Localized Recalculation of the Explicit Corridor Map

Eva Linssen
Utrecht University
3902749

December 22, 2016

Abstract

The Explicit Corridor Map (ECM) is a navigation graph inspired by the Vonoroi cells. In this navigational graph the edges are equidistant to the obstacles closest to them. Inserting or deleting an obstacle in the ECM can be done through a total rebuild, which is inefficient. A dynamic method exists, but it does not support concave or overlapping obstacles. In this work we find a method that is both fast and robust; we build a temporary ECM for only part of the obstacles and use this to repair only the relevant edges in the original ECM. We find that this method is faster than a total rebuild and that it has the advantage of supporting overlapping and concave obstacles. In an experiment relating door-like obstacles, we find our method gives a speed-up of 8 times for our biggest environment.

1 Introduction

In games and other types of virtual worlds, having a navigation mesh is quite common. It is used to know where characters can and cannot walk. In case of many characters (crowd simulations) many paths must be calculated. It is useful then, to have a data structure that describes the walkable space so that path planning can be done quickly. The Explicit Corridor Map (ECM) [1] is one such data structure. Inspired by the Vonoroi cells around the obstacles, the ECM is a navigation mesh with an underlying graph in which all edges are equidistant to the obstacles closest to them. Using the ECM a smooth path can quickly be found, also taking the radius of the character into account. The ECM describes the walkable areas of an environment on a 2D plane. By using multiple (interconnected) 2D layers it can also be extended to work for 3D environments.

The complexity of calculating the ECM is proportional to the number of vertices in the graph. This number depends on the obstacles in the environment. Increasing the number of obstacles thus also increases the complexity of the algorithm. Van Toll [2] created a dynamic algorithm to update the graph only

locally in case of insertion or deletion of one obstacle. This algorithm extracts knowledge about relevant obstacles from the edges locally, using the closest point annotation. This makes its running time dependent on the *local* complexity of the graph, in other words the number of edges and (the degree of) vertices that are involved. In comparison to a total rebuild this algorithm is very fast. A disadvantage, however, is that it supports neither concave nor overlapping obstacles.

In this work we propose a different algorithm to handle dynamic updates to an ECM. Instead of using local knowledge, like Van Toll does, we put a bounding box around the area that changes. From the obstacles within this bounding box we calculate a temporary ECM; this temporary ECM is used to repair the edges in the original ECM. Because we rebuild the ECM for only a portion of the obstacles, we expect the total running time to decrease. Another advantage of our method is that it does not assume anything about the shape or location of the obstacles in the environment. This means that it automatically supports concave and overlapping obstacles.

In Section 2 we outline some properties of the ECM. In Section 3 we will give an outline of our new method of dynamically updating the ECM. In Sections 4 and 5 we will delve deeper into the theory and explain why the method is correct. In Section 6 we will compare the efficiency of our method with both a total recalculation and van Toll's dynamic method.

2 Definitions

The Explicit Corridor Map (ECM) is a navigation mesh with an underlying graph in which edges and vertices represent the medial axis between obstacles. It is used for planning paths of characters in a simulation.

We have a 2D environment, of which the space is bounded by an axis aligned bounding box b. In this environment there is a set of obstacles O. The obstacles can be points, lines or simple polygons. This means we have a 2D space S, containing all points that lie in b. All points that lie on the interior of any obstacle $o_i \in O$ are part of the obstructed space, which is denoted S_{obs}. All free space is then $S_{free} = S - S_{obs}$. Points on the borders of the obstacles are thus counted as part of S_{free}.

Let us define an ECM as a bidirectional graph $G = (E, V)$. Each point on $e \in E$ is equidistant from the two closest obstacles/obstacle parts. Vertices lie on the points where edges meet, or in the non-concave corners of obstacles. All points closest to one ECM edge form an area that we call an *ECM cell*.

3 Dynamically Updating the ECM

The relation between an environment and its ECM comes from the fact that free space S_{free} is exactly equal to the space in the union of all ECM cells. There are two types of changes: an *insertion* and a *deletion* of an obstacle. The action

of moving of an obstacle can be simplified to deleting and then re-inserting it. The difference between an insertion and a deletion in our method is about two lines of code (Algorithm 1, lines 7-8). We will use the term *update obstacle* to mean the obstacle that is either being inserted or deleted in this update.

When an obstacle o_{ins} gets inserted into the environment, the set of obstacles changes to $O' = O \cup \{o_{ins}\}$. This means that S_{obs} changes to $S'_{obs} = S_{obs} \cup interior(o_{ins})$; thus $S'_{free} = S - S'_{obs}$. Since $S'_{free} \neq S_{free}$, the current ECM graph G does not relate to its environment any more. G needs to be updated to $G' = (E', V')$ that corresponds to S'_{free}.

G gets outdated in a similar manner when an obstacle o_{del} gets deleted from O and $O' = O - o_{del}$.

Algorithm 1 UpdateECMGraph

Input: graph G, AABB envBounds, obstacle ObsUpdate, set of all obstacles ObsAll (excluding ObsUpdate), updateType TypeUpdate
Output: the updated graph G'

Find the parts of ObsUpdate that are in free space
1: ObsOverlap ← FindAllThatOverlapObsUpdate(ObsAll, ObsUpdate)
2: ObsPieces ← Cut(ObsUpdate, ObsOverlap)
Find all influenced vertices (free and end), and the influenced edges
3: VInfl, VEnd, EInfl ← FindInfluencedGraphComponents(G,ObsUpdate, ObsPieces)
Find all obstacles needed to calculate the temporary graph
4: V ← VInlf ∪ VEnd
5: ObsImpact ← FindImpactingObstacles(ObsAll, V)
6: **if** TypeUpdate is 'Insertion' **then**
7: ObsAll.insert(ObsUpdate)
8: **end if**
Calculate the temporary graph
9: Gtemp ← CalculateECMGraph(envBounds, ObsImpact)
Find all influenced components in the temporary graph
10: VInfltemp, VEndtemp, EInfltemp ← FindInfluencedGraphComponents(Gtemp,ObsUpdate, ObsPieces)
Update the graph
11: G' ← RemoveOld(G, VInfl, EInfl)
12: G' ← CopyNewInto(G', VInfltemp, EInfltemp)
13: G' ← ConnectEndVertices(G',VEnd,VEndtemp)
14: **return** G'

In Algorithm 1 our method of dynamically updating the ECM is shown. It works for both deletion and insertion, since the only difference is the existence of the update obstacle o_{update} in the temporary graph. In the algorithm we start by cutting of all parts of o_{update} that were already in obstructed space (lines 1 and 2). We do not need those parts as they have not changed between G and G'; they stay in S_{obs}.

