AARHUS UNIVERSITY

Master's Thesis

Representations for Path Finding in Planar Environments

Author:
20062154
Andreas Koefoed-Hansen
u062154@cs.au.dk

Supervisor:
Associate Professor
Gerth Stølting Brodal
gerth@cs.au.dk

February 29, 2012

Abstract

In this thesis, different methods of representing planar domains are compared in the context of path finding. Three different representations are compared: grids, visibility graphs and navigation meshes. The first part of the thesis is a survey covering the theoretical details on how the representations are constructed, how graph search algorithms can be applied to find the shortest paths, and how the path finding process can be accelerated by using abstractions of the representations. In the second part of the thesis, the representations are evaluated by analyzing theoretical running times and actual running times of the implementations, the number of states, the length of the paths and the memory consumption.

Acknowledgements

I would like to thank my supervisor Gerth Stølting Brodal for excellent supervision, good advices and proof reading of my thesis.

Contents

Chapter 1

Introduction

Path planning is used to solve the problem of finding a path between two points in a graph. While navigation through graphs, that represent roads or other point-to-point connections, is widely used in real life applications, navigation through a terrain is mostly used in virtual environments. This is often computer games where agents have to get from a start point to a goal.

To use the *path planning* in real terrain, we need to collect data about the environment. This can be done using an airplane, which measures the height of the terrain by shooting laser beams down at the ground. It measures the time it takes for the laser pulse to be reflected back from the ground to the airplane. Knowing, that that the pulse is traveling with the speed of light, it is possible to calculate the distance to the ground. This technology is known as LIDAR [1], and can be used to generate three-dimensional high resolution grids or meshes. This data can be used to find the obstacles in the environment, and generate representations for *path planning*.

The algorithms used to create representations of three-dimensional maps, are much more complex and time consuming than the algorithms used in two dimensions. We can avoid this overhead by projecting the map to a plane before generating the representation. Figure 1.1 illustrates how this could be done. The gray areas in the grid are the sloped parts of the map, which we deem impassable. These areas are also marked in the planar grid, and when representations are made from the grid, we can plan paths around the marked areas.

To solve the problem of finding an optimal path, graph search algorithms are applied. One of the most used algorithms is *Dijkstra's algorithm* that solves the single-source shortest path problem for graphs with non-negative weights. The worst-case upper bound of the algorithm is $O(m + n \log n)$, where n is the number of vertices and m is the number of edges in the graph [2].

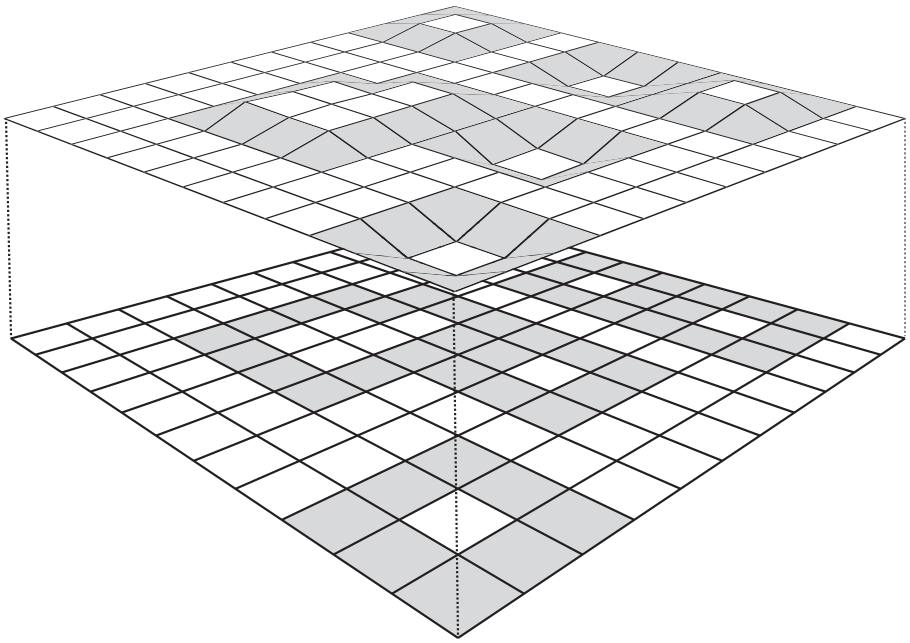


Figure 1.1: Three dimensional terrain projected to a plane.

For many applications, the speed of the path calculation is crucial. It could be in an environment where dynamic changes occur, and frequent recalculations are required. It is the complexity of the graph, which determines the running time. This means that we want to have as few nodes and edges as possible, while still having all the walkable areas in the environment represented. The graph is called a *connectivity graph*, and it contains all the valid moves in a representation. The nodes in the graph correspond to the nodes or states in the representation, and the edges define the possible moves or transitions between the nodes.

1.1 Representations

Before going into a brief introduction of the representations, a few definitions have to be introduced: A *map* defines the environment for which we create the representations. The *map* may contain a number of *obstacles*, either static or dynamic, described by a set of *constraints*. A *constraint* is a line segment that can be connected to other segments, in order to outline the *obstacles*. An illustration of the map elements can be seen in Figure 1.2.

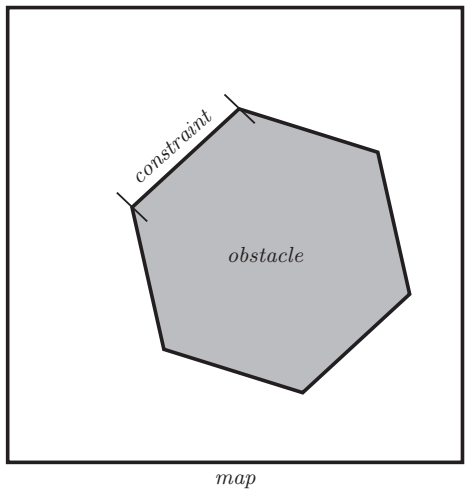


Figure 1.2: A *map* with an *obstacle* which is outlined by *constraints*.

1.1.1 Grid Representation

Grid based representations are composed of a lot of cells arranged in a grid. The *cells* are often rectangular and called *tiles*, but in general any type convex polygons which can be fit together in a grid, can be used as cells. Each cell has a cost associated with it that describes if the cell is walkable or not, and if it is, how much it would cost to move through it. A cell is connected to other cells in the grid by a number of portals. The number of portals varies, but normally 4 are used because it allows to move to the neighboring cells, which shares an edge with the current cell. When diagonal movement is required, we have to introduce more portals. The normal approach is to add 4 additional portals, one for each of the diagonal neighbors. The cells with 8 portals are called *octiles*. An illustration of the moves in the grid can be seen in Figure 1.3a.

1.1.2 Visibility Graph

A *visibility graph* $G = (V, E)$, where V is the set of vertices and E is the set of edges, can be constructed from a set of *constraints* and used for planning routes that does not collide with these *constraints*. The vertices in the graph are connected such that only those, which have a clear line of sight between them, are connected by an edge. An illustration of the moves in the graph can be seen in Figure 1.3b.

1.1.3 Navigation Mesh

A *navigation mesh* is any type of polygonal mesh used to represent the walkable areas of a *map*. The meshes can be created by triangulating the *map* while considering the *constraints*. The algorithms used to construct the mesh will depend on the complexity of the environment n. A uniform grid-based representation will rely on the extend of the environment and the resolution of the grid. This means that if we have a environment with large walkable areas, a mesh will have an advantage compared to a grid, where the areas would have to be split up in many cells [3]. An illustration of the moves in the mesh can be seen in Figure 1.3c.

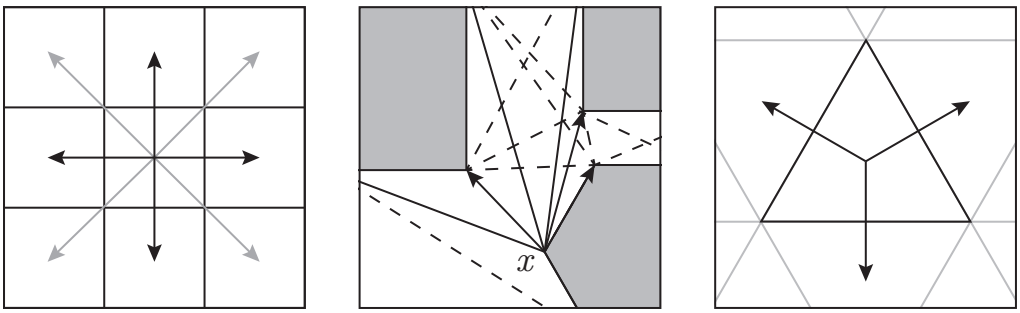
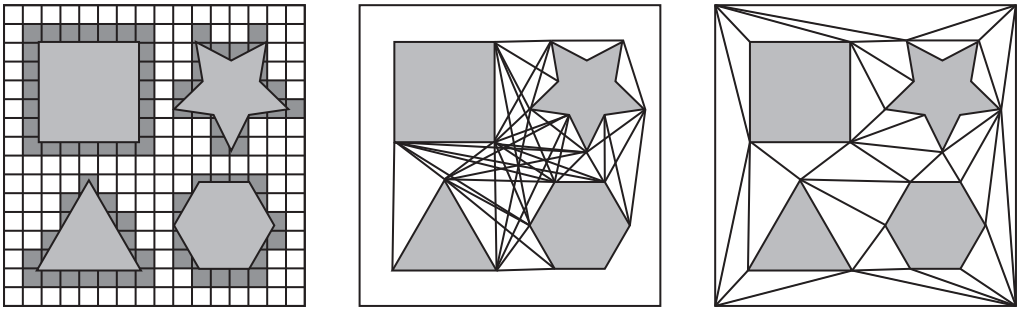


Figure 1.3: (a) The valid moves in a *grid representation*. The black arrows show the moves when using normal *tiles*, and the ones added when using *octiles* are colored in gray. (b) The valid moves in a *visibility graph* from the vertex x. (c) The valid moves in a *navigation mesh*.

