcsR}$ and $\text{arcsL}$.

\[e \leftarrow \text{the sequence of bisection arcs in } \text{arcsR} \text{ and } \text{arcsL}\]

\[(t_1, t_2) \leftarrow \text{the endpoints of } e\]

\[\text{arcsR} \leftarrow \text{reverse the sequence of bisector arcs in } \text{arcsR}\]

\[m \leftarrow \text{the midpoint of the bisector arc from } t_1 \text{ to } t_2\]

\[\text{arcsL} \leftarrow \text{TRACEBISECTOR}(p, ob, obSeg, obPt, b, m, t_2)\]

\[\text{arcsL} \leftarrow \text{TRACEBISECTOR}(p, ob, obSeg, obPt, b, m, t_2)\]

The position of $\text{obPt}$ is now complete.

\[\text{Update closest point information for all modified cells.}\]

Algorithm 2 GETBISSECTORARC(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.

\[\text{if ob is a point then return the line segment bisection of } p \text{ and } obPt}\]

\[\text{else if obPt is an endpoint of } obSeg \text{ then return the line segment bisection of } p \text{ and obPt}\]

\[\text{else return the parabolic bisection of } p \text{ and } obSeg\]

intersects the boundary of $C_{ob}$. Note that the point $\text{icell}$ is a bisection vertex because $\text{icell}$ has at least three closest obstacles (including the new obstacle p).

As shown in Fig. 4b, a bisection vertex can also appear at a point $\text{inormal}$ because a new vertex or line segment of the polygon ob starts inducing the bisection $B_{p,ob}$ at some point in the interior of $C_{ob}$. The position of $\text{inormal}$ 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 bisection vertex, TRACEBISECTOR repeatedly determines the positions of both $\text{icell}$ and $\text{inormal}$. The intersection that occurs first along the directed bisector arc becomes the endpoint of the current arc, until an instance of $\text{icell}$ is chosen and the bisection 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 bisection vertices. This means that the TRACEBISECTOR subroutine should consider three candidate bisection endpoints at each step. As before, a bisection vertex can occur when a bisection arc intersects the boundary of a cell or when a bisection arc intersects a surface normal through an endpoint of the current closest obstacle.

The new scenario is that a bisection vertex can now also occur when a bisection arc intersects a surface normal through a vertex of the line segment that is being inserted. Please refer to Fig. 5a.

The GETBISSECTORARC subroutine should consider the ‘active part’ of the line segment that is being inserted. For example, when drawing the portion of the bisection in between the surface normals through the new segment’s endpoints, we need to return the bisection of a line segment and the closest obstacle. By contrast, when drawing the remainder of the bisection, we need to return the bisection 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 bisection changes whenever a bisection arc crosses one of the surface normals that passes through a vertex of the polygon. See Fig. 5b.

Algorithm 3 TRACEBISECTOR(p, ob, obSeg, obPt, b, icell, tcell)

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 $\partial C_{ob}$.
Output: A sequence of bisection arcs from $i_{start}$ to $\partial C_{ob}$.

result $\leftarrow$ empty list that will store bisection arcs
finished $\leftarrow$ false

while finished $\leftarrow$ 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

\[\text{inormal} \leftarrow \text{the first intersection of } b \text{ with the normals through the endpoints of } obSeg\]

else

\[\text{inormal} \leftarrow \text{NIL}\]

if $i_{cell}$ precedes $\text{inormal}$ along b then

\{The bisection intersects the boundary of the cell\}

\[\text{result add}(\text{icell})\]

\[\text{finished} \leftarrow \text{true}\]

else \{The current arc has an endpoint inside the cell\}

\[\text{result add}(\text{inormal})\]

\[(obSeg, obPt) \rightarrow \text{CLOSESTOBSTACLE}(\text{inormal}, ob)\]

\[i_{start} \leftarrow \text{inormal}\]

\[b \leftarrow \text{GETBISSECTORARC}(p, ob, obSeg, obPt)\]

\[i_{cell} \leftarrow \text{the first intersection of } b \text{ with } \partial C_{ob}\]

\[\text{return result}\]

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.

As depicted in Fig. 6, let \( N_p \) 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 \( N_p \) 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 \( N_p \) 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 (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.

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>
<thead>
<tr>
<th>Environment</th>
<th>Name</th>
<th>Size(m)</th>
<th>vertices</th>
<th>Navigation Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Size(m)</td>
</tr>
<tr>
<td>Empty</td>
<td>100x100</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Military</td>
<td>200x200</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zelda</td>
<td>100x100</td>
<td>560</td>
<td></td>
<td></td>
</tr>
<tr>
<td>City</td>
<td>500x500</td>
<td>2638</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layers</td>
<td>100x100</td>
<td>213</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 0.2\,ms and 0.6\,ms.

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 9\,ms and 22\,ms. 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.](image)

### 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 1.3\,ms and 2.2\,ms. 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 1.5\,ms and 2.4\,ms. This means that polygons can be inserted quickly enough to be useful in real-time applications.

![Fig. 9.](image)

### 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 1.2\,ms and 2.3\,ms. 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 4.3\,ms and 5.4\,ms. 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 environment. 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 0.29\,ms in Empty, 0.78\,ms in Military, 0.65\,ms in Zelda, 1.09\,ms in City, and 0.55\,ms in Layers. The speed of these operations means that the navigation mesh can be maintained in real-time as multiple obstacles are moved.
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 online1. 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.ni/).

References


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