Next we find all edges and vertices that are influenced by the existence of o_{update} (line 3). The function FindInfluencedGraphComponents is defined in Algorithm 2 in Section 4. This function finds all graph components C whose closest point annotation becomes incorrect due to the update with o_{update}. It selects thus precisely all graph components that need to change. At a later time those graph components need to be removed from original graph G. At this point, though, we can use them to find all obstacles that are relevant to the build of the temporary graph G_{temp}.

From the influenced vertices we create a bounding box that contains at least all obstacles that influence the recalculated area (line 5). The function FindImpactingObstacles is defined in Algorithm 3 in Section 5. Depending on the update being an insertion or deletion, we add o_{update} to this set (lines 7-8). In the Function CalculateECMGraph we use Geraert's algorithm[1] to calculate the temporary graph G_{temp} (line 9). We find all influenced graph components C_{temp}in G_{temp} (line 10). Per definition these are exactly all graph components that need to be copied over to G. Finally, we remove the invalidated graph components C from G and connect C_{temp} into the hole we left, creating G' in the process (lines 11-13). Note that the query structure of the ECM needs to be updated after any update.

We only actually change G into G' in the last step of the algorithm. This creates opportunities to parallelize the algorithm; we can put the calculation of C, $Gtemp$ and $Ctemp$ in a background thread. During this calculation, the simulation that is using the ECM can keep access to G. Changing G into G' takes very little time in comparison to the total execution time of our algorithm. This keeps the lag in the simulation caused by the update to a minimum.

In Figures 1a) to 1f) the method is displayed step by step for the insertion of an obstacle.

4 Finding Influenced Components

In our method we need the influenced graph components for both the original and the temporary graph. We claim that this selects precisely all edges that make up the difference between the graphs with and without the update obstacle o_{update}. In this section we will describe why and how this works. Pseudo-code for finding the influenced graph components is shown in Algorithm 2.

The difference between the two graphs before and after the update is only one obstacle o_{update}. If this update is an insertion, the differing obstacle is in the after-graph, but not in the before-graph. If this update is a deletion, the obstacle is in the before-graph, but not the after-graph. Fact is that every edge or vertex that exists in both graphs is not affected by o_{update}. Thus, we only need to focus on all edges and vertices that *are* affected by the (non)existence of the update obstacle.

The question is now how we can identify the edges and vertices that are affected by our update obstacle o_{update} in an efficient manner. We can use the definition of the ECM cell for this. The ECM cell of an edge e contains all points