Even though the three representations might be based on the same environment, the number of nodes and edges in the *connectivity graph* will most likely be different. Figure 1.4 shows an example of this. The *grid representation* always has the same number of nodes and edges regardless of how many *obstacles* there are. This is not the case with the *visibility graph* and the *navigation mesh*, where the complexity solely relies on the *constraints* in the *map*. The worst-case

scenario for the *visibility graph* is that there will be $O(n^2)$ edges, where n is the number of vertices used to represent *constraints*. The number of triangles and edges in the *navigation mesh* will always be $O(n)$.



(a) Grid representation. (b) Visibility graph. (c) Navigation mesh.

Figure 1.4: Different representations of the same *map*.

Chapter 2

Survey

In this chapter, we look at some of the contributions made to the academic community in the fields of artificial intelligence, map representations and path planning. This will also work as a build up to the following chapters, where some of the described algorithms will be implemented and compared.

2.1 Grid Representation

Grids are often favored in the choice of map representation for applications such as Real-Time Strategy (RTS) computer games. This is mainly due to the simplicity of the data structure and the way that the grid easily can be updated. Besides path finding, another thing often wanted in such games, is detection of unit collision. This can be archived by adding a flag to a cell to indicate if it is occupied or not. These flags will have to be updated as the units move around, but it will allow units to avoid colliding when the flags are considered in the graph search algorithm.

As mentioned in Section 1.1.1, *octiles* allow for diagonal moves. These moves can often reduce the path length, but allowing them also opens up for possible ambiguous representations. An example of this can be seen in Figure 2.1. Peter Yap [4] has looked into this problem and introduced an alternative way of aligning the *tiles* in the grid. Instead of all the tiles being aligned next to each other, every second column of *tiles* are moved half a tile-height down. The new cells in this representation are called *texes*, and each *tex* now gets up to 6 directly connected neighbors. In Figure 2.1, it is shown how it solves the problem of ambiguity. The empirical results [4] show that compared to the normal *tiles*, the use of *texes* improves the length of the paths by around 20%, and reduces the time for generating paths significantly, especially when there are a lot of *obstacles* on the *map* [4].

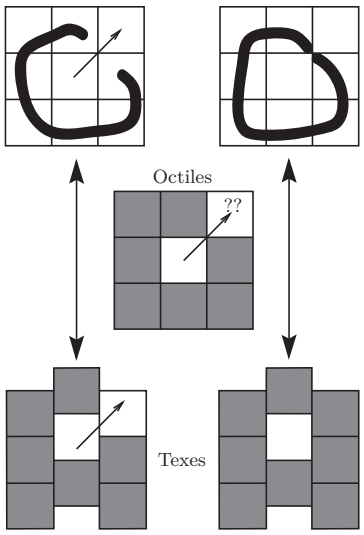


Figure 2.1: Two different scenarios for diagonal moves. The *octiles* represents each scenario the same way, while the *texes* represents each scenario differently.

Peter Yap later wrote a short article [5], where the *texes* were compared with both *tiles* and *octiles*. Rather than the length of the paths, it was the number of nodes expanded that was measured. The experiments were made on three types of maps: obstacle-free, from Baldur's Gate II and random data. Table 2.1 shows the results of the different maps where an $A*$ algorithm (see Section 2.4.2) has been used to find the paths. The results show that *octiles* outperform the other representations, and *texes* sometimes are better than regular *tiles*, but on random data, they are actually slightly worse.

Type	*Tiles*	*Texes*	*Octiles*
Obstacle free	100%	81%	70%
Baldur's Gate	100%	97%	74%
Random data	100%	105%	95%

Table 2.1: Number of nodes expanded compared to the number of nodes expanded in the *tile* representation. The values are calculated as $p = \text{expNodes}_{rep}/\text{expNodes}_{tile}$ [5].

When constructing the *grid representation*, the complexity relies on the resolution of the grid. Before adding the *constraints*, the grid has to be initialized. As we have to go through all the *cells* of the grid, the running time must be $O(r^2)$, where r is the resolution of the grid. This is under the presumption that the grid is squared. A *constraint* is inserted by locating the *cells* in the grid, which contains the end points of the *constraint* (see Figure 2.2a). This step

takes constant time. The next step is to mark all the *cells* that the *constraint* is intersecting. This is done by finding the gradient of the *constraint*, and using this to mark the *cells* between the two endpoints (see Figure 2.2b and 2.2c). The second step takes $O(r)$ because the *constraints* in worst-case spans the entire grid width or height. As this is done for all the *constraints* in the *map*, we get a total worst-case construction time of $O(r \cdot n)$, where $n = |C|$ is the number of *constraints* used to describe the *obstacles* in the *map*. If we assume that the *cells* that the *constraints* cover do not overlap, we access each *cell* at most once during the insertions of the *constraints*. This means that the worst-case time is bounded by the time it takes to initialize the grid.

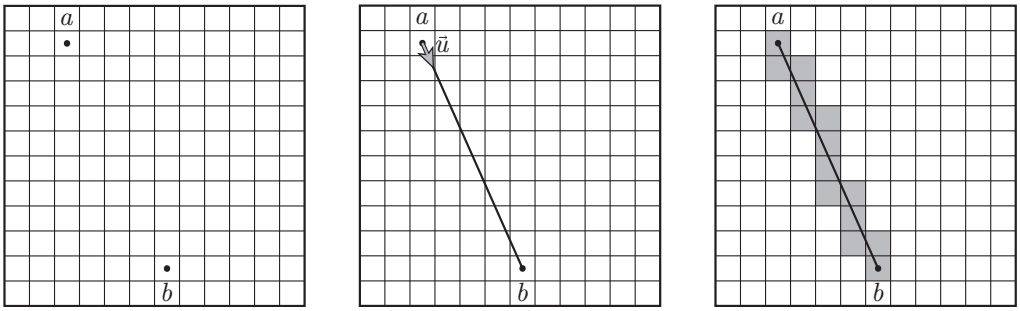


Figure 2.2: (a) Locate the *cells* that contains the endpoints. (b) Find the unit vector $\vec{u} = \vec{ab}/\|\vec{ab}\|$. (c) Locate the *cells* overlapped by the line ab by adding $\vec{u}$ to a.

2.2 Visibility Graph

Visibility graphs are not usually used to represent planar environments, but can be used as an alternative to *grid representations*. One advantage that *visibility graphs* have over grids is that the paths found in a *visibility graph*, always will be the actual shortest paths. The length and precision of the paths found in a *grid representation* will rely on the resolution of the grid.

The disadvantage is, that if the start and goal points do not exists in the graph, we will have to add them before finding the path. In most cases, it will take longer to add the points, than finding the path between them.

A *visibility graph* $G = (V, E)$ can be constructed trivially in $O(n^3)$ time [6], where $n = |C|$. The first step is to add all the endpoints of the *constraints* as vertices in the graph. Secondly, for each vertex $v \in V$ check if $u \in V$ is visible from v. If this is the case, an edge (u, v) is inserted into E.

The construction time can be reduced to $O(n^2 \log n)$ by using a *binary search tree* and a *cyclic plane sweep* (see Figure 2.3). The idea is, that for each vertex $v \in V$ we sort the other vertices $w \in C$ by the angle of the vector $v\vec{w}$. Then the sweep is initialized by inserting the *constraints* intersected by the right going horizontal ray, starting from v, into a search tree T. The sweep progresses clockwise and keeps the search tree updated, such that only the *constraints* that intersect the sweep line are contained. Each time a new vertex $w \in C$ is encountered, we check if it is visible from v, by looking in the search tree. The running time of a *cyclic plane sweep* is dominated by the time it takes to sort the vertices: $O(n \log n)$. Since we have to do a sweep for each of the vertices in V, the running time becomes $O(n^2 \log n)$ [6]. This method will only work if none of the *constraints* is intersecting. If the *constraints* are intersecting, we can split them up at the intersection points, which are found by a line-line intersection algorithm.

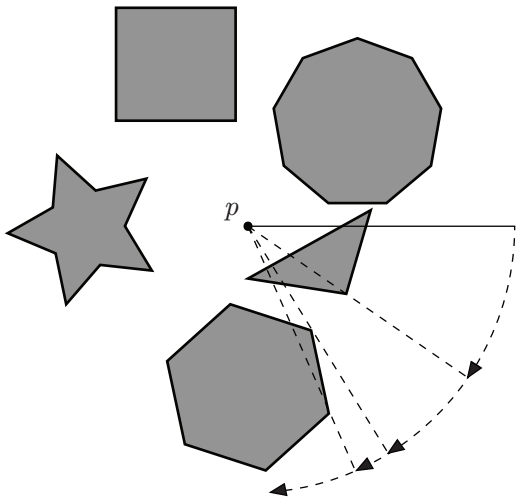


Figure 2.3: A cyclic plane sweep around the point p.

There have been introduced other faster algorithms, and an optimal algorithm that runs in $O(E + n \log n)$, where E is the number of edges in the resulting graph [7]. Because the upper bound on edges in a *visibility graph* is n^2, the running time may be $O(n^2)$ for some maps, but it is still an improvement.

2.3 Navigation Mesh

Navigation meshes combines the nice properties from the two other representations, by having a low number of edges per node, like the *grid representation*, while still having a linear number of nodes, like the *visibility graph*.

We will only look at triangular meshes in this thesis, but in general, a *navigation mesh* can be composed of polygons with an arbitrary number of sides, as long as they are convex.

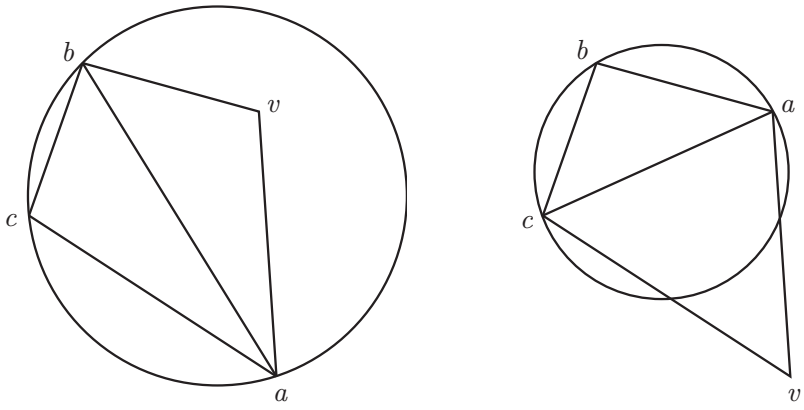
The triangulation of a *map* can be done in $O(n \log n)$ time, where n is the number of vertices inserted. The number of triangles is proportional to the number of inserted vertices, as each insertion of a vertex results in a constant increase in the number of triangles [6]. There are different approaches, on how to perform the triangulation, which have the same worst-case running times. The most common is to use either a sweep line or divide-and-conquer algorithm. The advantage of those algorithms is that we can guarantee a worst-case bound to be $O(n \log n)$ for every scenario. There are another kind of algorithms, which are based on randomization, that in some cases will get a running time of $O(n^2)$, but on average will perform more or less like the other algorithms. The randomized algorithms are incremental, which allows for insertion and removal of *constraints* dynamically, without having to rebuild the whole mesh.

An iterative algorithm by Kallmann et al. [8], has been designed specifically to be used for *navigation meshes*. It allows for dynamic updates in the mesh, both insertion and removal of *constraints*, that only affects the triangles in mesh locally. The output will be a *Delaunay Triangulation*, which will be described in the section below.

2.3.1 Delaunay Triangulation

In a *Delaunay Triangulation*, we aim at getting all the triangles to be angle-optimal. During the construction of the triangulation, the algorithm maximizes the minimum angle, such that we get as few "skinny" triangles as possible. This gives us a navigation mesh, where the triangles are as uniform as possible.

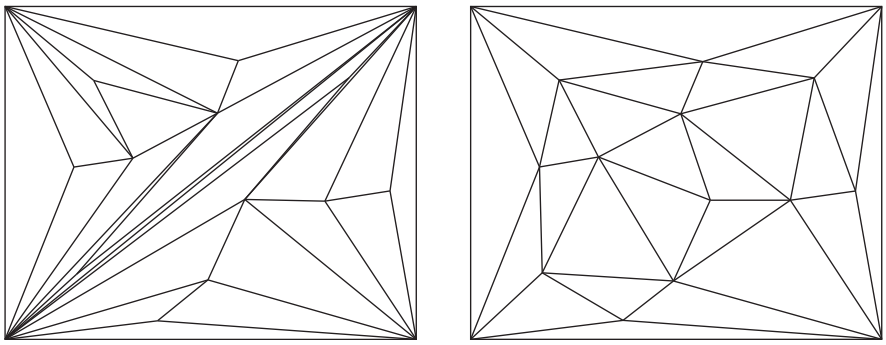
In order to ensure that the triangles are angle-optimal we use a circumcircle-rule. It checks all edges in the triangulation by the following procedure: For each half-edge e, get the triangle t, which e is a part of. If the circumcircle of t contains a vertex, e is illegal, but if no vertices are inside the circumcicle it is legal. If an edge is illegal, we flip it such that circumcircle of t changes, and there will be no vertices inside the new circumcircle [6] (See Figure 2.4). If the triangulation does not contain illegal edges, it is a valid *Delaunay Triangulation*.



(a) v inside the circumcircle. (b) v outside the circumcircle.

Figure 2.4: An example of a triangulation before and after an edge flip.

Using this technique, we can convert any triangulation into a *Delaunay Triangulation*. Figure 2.5 shows the difference between a triangulation before and after edges have been flipped.



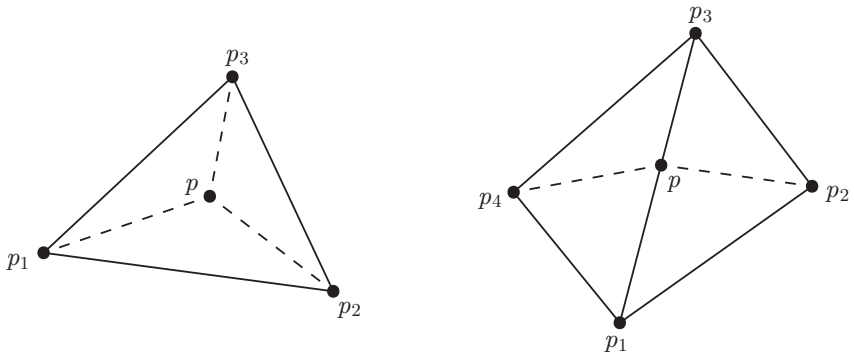
(a) Before flips. (b) After flips.

Figure 2.5: Triangulation before and after edges have been flipped.

Mark de Berg et al. [6] use a randomized incremental approach by Guibas et al. [9] when constructing a *Delaunay Triangulation*. The algorithm starts with one or more bounding triangles, which will encapsulate all the points P that will be added to the triangulation. Each point $p \in P$ will be added in random order using the following procedure: First, we find the triangle t, in which the point p is located. This is done by using one of the methods covered in Section 2.3.4. When the triangle is found, the point is inserted by creating three new triangles

and deleting t from the triangulation. The triangles will all share p as a corner point, and will have edges that connect it to the corner points of t (p_1, p_2, p_3) (see Figure 2.6a).

In some cases p will be located on an edge e in the triangulation. If this happens, we locate the two triangles t_1 and t_2 which shares e. The two triangles are removed from the triangulation and four new triangles are created. The new triangles will, like in the previous case, all share p as a corner point, and have edges connecting it to the points of t_1 and t_2 (p_1, p_2, p_3, p_4) (see Figure 2.6b).



(a) Insertion of a point in a triangle.

(b) Insertion of a point on an edge.

Figure 2.6: Insertion of points in a *Delaunay Triangulation*.

The final step is to check if the added triangles obey the circumcircle-rule as described previously. If this is not the case, we need to flip their edges. The edge flips also affects the neighboring triangles, and might invalidate the *Delaunay Triangulation*. To fix this, we also flip the edges of the neighboring triangles, and keep flipping edges until we get a valid *Delaunay Triangulation*. In worst-case, the insertion of a point may require $O(n)$ edge flips, but for randomized input the expected number of edge flips is constant [9].

The algorithm produces a number of triangles, which use a data structure called *SymEdge*. For each edge in the triangulation, two *half-edges* are made. The two *half-edges* each originates from a vertex, one from the first vertex of the edge and the other from the second. This makes the two *half-edges* point in opposite directions, and makes it possible to determine the previous and next *half-edge* that are a part of the triangle. The reason that the data structure is called *SymEdge* is that each *half-edge* has a pointer to its symmetrical edge (see Figure 2.7). With this information, we are able to figure out how the triangles in the mesh are connected. This structure is used for both point location

and navigation through the mesh. The *SymEdge* structure stores the same information as the *Double-Connected Edge List* (DCEL), used by Mark de Berg et al. [6].