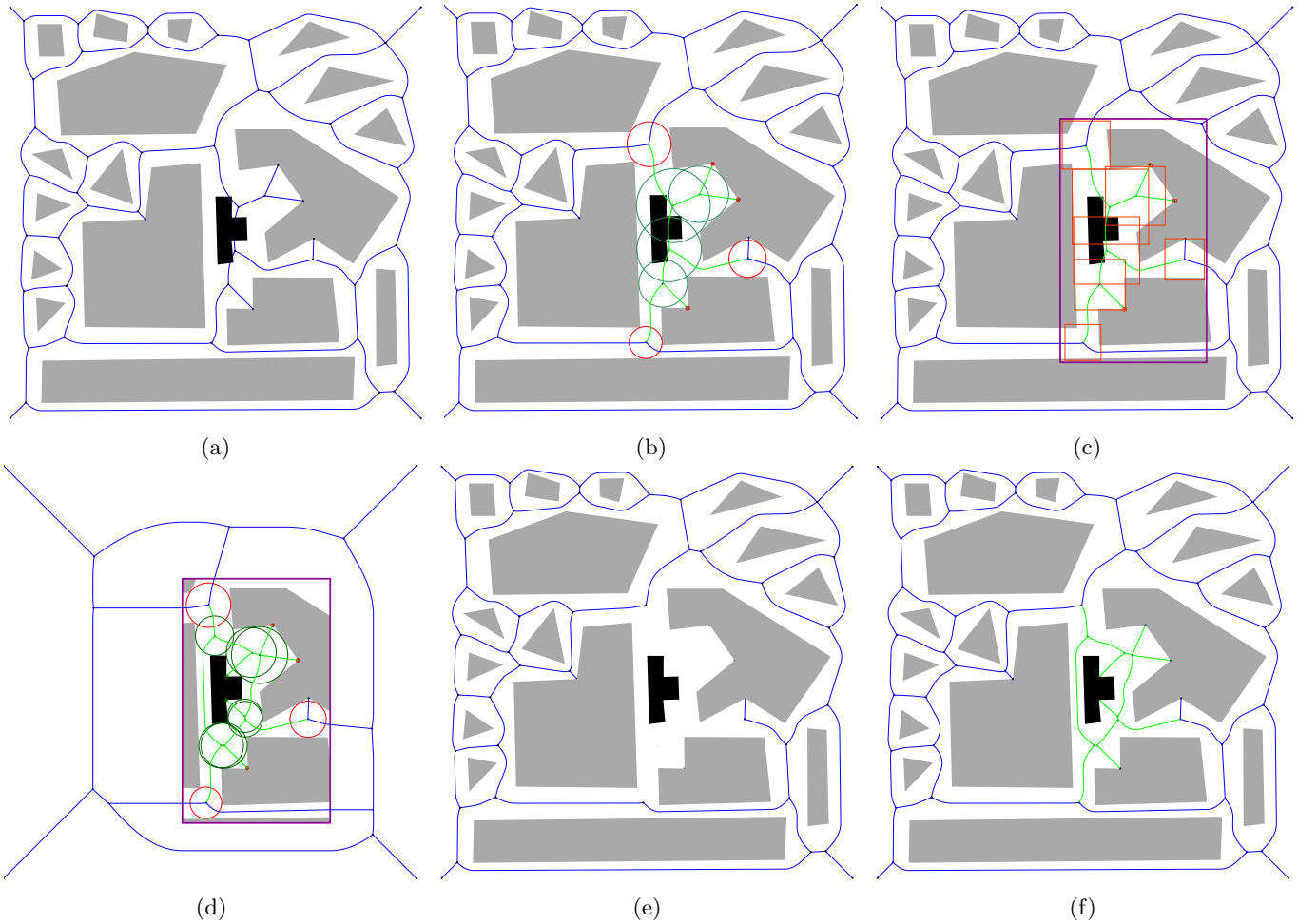


Figure 1: Our method visualised in six steps for an insertion update. a) The black obstacle is inserted into the environment. b) In a graph search all influenced edges (green) and vertices (in green circle) and end-vertices (in red circle) are found, based on the minimum distance to the black obstacle. c) We unite the bounding boxes around the influenced vertices and end-vertices (both in red squares) found in the previous step. The resulting bounding box is shown as a purple rectangle. d) The temporary graph is build from the obstacle parts within the bounding box and the influenced components are found. e) The influenced components in the original graph are removed. f) The influenced components from the temporary graph are placed into the cleaned original graph.

Algorithm 2 FindInfluencedGraphComponents

Input: graph G, update obstacle ObsUpdate, and set of obstacle ObsPieces (that describes ObsUpdate after all parts that lie in obstructed space have been cut off)

Output: set of vertices InfluencedV, set of vertices EndV, set of edges InfluencedE

```
1: for all obstacle ObsPart in ObsPieces do
2:   Find starting edge
3:   Edge startE ← G.getAClosestEdge(ObsPart)
     Initialize queue of vertices Q
4:   Q.push(startE.sourceVertex)
5:   Q.push(startE.targetVertex)
6:   while (not Q.empty()) do
7:     Vertex V ← Q.pop()
       Does the obstacle lie in the clearance disc around V?
8:     if (MinDist(V.pos, ObsUpdate) ≤ V.radius) then
9:       The vertex is influenced
10:      InfluencedV.insert(V)
         mark its edges as influenced
11:      for all (Edge e in V.edges) do
12:        if (e not in InfluencedE) then
13:          InfluencedE.insert(e)
             Add unseen vertices to queue
14:          Q.push(e.targetVertex)
15:        end if
16:      end for
17:    else
18:      The vertex is not influenced, so mark it as end-vertex
19:      EndV.insert(V)
20:    end if
21:  end while
22: end for
```

$c \in S_{free}$ that are closest to only this edge. Because of the query structure that is part of the ECM, we can quickly find the edge closest to any point $p \in S_{free}$.

4.1 Updating with a point obstacle

Suppose we insert a point obstacle o_{ins} into an ECM on position p_o. We have the before-graph G that does not contain o_{ins} and an after-graph G' that does contain o_{ins}. We want to find all influenced edges, all influenced vertices and the vertices that exist in both graphs and which also neighbour at least one influenced vertex. These are called *end-vertices*. The end-vertices are used to connect the influenced components of the temporary graph into the hole we created in the original graph. This can also be seen in Figures 1b) and 1d); the vertices in the red circles lie on the same position in both graphs.

The end-vertices are the first vertices the algorithm runs into that are not influenced by the (non)existence of o_{update}. Everything but the influenced components in both graphs is exactly the same. This means the set of end-vertices found in the original graph and the set found in the temporary graph should also be equal.

4.1.1 Graph that does not contain the update obstacle

Let us start with the influenced components in the before-graph. We find an edge e_{start} closest to p_o. We call this edge the *starting edge*, because from here we will do a graph search to find all influenced components.

We can find the point q_e on e_{start} that lies closest to p_o and the distance $d = Dist(q_e, p_o)$. Because q_e is the point closest to p_o in edge e_{start}, d is the *minimum distance* between o_{update} and e_{start}. Because of the closest point annotation in the edges, we can also find the distance r between q_e and its currently closest obstacle.

Now, if $d < r$ this means the inserted obstacle is closer than the currently closest obstacle; this invalidates the edge in question.

Instead of doing the same for every edge in the graph, we look at the vertices during a graph search. Suppose a vertex v lies on a crossing of edges $E_v \in E$. Because of this, the set of points closest to v on all edges $e \in E_v$ all have the same distance d_v to v. Imagine a disc around v with radius $r_v = d_v$. If point obstacle o_{ins} lies inside the disc, its minimum distance d_o to v has $d_o < r_v$. This means that v gets invalidated. If the vertex gets invalidated this means that all edges that cross on this vertex get invalidated as well. Any graph component that gets invalidated by o_{ins} is an influenced component.

Starting from the two vertices at the ends of e_{start}, we do a graph search along the graph. If a vertex gets invalidated, we mark it and its edges as "influenced". We set each neighbour vertex to be checked as well. If the graph search runs into a vertex that is not influenced, we mark this an *end-vertex*.

Doing this graph search we get all influenced components and end-vertices in the before-graph that o_{ins} is not yet part of.

4.1.2 Graph that does contain the update obstacle

Let us now take a look at the after-graph in the case of a point obstacle insertion.

Since the obstacle o_{ins} already exists, technically this obstacle itself cannot invalidate any graph component any more. This is because the closest point annotation on the edges already take note of o_{ins}. We know o_{ins} can never lie any closer to an edge/vertex than the current closest obstacle $o_{closest}$: either o_{ins} lies further away ($d_o > r$), or $o_{ins} = o_{closest}$ ($d_o = r$). In any case, $d_o < r$ never happens. We need to change our definition of "influenced" a little, so we can also find those components in the graph in which o_{ins} already exists. We change the definition of influenced to $d_o \leq r$ so that we can also select edges and vertices that have $o_{closest} = o_{ins}$ (which are already are under influence of o_{ins}).