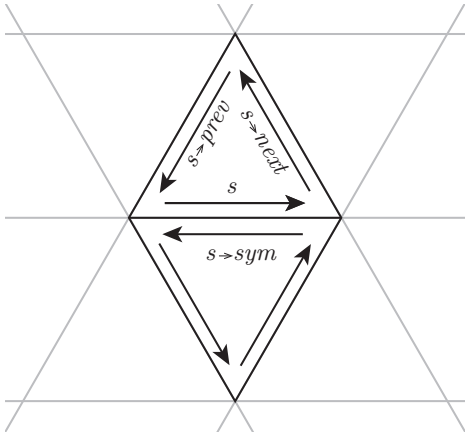


Figure 2.7: The *SymEdge* data structure.

2.3.2 Constrained Delaunay Triangulation

Given a set of *constraints* C, where each *constraint* $c \in C$ is a line segment, a *Constrained Delaunay Triangulation*, $CDT = (V, E)$, upholds the property that each *constraint* is contained in the set of edges E.

The first step of constructing a CDT is to build a $DT = (V, E)$ by inserting the endpoints of the segments. To ensure that the output will be a CDT we have to check that each constraint $c = (p_1, p_2)$ is contained in E. If the segment is not contained, we have to add it using the following procedure by Anglada [10]: First, we find all the triangles intersected by c. This is done by finding the triangles that have p_1 as a corner point, check if an edge of the triangle intersecting with c. If an intersection is found, the triangle that shares the intersected edge with the current triangle is returned as the next triangle to look at. If the line is passing through a triangle, there will be two edge intersections, and we have to determine which of the neighboring triangles to look at next. This is done by eliminating the choice of going back to the previous triangle. We keep doing this until the triangle that contains p_2 is found. Finally all the intersected edges are removed from E, and a polygon made up of the vertices that was connected by them, will be retriangulated.

The first part of the retriangulation, is to split the vertices into two sets, creating two *pseudo-polygons*, which share the *constraint* c (see Figure 2.8). The two *pseudo-polygons* are now triangulated individually using the function

TriangulatePseduoTriangle. It takes a set of vertices P, a *constraint* c, and the list of triangles in the triangulation T as input. If P has more than one vertex, the vertices are sorted clockwise by angle. Then the first vertex is assigned to the variable x. Each of the vertices $v \in P$ is tested to see if they are inside the circumcircle of the triangle (x, p_1, p_2). If v is inside the circumcircle it becomes the new x. The vertices in P is divided into two new sets P_E and P_D giving $P = P_E + x + P_D$. The function is called recursively with c being (p_1, x) and (x, p_2) and P as the two newly created sets respectively. If P is not empty, the triangle (x, p_1, p_2) is added to T. An example of a triangulation of a *pseudo-polygon* can be seen in Figure 2.8.

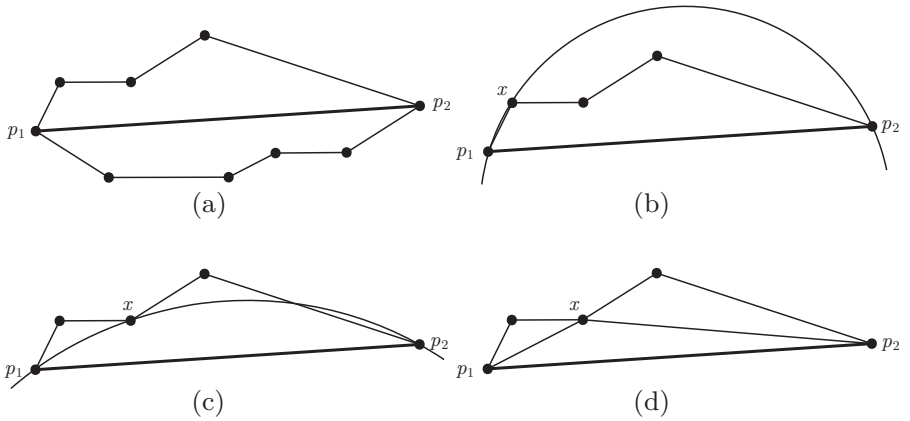


Figure 2.8: (a) The vertices are divided into two sets split by the segment. (b) The circumcircle for the triangle (x, p_1, p_2) are found, where x is the first vertex. (c) The circumcircle for the triangle (x, p_1, p_2) are found, where x is the second vertex. (d) As there are no other points within the circumcircle, the triangles are added to the solution.

An alternative approach of handling the insertion of *constraints* is presented by Bernal [11] that are based on the work of Lawson [12]. Instead of deleting edges and retriangulating the pseudo-polygons, the edges are flipped to remove intersections. The result of the two algorithms will be the same, however this approach will have the advantage that we at all times have a complete triangulation, in contrast to the other approach that uses deletion and reinsertion of edges. The flipping approach works as follows: As with the other algorithm, the intersected triangles and edges are found. Instead of deleting an intersected edge e, we find the quadrilateral q that is spanned by the two triangles, t_1 and t_2 that shares e. If q is convex, e is flipped and the next edge is processed. If q is concave, we cannot flip it, as it would create an invalid triangulation. Instead, we wait until the other edges of t_1 and t_2 have been processed and then process e afterwards. A final step is to optimize the edges with quadrilaterals that lie entirely on one side of the inserted *constraint*. The optimization flips

those edges if the triangles that share the edge will get a larger minimum angle. An example of the process can be seen in Figure 2.9.

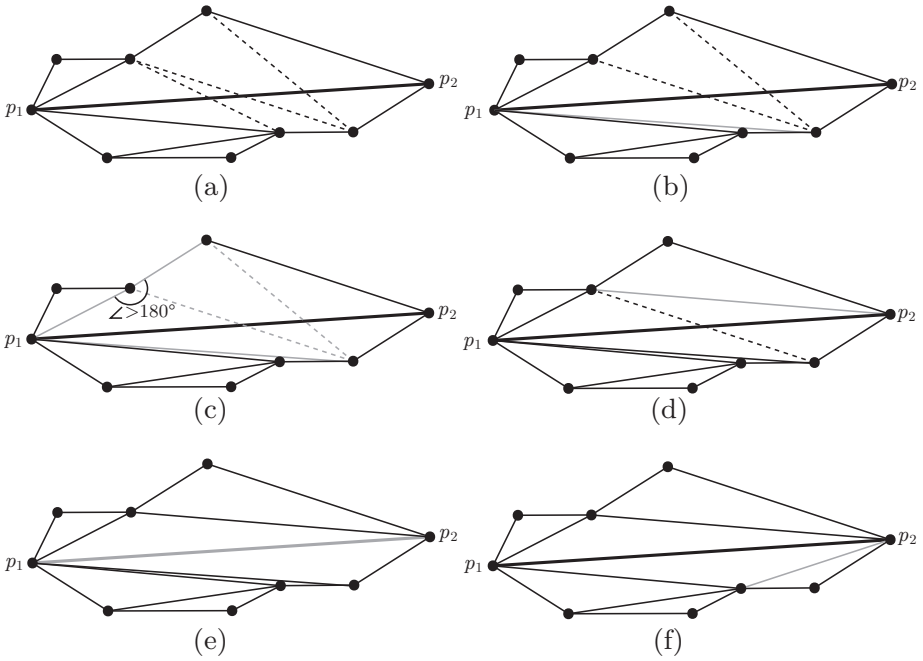
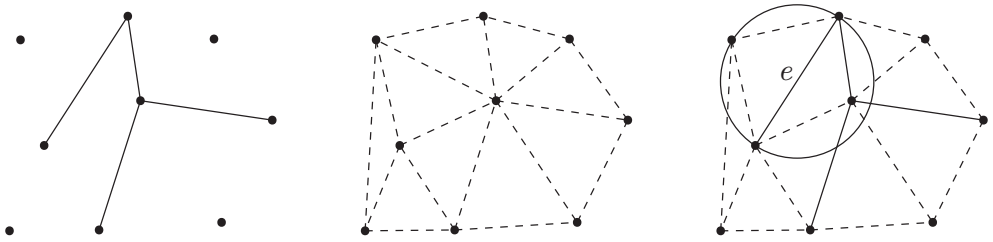


Figure 2.9: (a) The initial triangulation. The bold line from p_1 to p_2 is the *constraint*. (b) The first edge is flipped. (c) The next edge has a quadrilateral, which is concave, resulting in the edge being skipped. (d) The final edge is flipped. (e) Now that the skipped edge no longer has a concave quadrilateral, and we can flip it. It becomes the edge connecting p_1 and p_2. (f) Finally, the edges can be optimized, such that we maximize the minimum-angle of the triangulation.

All the triangles in the resulting mesh will have their minimum angle maximized, but we cannot ensure that they all will have Delaunay properties. Because some of the triangles have constrained edges that cannot be flipped, their circumcircle might contain other vertices (see Figure 2.10).