Note that by switching the order of the before-graph and the after-graph in this insertion example, we create a deletion. Proving that a point obstacle deletion finds the correct influenced components in our algorithm is thus done in a similar manner.

4.2 Extension to support lines and simple polygons

We base our algorithm on the *minimum distance* between the update obstacle and the vertex. Extending this idea to work for obstacles more complex than a point is easy. A minimum distance between a line or a simple polygon and a vertex location (which is a point) can also be calculated.

There are a few other things to keep in mind though. Since lines and simple polygons contain multiple points: a) we have more points to choose from to find a starting edge e_{start} and b) the obstacle can lie partially in S_{free} and partially in S_{obs}. Also, in case the obstacle is concave, we must correctly mark the concave corners as influenced or not.

4.2.1 Selecting the starting point

We still need to find a way to find a good starting point for the obstacle (piece). The query structure of the ECM can not find closest edges for any point $p \in S_{obs}$. We want a starting point that can be used for both the temporary and the original graph. This excludes any point in the interior of o_{update} as in at least one of those graphs it is part of the obstructed space S_{obs}. This leaves us with any point on the exterior of o_{update}.

A set of points P can be called a continuous components, if for any $p, q \in P$ there exists a continuous path pq between p and q. For all continuous components $P_{free} \subseteq S_{free}$ there thus exist path pq between any $p, q \in P_{free}$. Each connected component of the ECM graph depicts one continuous component of S_{free}. Because of the relation between the ECM graph and S_{free}, there must also exist a path for p and q through a connected component in the ECM graph.

For an obstacle that does not overlap any obstructed space, there is a continuous path between any two points on the obstacle. This means that there

should also always be a path between these points within the graph. This also holds for any two points on the border of the obstacle.

We base our graph search on the minimum distance between the obstacle and the vertices in the graph. This minimum distance is 0 for any vertex that lies within the obstacle. Any other minimum distance is calculated between a vertex and its closest point on the *border of* the obstacle. Since all border points are connected by a continuous path, all vertex that lie close enough to be called "influenced" will be reached. Consequently, starting a graph search from only one point on a non-overlapping obstacle should find all influenced components.

We can choose any obstacle-vertex as a starting point when the update obstacle does not overlap any other obstacles.

4.2.2 Partially obstructed obstacles

Ensuring we find all influenced components for a partially obstructed obstacle is more difficult.

We need to first remove all parts of the obstacle that lie in S_{obs}. These points would not do anything for the algorithm as they lie in S_{obs} in both graphs.

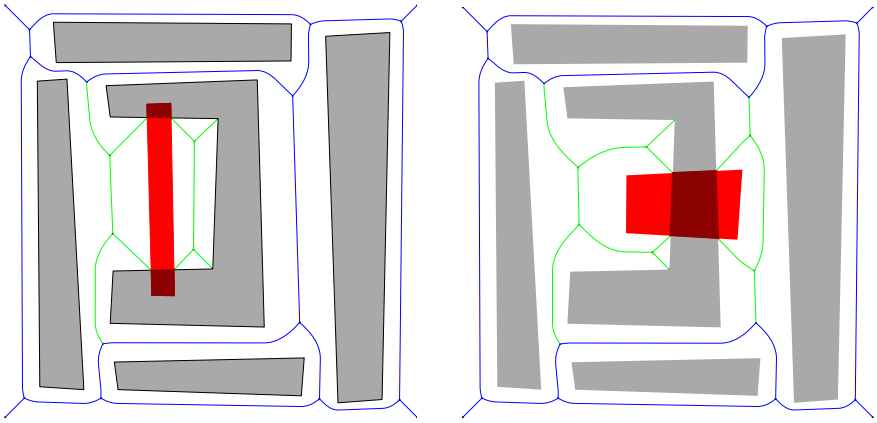
Removing all obstructed space from an obstacle can fracture it into multiple *obstacle pieces*. For neither the points in the free space beneath the obstacle, nor the points on the border of the obstacle, is the existence of a continuous path between any two points a guarantee. Consequently, we can not guarantee that all influenced components can be reached from one starting point on the border of the obstacle. Multiple graph searches are needed, which start from different parts of the obstacle.

In Figure 2 two cases are shown in which the update obstacle influences different disconnected parts of the graph. Note that in Figure 2a) the graph search has to start at least twice, because the set of points on the border of the obstacle has been split into two continuous sets.

We let the graph search start once for each obstacle-vertex on each obstacle-piece. Our obstacles are build up out of a collection of lines. Since the obstacle vertex connect the lines in the obstacle border, a vertex has a continuous path to all points on its connected edges. By trying all obstacle vertices on all pieces of the obstacle, we are ensured that all points on the obstacles border that lie in S_{free} reach an influenced edge. Thus we are guaranteed to find all influenced edges, vertices and end-vertices.

Many of the starting points will be redundant as they result in the same found graph component. We can not know however, which obstacle points belong to which graph component, we will have to start a search for all of them. Because we keep track of the vertices we already visited during the search, a vertex will not get visited multiple times.

It is important to note that the minimum distance must not be computed in relation to an obstacle piece, but to the entire obstacle. Otherwise pieces of a concave obstacle that lie closer might be skipped during the calculation of the minimum distance.



(a) (b)

Figure 2: Two examples in which the graph components influenced by the red obstacle are not interconnected. The influenced components are green. Note that the dark red part of the obstacle gets removed before recalculation; this results in the obstacle becoming multiple obstacles parts for b).

4.2.3 The marking of Vertices in concave corners

Having concave obstacles in the environment means we need to correctly mark the concave corners as well. In the ECM the concave corners cause the existence of a vertex on their position. Since these vertices lie on the boundary of the concave obstacle, they have their closest obstacle points at the distance of 0.

This means that any vertex v in a concave corner of an obstacle o only gets invalidated if a) the inserted updated obstacle $o_{update} \neq o$ and overlaps the vertex, and b) the obstacle $o = o_{update}$ and gets deleted.

In case a) the minimum distance to o_{update} is 0 because of the overlap. In case b) it is 0 because the closest points and the update obstacle are the exact same. This means that our algorithm already marks the corner-vertex correctly.

5 Filtering the Obstacles

We have shown in the previous section how we find all edges and vertices that make the difference between the graphs before and after an update. These differing edges and vertices were marked as "influenced" by the update obstacle. We have not yet considered, however, that we want the temporary graph to be build from as few obstacles as possible.

Algorithm 3 FindImpactingObstacles

Input: list of all obstacles ObsAll, set of vertex InfluencedV
Output: list of obstacles that includes at least those we need for a local recalculation

```
1: AABB ImpactBounds ← findVertexBoundingBox(V)
2: list of obstacles ImpactList
3: ImpactList ← Intersection(ImpactBounds, ObsAll)
4: return ImpactList
```

Algorithm 4 FindVertexBoundingBox

Input: set of vertices InfluencedV
Output: Axis aligned bounding box (AABB) that contains at least all obstacles/obstacle parts we need for a local recalculation

```
1: AABB resultBounds
2: list of AABB VertexBoundsList
   find a bounding box for each individual vertex
3: for (Vertex V in InfluencedV) do
4:    distFromCenter ← V.disc.radius+margin
5:    boundsOfV ← AABB(V.pos.x-distFromCenter, V.pos.y-distFromCenter,
      V.pos.x+distFromCenter, V.pos.y+distFromCenter)
6:    VertexBoundsList.add(boundsOfV)
7: end for
8: resultBounds ← TakeUnionOfAABBs(VertexBoundsList)
9: return resultBounds
```

Theoretically we could build the temporary graph from all obstacles in the after-graph. Then, however, our algorithm gets per definition slower than the total rebuild; the building of the temporary graph by itself would then be a total rebuild. All other calculations in addition to the rebuild would make this algorithm slower than a normal total rebuild.

We thus need to decrease the number of obstacles used for our temporary graph. We have only one requirement on the set of obstacles we select. In our selection we need at least all obstacles that have impact on the formation of the influenced edges in the original graph. The influenced vertices and end-vertices lie on these edges. Since we select all obstacles that form the end-vertices, we ensure that the sets end-vertices in both graphs are equal.

This part of our algorithm is shown in in pseudo-code in Algorithms 3 and 4. In the rest of this section we will explain why this is results in the correct set of obstacles.

5.1 Finding the bounding box around the influenced components

We need to create a bounding box b_{infl} that contains at least all the obstacles O_{infl} that had impact on the formation of all influenced edges. Then we apply the update (the removal or insertion of the update obstacle) to only O_{infl}; We can create a temporary graph from the obstacles in the updated O_{infl} which we can then use to repair the edges in the original graph.

There are a few observations about the ECM graph we can use to create the bounding box b_{infl}. A concave obstacle corner causes the existence of a vertex in that corner. A corner-vertex always has one outgoing edge. If two convex obstacles have neighbouring Vonoroi cells, there is precisely one edge between them in the ECM graph. Between a convex obstacle and a concave obstacle with neighbouring Vonoroi cells, there can be multiple ECM edges. This is logical: the edge outgoing from any corner-vertex needs to connect to the graph. The two obstacle boundary lines that form the concave corner and the closest point on the other obstacle cause the existence of another vertex. This vertex is connected to the single edge that comes out of the corner vertex.

An ECM cell c_e around any edge e contains all free space closest to this edge. If we increase the size of c_e with a margin m, at least all obstacles that impacted the formation of edge e overlap c_e. An ECM cell can be a complex shape, so we find the bounding box b_e around c_e instead. We take the two vertices v_1 and v_2 at the ends of e. We define $box(v)$ to mean the bounding box around the maximum clearance disc around vertex v increased by margin m.

Theorem 1 Suppose we have any edge e with at its endings the vertices v_1 and v_2. The bounding box b_e is calculated as $b_e = box(v_1) + box(v_2)$. Now, the set of obstacles $O_e = O_{obs} \cap b_e$ contains at least all obstacles/obstacle parts, so that if we build an ECM from only the obstacles O_e, then the edge e would exist in this ECM.

Proof The edge e is the medial axis between two obstacles. The parts of the two obstacles that form the edge e are both convex. Let us call these two obstacle parts obstacles I and J. Neither I nor J can be concave because any concave corner in combination with the closest position on the other obstacle would cause a vertex partitioning e. The multiple parts that e would be split up in, are then formed by multiple tinier convex obstacles parts of I and J.

To find all obstacle parts that precisely form edge e, we need the set of all closest obstacle points P_{close}, closest to the two vertices. These points lie inside of the union $b_e = box(v_1) \cup box(v_2)$. Since a bounding box is rectangle (a convex shape) we know that any lines (p, q) between points $p, q \in P_{close}$ also lie completely inside of $box(v)$. Let us consider the four points $i_{v_1}, i_{v_2}, j_{v_1}$ and j_{v_2}, respectively the points on I and J that are closest to v_1 and v_2. We can make the convex polygon H formed by the points $(v_1, i_{v_1}, i_{v_2}, v_2, j_{v_2}, j_{v_1})$. An example of how this might look is shown in Figure 3. Per definition this polygon H lies completely within the bounding box b_e. The obstacle part I is convex, so