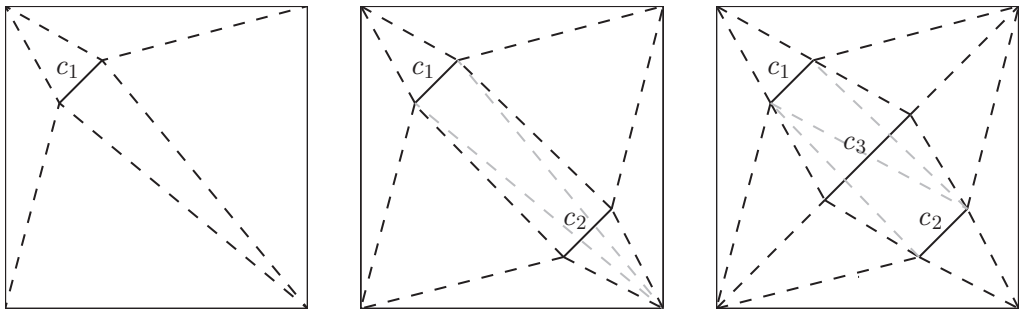


(a) Graph $G = (V, E)$. (b) DT of V. (c) CDT of G.

Figure 2.10: The process of creating a *Constrained Delaunay Triangulation*. Notice how the flipped edge e in the CDT results in the triangles not upholding the circumcicle rule.

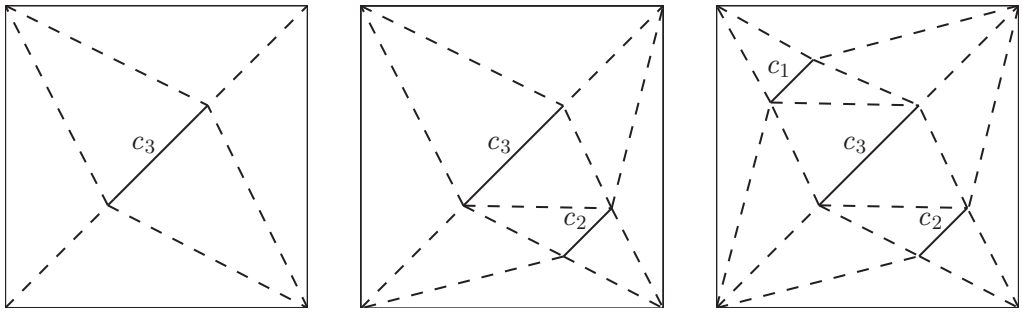
2.3.3 Analysis

Each time a point p is inserted in the *Delaunay Triangulation* it is located inside a triangle or on an edge of a triangle. In the first case, we add 3 new edges all going out from p. In the second case we add 4 new edges going out from p, but we remove the edge that p was lying on. In both cases, we get an increase of 3 edges for each point inserted. Each segment in a *constraint* has 2 points meaning that we in worst-case has to add 6 new edges, but as one of them will be constrained we only get 5 edges that we have to consider for the subsequent insertions. Assuming that none of the constrained edges intersects each other, we can argue that the order, in which the edges are inserted, has a great impact on the running time. The reason is that the number of intersections between a new constrained edge and the existing edges varies a lot, depending on the order of insertion. Consider the following example: We have three constrained edges that should be inserted, c_1, c_2 and c_3. In Figure 2.11 and 2.12, the three edges are inserted in two different orders. The order a results in a total of 5 intersected edges that have to be flipped, whereas b results in an order where no additional edges have to be flipped.



(a) Inserting c_1. (b) Inserting c_2. (c) Inserting c_3.

Figure 2.11: Incremental Delaunay Triangulation insert order *a*.



(a) Inserting c_3. (b) Inserting c_2. (c) Inserting c_1.

Figure 2.12: Incremental Delaunay Triangulation insert order *b*.

If we are unlucky, and have a bad order of insertion, we will get a worst-case time of $O(n^2)$, where n is the number of constrained edges, as the number of intersections will be an arithmetic series [2]:

$$\begin{aligned}\sum_{k=1}^{n} 5k+1 &= 6+11+\ldots+5n+1 \\ &= \frac{5n+2}{2}(n+1)-1 \\ &= O(n^2)\end{aligned}$$

In practice the algorithm will be faster than the worst-case time, but it is still an open question if it is possible to construct a *Constrained Delaunay Triangulations* using a randomized algorithm, with an expected running time of $O(n \log n)$ [13].

There exist divide-and-conquer algorithms that can construct *Constrained Delaunay Triangulations* in $O(n \log n)$ [8].

2.3.4 Point Location

The point location is a vital part of the *Delaunay Triangulation* algorithms. It is done for every inserted point and thereby has a great impact on the overall running time. There are different algorithms that vary in both running time and memory consumption.

Trivial Approach

Simply loops though all the triangles in the triangulation, and finds the triangle, which contains the point. It takes $O(n)$ time, and requires no extra memory.

Jump-and-Walk

The algorithm is by Münke et al. and used by Kallman et al. [8]. It works as follows: First, a random sample of $O(n^{1/3})$ vertices is chosen from the triangulation, where n is the number of vertices in the triangulation. Then each of the sample vertices is evaluated, to determine which one is closest to the point p. Finally an oriented walk towards p is performed, starting from the triangle t, adjacent to the found vertex. The oriented walk is done by selecting an edge e from t, which splits the center of t and p into two distinct planes. The triangle that shares the edge e with t is now selected, and the oriented walk continues until the triangle containing p is found. The original algorithm is only known to work for *Delaunay Triangulations*, and not *Constrained Delaunay Triangulations*. An example of a how non-delaunay triangulation can get the algorithm to go into an infinite loop can be seen in Figure 2.13. This cannot occur in a *Delaunay Triangulation*, as there is a strict ordering of distances between the centers of circumcircles, c_1 and c_2, of two triangles, t_1 and t_2, to the query point q.

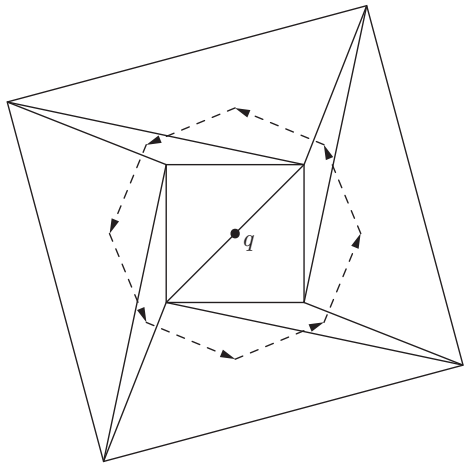


Figure 2.13: An infinite oriented walk.

Kallman et al. [8] modified the algorithm a bit, to ensure that it also works on *Constrained Delaunay Triangulations*. As described earlier the problem with *Constrained Delaunay Triangulations* is that we cannot ensure that all triangles uphold the *Delaunay properly*. This can make the algorithm visit triangles more than once, and in some cases enter an endless loop, where the same triangles are visited over and over. To avoid this, an integer is incremented for each point location search, and used to update a value on each of the visited triangles. This value is used as a flag to determine if the triangle has been visited in this search, if so; the algorithm will not consider it when doing the oriented walk. The algorithm takes $O(n^{1/3})$ time and requires no data structure, hence no extra memory is needed.

Sector Based Jump-and-Walk

Demyen et al. [14] use a modified version of the *jump-and-walk* method that relies on a 2D grid structure. The grid has a predefined resolution, and each *cell* covers a sector of the *map*. Each sector has a pointer to the triangle, which contains the midpoint of that sector. When searching for a triangle, which contains the point p, the algorithm uses the grid to find the sector, in which p is located (see Figure 2.14).

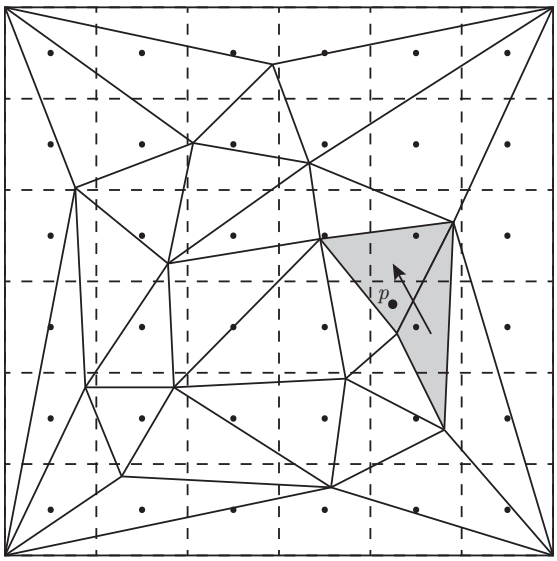


Figure 2.14: *Sector Based Jump-and-Walk*: The sector grid.

The time to find the sector is constant, and if the triangles are divided equally across the sectors, we get an expected running time of $O(n/s)$ where s is |sectors|. We can however, get triangulations where almost all triangles are located in one sector, and the running time of a search in that sector will be

$O(n)$. For random queries, the average time will still be $O(n/s)$ for those *maps*. This is because, query points located in sectors with few triangles, will need few steps in the walk, to get to the triangle containing the point. The algorithm requires a static data structure that uses $O(s)$ space.