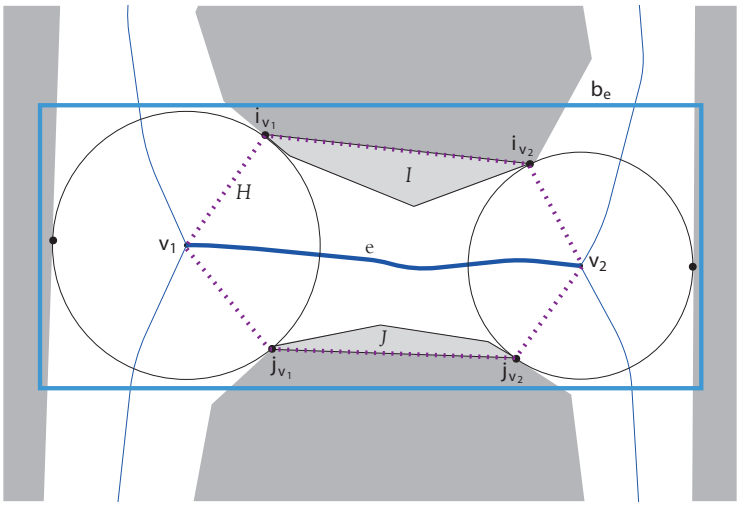


Figure 3: Edge e lies in between vertices v_1 and v_2 is formed by obstacle parts I and J. As can be seen, the ECM cell around e and the complete I and J lie in the polygon H. Because we know H and all obstacle points touching the maximum clearance disks around v_1 and v_2 lie completely inside bounding box b_e, we can say that bounding box b_e contains all obstacles needed to build e.

the continuous path from i_{v_1} to i_{v_2} is entirely contained by the polygon H. The same goes for obstacle J. The union of the two bounding boxes around the two vertices of any edge does thus include all obstacles needed to form that edge.

This also works for edges that have one vertex in a corner of a concave obstacle; in this case v_1 lies in the concave corner, the points at $v_1 = i_{v_1} = j_{v_1}$, and H still lies completely inside b_e. As $c_e \subseteq H$, and $H \subseteq b_e$, we know that all obstacles needed to rebuild e are inside of b_e.

Theorem 2 Suppose we have any subset of edges F in an ECM graph. We define the bounding box $b_F = \sum_{e \in F} b_e$. Now, the set of obstacles $O_F = O_{obs} \cap b_F$ contains at least all obstacles/obstacle parts, so that if we build an ECM from only the obstacles O_F, we recreate all $e \in F$.

Proof In Theorem 1 we have proven that b_e contains all obstacle parts to rebuild each $e \in F$. As b_F is the bounding box around all b_e, we know that any $b_e \subseteq b_F$. We have defined $O_F \subseteq b_F$. So if we rebuild an ECM from obstacle set O_F, all $e \in F$ must exist in this ECM.

Theorem 3 The bounding box $b_{infl} = \sum_{v \in V_{infl}} box(v)$ contains at least all obstacle parts that we need to build a correct temporary ECM so that its end-vertices match with those in the original ECM.

Proof In Section 4 we have found the set of edges E_{infl} that are influenced by an obstacle o_{update}. We found all edges that became invalid by the insertion/deletion of obstacle o_{update}. Edges can only be invalided by obstacles that overlap their ECM cell. We can reason that the bounding boxes around all edges $e \in E_{infl}$ contains the complete area of the ECM that needs to be updated.

The bounding box b_{infl} around all influenced edges E_{infl} is the combination of all bounding boxes around the ECM cells around all $e \in E_{infl}$. A different way to calculate b_{infl} is

$$b_{infl} = \sum_{v \in V_{infl}} box(v)$$

in which $V_{infl} = \{v \in V | v$ is an endpoint of any $e \in E_{infl}\}$. Because all end-points of edges in E_{infl} are either influenced or end-vertices, we can say $V_{infl} = v \in V | v$ is an influenced or end-vertex$\}$. As the end-vertices in both ECM have obstacles closer to them than o_{update} is, and those closest obstacles are per definition inside of b_{infl}, the sets of end-vertices in both ECM's are equal. So, the update in the temporary graph must be correct.

This proves that by taking the combination of all bounding boxes around all influenced vertices and end-vertices we include all obstacle parts we need to calculate the temporary graph correctly.

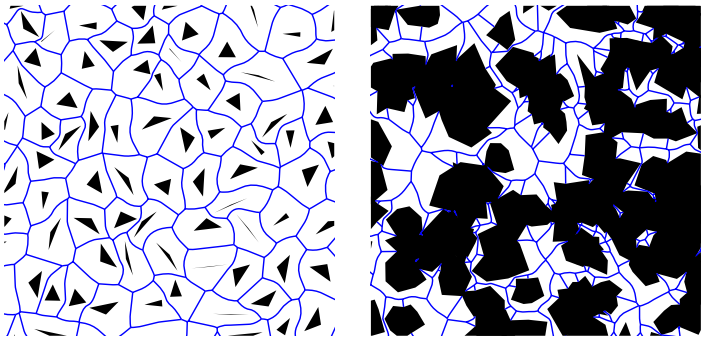
5.2 Multiple connected influenced graph components

When the influenced components C are build up from n connected components $C_x \subset C$, $x = (0, n]$, we can decrease the area of b_{infl} by defining it as the union of bounding boxes around all C_x. This is still contains all space for which the original graph needs to be updated: the bounding box around C_x contains all obstacles that are needed to build all edges in this connected component. Because per definition there are no edges interconnecting any two $C_x, C_y \subset C$, we can take one individual bounding box per connected component.

These loose bounding boxes have less volume than a union of them would, while still containing all needed obstacles. This will reduce the volume of the area we take the intersection of. This in turn reduces the calculation time of the temporary graph.

6 Results

To see if our method improves update time in comparison to a total rebuild or Van Toll's dynamic method, we conducted two experiments. The experiments were performed in Visual C++ on an NVIDIA GTX 960M graphics card and an Intel Core i7-5700HQ CPU (2.7 GHz) with 8 GB memory. We added a check after every update, to verify our method created an ECM equal to the result of the total rebuild. As long as the code did not fail, this seemed to be the case.



(a) (b)

Figure 4: Example of how an environment would look after insertion of 2000 obstacles. Only a part of the environments is shown, as 2000 obstacles makes for an unclear image. a) result of inserting triangle obstacles without allowing them to overlap. b) result of inserting concave polygon obstacles without the overlap restriction

6.1 Experiment with 2000 insertion updates

In the first experiment we looked at how the update time relates to the number of obstacles in the environment. We incrementally inserted 2000 obstacles on random positions into an environment, starting from an empty one. We did this for: point, triangle, convex and concave polygon obstacles.