DAG Structure

Mark de Berg et al. [6, p. 202] use a *Directed Acyclic Graph* (DAG) by Seidel [15] to locate a triangle containing the query point. The DAG is initialized with a number of nodes corresponding to the triangles in the initial triangulation. When a point is inserted and new triangles are created, the DAG is updated to reflect the changes in the triangulation. The node that represents the triangle, that contains the new point, has three new child nodes added, which relates to the new triangles that replaces the original triangle. When an edge is flipped, and two new triangles are added, both of the old triangles will get them as children. An illustration showing an insertion of a point, and an edge flip, can be seen in Figure 2.15.

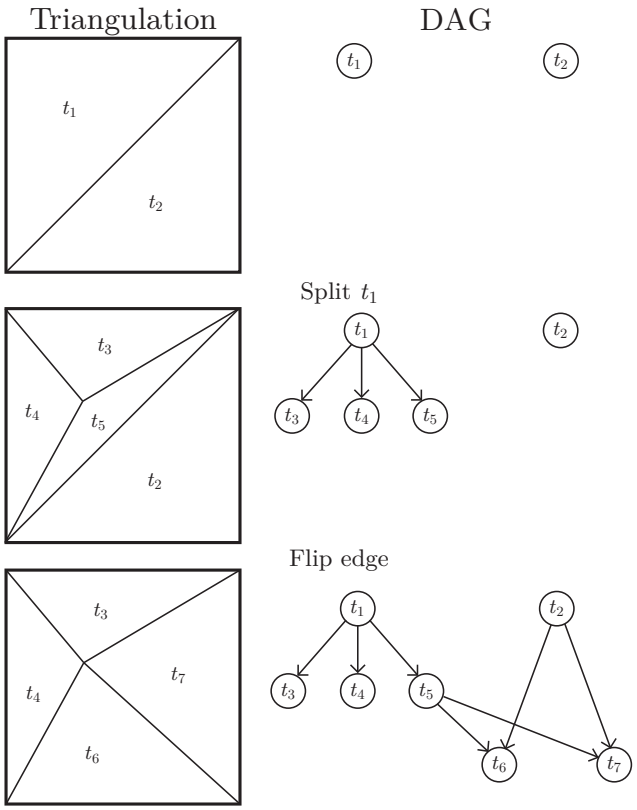


Figure 2.15: DAG structure.

The construction of the data structure, is done incrementally along with the triangulation, and do not add an overhead to the overall running time. Searching for a triangle is expected to take $O(\log n)$, if the DAG is balanced. As with other randomized algorithms, we can get unlucky, and the depth of the DAG can be linear in the number of nodes. In such a case, the running time becomes $O(n)$. The DAG stores information about all the triangles created during the triangulation process, which is expected to be $O(n)$. This means that the memory consumption is $O(n)$.

2.4 Graph Search Algorithms

The purpose of graph search algorithms, is to find the shortest path from one node in a graph, to either a set of goal nodes (*single-source shortest-paths*), or just a single goal node (*single-pairs shortest-paths*). The worst-case running time in both scenarios are $O(m+n\log n)$, because even if we only need to find a path between a pair of nodes, we might have to visit all the nodes in the graph, to find the path [2]. To achieve this bound, the algorithms rely on a *queue* to keep track of what vertices to explore next. The *queue* support three operations: insert, decrease key, and extract min. The worst-case time for insert and extract min operations must be $O(\log n)$ and $O(1)$ for the decrease key operation, in order for the algorithm to be able to obtain the worst-case bound of $O(m+n\log n)$.

There are several requirements to the path returned by a *graph search algorithm*. First, the path cannot make the agent reach an invalid state, which means, that no *constraints* can ever be intersected by the path. This requirement might seem obvious and intuitive, as we as humans would never consider a path that passes through *obstacles*, such as walls, to be valid. This might be why such flaws, when noticed in computer games, seems quite funny to look at. This could be when a non-player character (NPC) continuously runs into an *obstacle*, instead of trying to get around it. However, this problem often reflects a flaw in the representation, rather than an error in the *graph search algorithm*. Another requirement is that the path should be optimal. The definition of optimal may vary for different applications, but normally it means that the path has the shortest length, or it takes the shortest time to traverse. In some representation these two might be the same, if all weights on the walkable areas in a representation have the same cost, in respect to time it takes to move through the areas. If not, the path should preferably lead the agent around areas with increased traversal cost, such as mud holes and other difficult terrain. Another optimality constraint could be, not to be spotted by enemies, which would mean that the path should not intersect the line of sight of the enemies.

We will look at two graph search algorithms: *Dijkstra's algorithm* and *A**.

2.4.1 Dijkstra's Algorithm

Dijkstra's algorithm [16] solves the *single-source shortest-path* for directed graphs, $G = (V, E)$, with non-negative weights. The algorithm keeps track of visited vertices by adding them to a set S. The inputs for the algorithm are: a graph G, the source $s \in V$ and a set of weights w, that contains the cost of traveling between the vertices in V.

The algorithm evaluates the input as follows: While there are vertices left in the minimum-priority queue $Q = V - S$, extract the vertex v with the smallest shortest-path estimate from s, called v_e, and add v to S. Loop through all the adjacent nodes of v and relax them. The relaxation simply checks if the adjacent vertex u, has a shortest-path estimate u_e that is greater than $u_{e'} = v_e + w(u, v)$. If this is the case u_e is updated to the new estimate $u_{e'}$. When Q is empty, we have calculated the shortest path from s to every other node in G.

Figure 2.16 contains a small example illustrating how the algorithm works. In this example, we use the algorithm for a *single-pairs shortest-paths* search, and interrupting the search as soon as we reach the goal point.

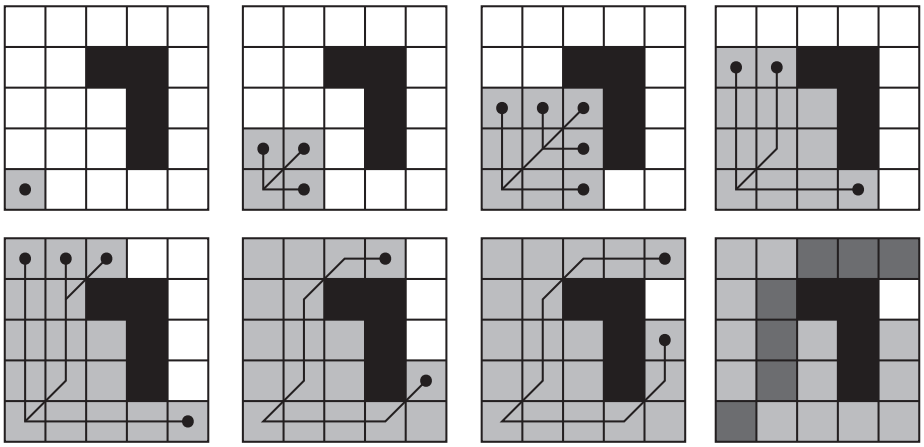


Figure 2.16: Example of *Dijkstra's algorithm*. The nodes colored in light gray are the visited nodes, the dotted nodes are the nodes to be evaluated next, the lines going out from the dots are the estimated paths, and the dark gray nodes denotes the final path.

2.4.2 A*

Dijkstra's algorithm only looks at the cost of the outgoing edges when determining the next vertex to visit. Other algorithms such as A^* use heuristics, to make smarter choices when determining which direction to search in a graph. This approach is only beneficial when we know both the source and goal node in the graph, which makes A^* a *single-pairs shortest-path* algorithm.

Given a graph $G = (V, E)$, a start node $a \in V$, an arbitrary node $v \in V$, and a goal node $b \in V$, we can estimate the length of the path from a to b, going through v, using the function $f(v) = g(v) + h(v)$. The function $g(v)$ measures the cost of going from a to v, and $h(v)$ estimates the cost of going from v to b. The algorithm starts out by looking at a's neighbors, and uses $f(v)$ to estimate the cost of each of them. The neighbor with the lowest cost is chosen to be the next node to look at. Now all the not estimated neighbors are estimated, and next node is selected. This process continues until the goal node is reached. The path to be returned is extracted by backtracking from the goal node.

To ensure the algorithm works properly and we get optimal paths, we need to make sure that the heuristics function is *admissible*. This means that the function will never overestimate the cost of traveling from a node to the goal point. This ensures that the A^* search is complete, and that when we reach the goal point, we will have the shortest path. Another property that a heuristic function can have is to be *consistent*. For a heuristic to be *consistent*, it needs to ensure that when we move from one node to another, we cannot get closer to the goal by more than the cost of traveling between the two nodes. If a heuristic is *consistent*, it is *admissible* as well. [17].

In this thesis, we will use the *Euclidean distance* heuristic that is known to be *admissible* for Euclidean graphs.

A^* can be a lot faster than *Dijkstra's algorithm*, given a good heuristic. While *Dijkstra's algorithm* keeps expanding nodes that are unlikely to be a part of the shortest path, A^* can skip these, and instead focus on the relevant nodes determined by $h(n)$. An example of how the algorithm works can be seen in Figure 2.17. Compared to Figure 2.16, fewer nodes are visited and hence less computation time is needed. A larger example can be seen in Figure A.1 in the appendix, where the two search graph algorithms are used on the same grid.

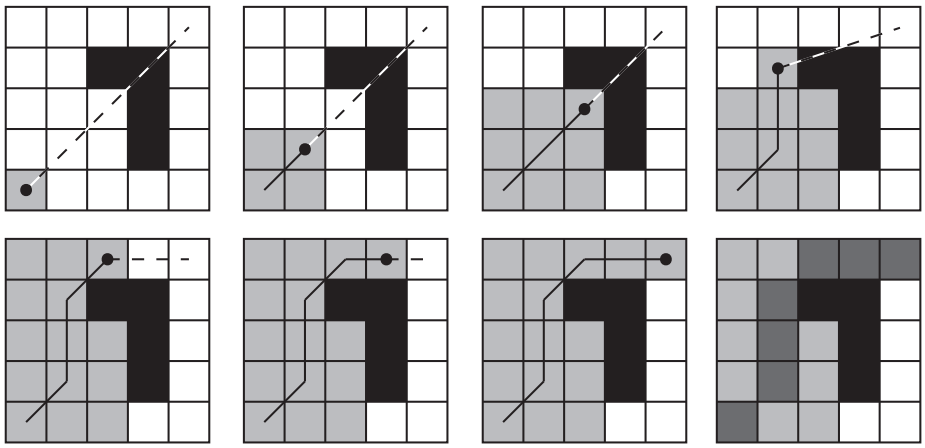


Figure 2.17: Example of the A^* algorithm. The nodes colored in light gray are the visited nodes, the dotted node is the node to be evaluated next, the solid line going out from the dot is the path from s to v, the dashed line is the estimate of going from v to b, and the dark gray nodes denotes the final path.

2.4.3 Path Planning in Navigation Meshes

Path planning in *navigation meshes* is a bit more complex, compared to *path planning* in the other two representations. The difference is that we do not get the exact path from just using a graph search algorithm on a mesh, as we get in the other representations. Instead, we get a list of triangles that the path passes through, and these triangles form a *channel* (see Figure 2.18a). To get the actual path from the *channel*, we have to use a *funnel algorithm* [18], which is described later.

Channel

To find the *channel* we need to generate a *connectivity graph* for the triangles in the mesh. This graph is used by the graph search algorithm to find the triangles, which are traversed by the shortest path from a to b. The graph can be constructed simply by locating the triangle, which contains the start point p, and creating connections from p to the middle points of the non-constrained edges. The graph keeps expanding by creating connections to the middle points of the edges, in the triangulation. An example of such a graph can be seen in Figure 2.18b. This graph is generated on the fly while the graph search algorithm is progressing, such that only the relevant edges are added to the graph.

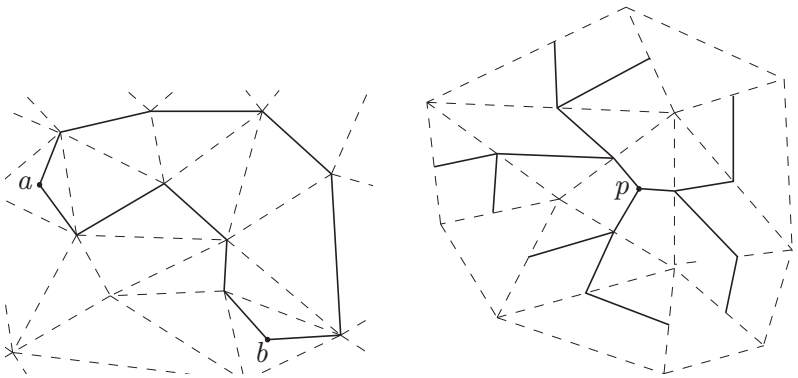


Figure 2.18: (a) A *channel* (solid lines) going from a to b. (b) A *connectivity graph* (solid lines) going out from p.

Kallman [19] uses the middle points of the edges to estimate the paths in a triangulation, but this does not always give the shortest path. In Figure 2.19, it is illustrated how the shortest path from a to b, is different from the estimated path.

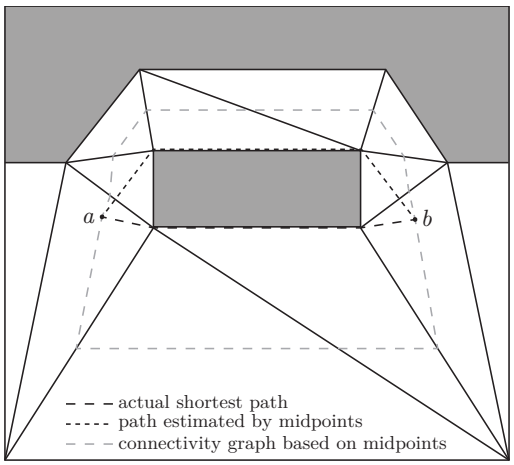


Figure 2.19: An example where the estimation using middle points fails to produce the shortest path.

Demyen et al. [14] recognized this problem, and made a modified version of A^* algorithm called *Triangulation* A^* (TA*). It works almost like a regular A^* algorithm, but there is a difference in the heuristic function. Instead of using the middle points, it uses the points on the edges, which have the shortest distance

to the goal point. This heuristic is *admissible* as it uses the *Euclidean distance*, which means that we always get the channel that contains the shortest path [14].

Funnel

When the triangles have been found, we need to extract the exact path. As the *channel*, which the triangles form, is not necessarily a convex polygon, we cannot just connect a and b directly. To handle all types of *channels*, we use a *funnel algorithm* that can find the shortest path within any polygon in $O(n)$ time, where n is the number of triangles in the *channel*.

The *funnel algorithm* uses three elements:

- A *path*, which contains the points known to be part of the final path, at a given point in the algorithm.
- A *funnel* that represents the area yet to be processed, and in which all shortest paths must lay. It is divided into two parts: The lines turning clockwise (right, with respect to the *apex*), and the lines turning counter-clockwise (left, with respect to the *apex*). The *funnel* is represented by a *deque* structure (double-ended queue).
- The *apex*, which is the vertex connecting the *path* and the *funnel*.

These three elements are illustrated in Figure 2.20a.

The algorithm begins by assigning the *apex* to be the start vertex a. The *funnel* is at the start expanded by the two vertices of the first *internal edge*. The *internal edges* are the ones spanning the *channel*. The rest of the *internal edges* are now added one by one, in the following way:
One of the two vertices from the edge is ignored, as it already exists in the funnel. The other vertex will be added to either the front or the back of the *deque*, depending on which side of the funnel it is. The *wedge*, in which the new vertex is contained, is located, and all vertices between the root of the *wedge* and the new vertex, are popped from the *deque*. The *wedges* can be seen in Figure 2.20b. If the *apex* is popped, the vertex next in line becomes the new *apex*. Each time the *apex* is updated, the old *apex* is added to the *path*. When all the *internal edges* have been processed, the goal vertex is added to either side of the *funnel*. The resulting path will consist of the vertices in *path*, and the vertices in the side of the funnel, to which the goal point was added. The process of adding a new vertex is shown in Figure 2.21a and 2.21b.

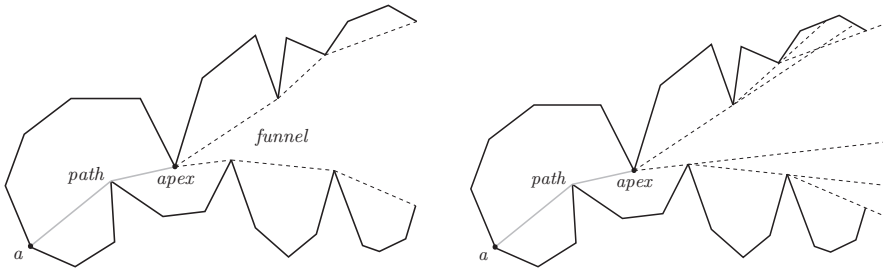


Figure 2.20: (a) The elements of the *funnel algorithm*. (b) The *wedges* (dashed lines).

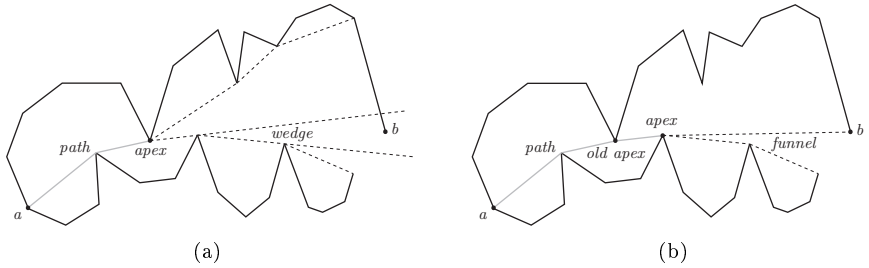


Figure 2.21: (a) The goal vertex is added, and the *wedge* containing it, is located. (b) The vertices from the *deque* between the root of the *wedge* and the goal vertex b is removed, and the *apex* is updated.