The points were inserted on a random position. To create the triangle and convex polygons we found a point, created a disc around it and chose 3 and 6 to 10 points on it. Connecting those points created respectively a triangle and a convex polygon. To create a concave polygon we did the same as for the convex one, only we let the size of the disc vary.

Since we needed to compare to Van Toll's method (which does not support overlap), we split the experiment into two parts. One part for which we allowed the obstacles to overlap, and one part in which we did not. On the overlapping and concave obstacles, we did not test Van Toll's method. We enforced the non-overlap restriction by asking the ECM query structure for discs that lay completely in free space.

Two impressions of how part of an environment would look after 2000 insertions are given in Figure 4: shown are the result of inserting 2000 non-overlapping triangles and 2000 overlapping concave polygons. The result of this experiment is shown in the graphs in Figure 5.

We repeated the process for deletion updates, but as the results from those experiments were very similar to the insertion version, we do not show those results here. Since our method generalises over the types of updates (the difference is only two lines of code) this similarity was expected.

In the graphs in Figure 5 we see that our method takes less time for an update than a total rebuild would. Van Toll's method seems to have more or

Environment	number of obstacles	number of vertices	total rebuild average time (ms)	our method's average time (ms)	speed-up
Military base	28	58	6.5	3.6	1.8
Zelda	135	289	21.5	5.0	4.3
City	563	1502	133.9	20.4	6.4
Zelda8x8	8645	18292	1414.2	182.4	7.8

Table 1: Comparison between update time of our method and a total rebuild in case of insertion of door-like obstacles. For deletions of the door-like obstacles we found similar update times.

less constant time in relation to the number (or complexity) of obstacles in the environment. Our method is consistently slower than Van Toll's method. We also found that even though we proved that our theory is correct, the algorithm still fails sometimes. When the update obstacle did not overlap any other obstacles, the algorithm always worked. When there was overlap the algorithm failed in about 10% of the cases. If you consider that Van Toll's algorithm failed in about 20% of the (non-overlapping) cases, this is not a bad result.

6.2 Experiment with door-like obstacles

We have seen that our method is faster than a total rebuild for the environments that were created during 2000 insertions. As the inserted obstacles were created with randomness, those environments do not look like a realistic environment of a virtual world.

We did a second experiment testing our method in more realistic cases. As we have seen in the previous experiment, Van Toll's method is consistently faster than ours for non-overlapping obstacles. For those types of update we would recommend using Van Toll's method.

We thought of a realistic case in which our method might be preferred: the opening and closing of paths between two obstacles. By letting a polygon with a low number of sides overlap two other obstacles, we "close a door" between those two obstacles. As a door-like obstacle can be very small in comparison to the size of the entire environment, a dynamic update method would be preferred to a total rebuild. As Van Toll's method does not support overlap, however, our method is the better choice here.

Examples of door-like obstacles are shown in Figure 6. These are also three of the four environments we did this experiment for. Number four is the environment in which we repeat the Zelda environment 8x8 times within an environment. For each of our environment we added and then deleted a door-like obstacle. We did this for ten different door-like obstacles. The results of this experiment are shown in Table 1.

As expected, as the number of obstacle and graph vertices increases, both the total rebuild and our method need more time for an update. The speed-up

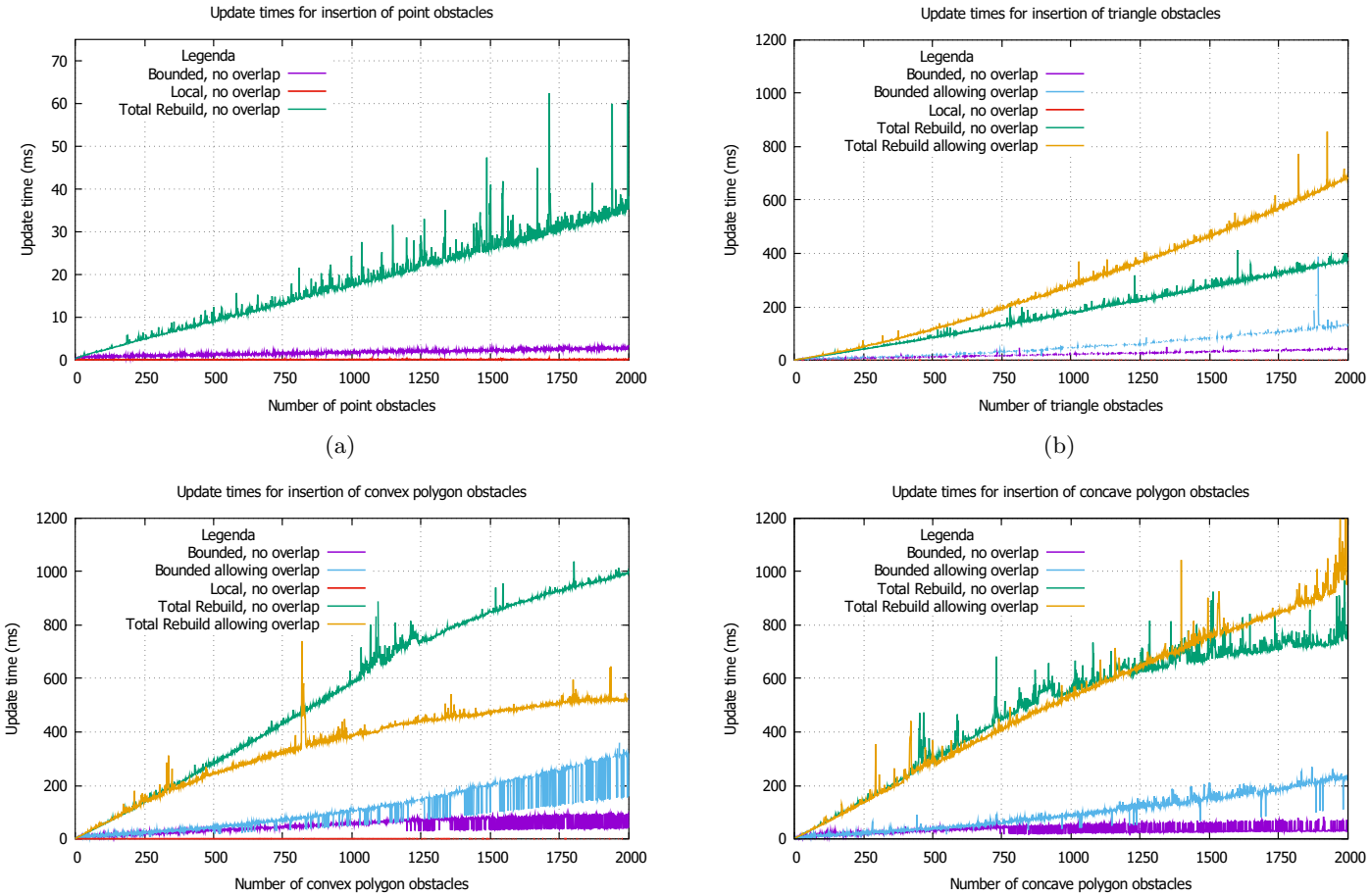
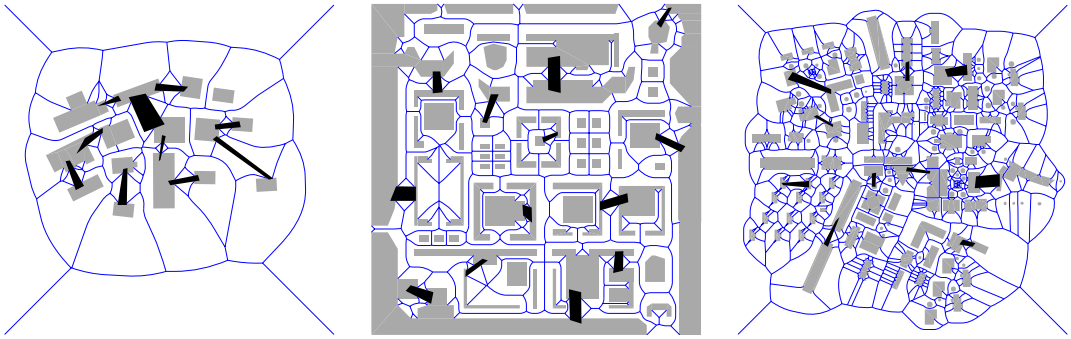


Figure 5: Graphs comparing the efficiency of our method ("Bounded") with a total rebuild and Van Toll's dynamic update method ("Local"). Note that the red line that depicts data of the local method is very close to the x-axis.



(a) (b) (c)

Figure 6: The black obstacles in these environments are examples of "door-like" obstacles. They obstruct the path between two obstacles. These are environment from of a) a military base b) Zelda c) a city.

of our method in comparison to a total rebuild increases however. We assume this is due to the relative size of the door-like obstacle becoming smaller as the environment grows. The more obstacles, the less the size of the updated area becomes, the more efficient our method is in comparison to doing a total rebuild.

7 Conclusions

With our method we wanted to improve the time efficiency of updating an ECM. We do this by calculating a temporary graph from a limited set of obstacles, after which we paste the relevant edges into the original graph. We have run experiments to test the time efficiency of our method. We observed that our method is faster than naively rebuilding the whole graph. Van Toll's method is steadily faster than our method, though in comparison to Van Toll's algorithm our version has the advantage of handling concave and overlapping obstacles.

We recommend using Van Toll's method for updates that include point, line and convex obstacles that do not overlap. In other cases (or when Van Toll's algorithm fails) our method is preferred, as it is usually faster than a total rebuild. If our method fails, a total rebuild is always a last option.

Our experiment with door-like obstacles tested our method in a realistic setting: closing and opening doors. Being able to close and open doors while quickly updating the ECM seems useful in simulations of virtual worlds. The door experiment shows promising results. This includes a speed-up in comparison to a total rebuild of about 8 times for our biggest environment. The runtime of our algorithm depends on the local configuration of an environment; there are too many variables in defining an environment (local complexity of obstacles, number of obstacles, number of influenced vertices, the degrees of those vertices, etc.) so we cannot really give an indication about expected running time of this algorithm.

8 Future work

We have already mentioned the possibility of parallelization of our method with the simulation. Calculating the ECM takes time in which the simulation currently cannot update. We can put the calculation of influenced components C in the original graph and C' in the temporary graph in a background thread. Meanwhile, the simulation can keep using the ECM. Swapping C for C' and updating the query structure is only a tiny part of our algorithm; it can be done in little time. This might keep the lag in the simulation due to an update to a minimum.

A way to improve the execution time of our method would be to create a query structure for finding obstacles. Currently the ECM includes a query structure for finding closest edges. There is no query structure for finding close obstacles yet.

In our algorithm we take a cut of the update obstacle and all other obstacles. We also take the intersection of a bounding box around the influenced area with all obstacles. As the number of obstacles increases, these cut and intersection operations will take more time. We could possibly decrease the time these operations take, by decreasing the obstacle set we do them on.

A query structure like an R-Tree to find the obstacles overlapping a boundary box would help; a bounding box around the update obstacle can be found quickly. We could then use this query structure to find the set of obstacles close to the update obstacle before taking the cut. We could also find the obstacles that overlap the bounding box surrounding the influenced area before taking the intersection. We suspect this might decrease the execution time of our algorithm.

References

[1] Geraerts, R. "Planning short paths with clearance using explicit corridors." Robotics and Automation (ICRA), 2010 IEEE International Conference on. IEEE, pp 1997-2004, 2010.

[2] van Toll, W.; Cook IV, A.; and Geraerts, R. "A navigation mesh for dynamic environments." Computer Animation and Virtual Worlds 23(6) pp 536-546, 2012.