2.5 Abstractions

When a representation contains many nodes, it can become a problem to locate an optimal path within a small time span. This can be resolved by using abstractions that stores meta data about the actual representation. There can be multiple abstraction levels that describe the same *map*, but with different level of detail. The abstraction levels are ordered in a hierarchy, such that the most abstract level (the one with fewest nodes) is searched first, and the least abstract level (the one with the most nodes, besides the original representation) is searched last.

There are a number of advantages gained by using abstractions. The most obvious one is the reduced search space, and hence less time required to calculate the paths. The reduction in search space is due to the hierarchical search, where we only do an exhaustive search on the most abstract level. If a path is found, we use the nodes from the path to limit the search space for the next abstract search. Another advantage is, the possibility to quickly determine if a

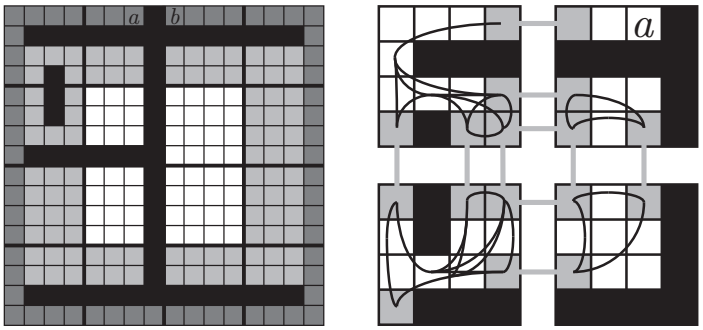
path exists or not by using the abstractions, rather than the original representation. This can save a lot of computation power, compared to an exhaustive search.

The different abstraction methods used for the representations are described in detail in the following subsections.

2.5.1 Hierarchical Grid Representations

Botea et al. [20] have come up with a hierarchical approach, for path finding in *grid representations*. The basic idea is to create abstraction levels, which store information about regions in the original representation. Each abstraction level divides the original *map* into *clusters* with predefined sizes. Figure 2.22 shows how a 16×16 *map* can be divided into 16 *clusters*.

The clusters are connected through a set of entrances that is located on the borders between the *clusters*. An entrance is defined by two *tiles* in the abstraction. These two *tiles* define the span of the entrance, which can be as large as the side of a *cluster*, or in some cases just one *tile*. Given the knowledge about how the *clusters* are connected, we can apply graph algorithms and find the shortest path from one *cluster* to another (see Figure 2.22b). When we know the route through one abstract level, we can use the abstract path to find a more refined path, using a more detailed abstract level. For each abstract level the path gets more and more precise, and when finally reach original grid, the path will be as exact as possible. The final path can be found by exploring a very limited search space, outlined by the abstract path from the least abstract level.



(a) (b)

Figure 2.22: (a) *Map* divided into *clusters*. The light gray area denotes the abstract path, and dark gray area is the actual path. (b) The connections between the clusters (gray), and internal connections between the entries (black).

Without the abstraction levels, we would have to store the entire detailed grid in memory, but with the *map* divided into *clusters*, we only need to have the parts of the *map* represented by one *cluster* in memory. One final advantage is, that the path can be calculated one step at the time. For applications such as computer games where response time is crucial and a full-length detailed path is not needed instantly, this is ideal. The first part of the path will be calculated quickly, while the rest of the calculations are postponed, and calculated on demand.

The results of this approach show, that for long paths the number of visited nodes is reduced to 10%, compared to the original *grid representation* [20]. For shorter paths, the difference is not that noticeable. The paths found using the hierarchical approach, will be exactly the same as the paths found using the original *grid representation*.

2.5.2 Highway-Node Routing

For *visibility graphs*, we have to use a different method. The abstractions are created by removing the least important vertices from the graph, using a technique called *contraction*. When a vertex v is *contracted*, all its edges will be removed and new *shortcut* edges will be added between the vertices that was connected through v, such that a path $\langle u, v, w \rangle$ becomes $\langle u, w \rangle$. If we divide the abstractions into multiple levels, we get *Contraction Hierarchies* [21].

The hierarchical levels consist of *overlay graphs*, $V_1 \supseteq V_2 \supseteq \cdots \supseteq V_L$, where V_1 has the most vertices and V_L has the fewest. The distance between corresponding vertices in different levels will be the same, even though some of the vertices of the low-level paths might be gone in the high-level paths. This approach is called *Highway-Node Routing* (HNR) [22]. The *overlay graphs* are constructed by iteratively removing the least important vertices by *contraction*. The importance of a vertex is estimated by using a heuristic, where multiple factors are involved. The most prominent factor is the *edge difference*, which denotes the number of edges in the graph before and after the vertex is removed. Among the other factors, some ensures uniform picking of vertices to be *contracted*. This is done to avoid large "holes" in the *overlay graphs*.

Queries are performed by using a bi-directional search. The forward search begins at the start node a and expands towards higher level nodes, while the backwards search starts from the goal node b and expands towards lower level nodes. If a shortest path exists from a to b, the two searches will meet up at a node v, which will have the highest order of all nodes in the resulting path. The search will never move downwards in the hierarchy, which means that the search space is greatly reduced. The resulting path will be the optimal path.

The speedup gained using this method is tremendous. Experiments on a graph containing information about the road network in Western Europe, shows

that this HNR combined with other methods can improve query times with a factor of more than 30000, compared to *Dijkstra's algorithm* [22]. There are other methods described in the same article, which improves query times even more (up to a factor 3 million), but these require more pre-computation time and memory.

2.5.3 Triangulation Graph Reductions

Demyan et al. [14] have developed a technique to create abstractions of triangulations. This is done by having a node for each triangle in the representation. The nodes are connected in an *abstract graph* that reflects the topological structure of the *map*.
The goal is to partition the triangles into structures, which are: decision points, corridors, dead ends or islands. The structures are identified by categorizing the nodes from level 0 to 3 inclusive, on each triangle. An example of an *abstract graph* can be seen in Figure 2.23. As [14] is a conference article, the abstractions are not described in much detail, due to the lack of space. A much more detailed description is given in the master's thesis [23] by Demyan.

A short description of the different node types can be found below, along with an example of a triangulation that has been categorized (Figure 2.24).

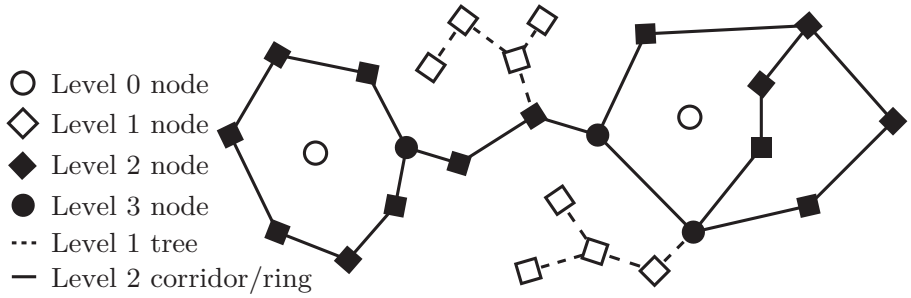


Figure 2.23: An abstract graph.

Level 0 Nodes

These nodes are related to triangles that have all their edges constrained. These are sometimes called islands and do not have any connections to the other nodes in the *abstract graph*. If a start point is contained in one of these, there can only be a valid path if the goal point resides within the same triangle.

Level 1 Nodes

They denote "dead ends" and are often connected to other level 1 nodes. A triangle is labeled as "dead end" if it has either:

- two constrained edges and it is indeed a dead end,
- one constrained edge and it is connected to a level 1 node, or
- no constrained edges and it is connected to at least two level 1 nodes

Adjacent level 1 nodes form trees, which has a root that connects the tree to other types of nodes. These trees can be ignored in the path finding process, if neither the start nor the goal node is located in the tree.

Level 2 Nodes

This type of nodes only has one constrained edge. They define "corridors" and can form rings of triangles as well. Level 2 nodes can be roots for the level 1 trees and often works as a link between level 3 nodes that denotes "decision points". If both the start and goal node are in level 2 nodes we check if they are in the same corridor. If this is the case, we can determine that a path is possible, but we still have to check outside the corridor to be sure to find the shortest path.

Level 3 Nodes

These nodes are "decision points" and are identified by triangles which have no constrained edges. These nodes are used as vertices in a graph called the *most abstract graph*, which only contains level 2 and level 3 nodes. If there is a corridor of level 2 nodes connecting two level 3 nodes, there will be an edge between those vertices in the most abstract graph.

Searching in the Abstract Graph

As with a normal path search in a triangulation, the first step is to locate the triangle in which the start point lies. However, when searching in the abstraction we also need to know which triangle the goal points lies in to get the abstract node associated with it. The point location process can be rather costly, depending on which method is used (see Section 2.3.4). Demyan et al. [14] found that when they experimented with their modified version of A^*, called *Triangle Reduction* A^* (TRA^*), the point location step was talking up a lot of the time, compared to the time of actually finding a path. Because of this, they came up with the *Sector Based Jump-and-Walk* method, which greatly reduced the time spend on point location.

During the labeling of the triangles in the graph, the related nodes also keeps information about which component they are connected to. The generation of

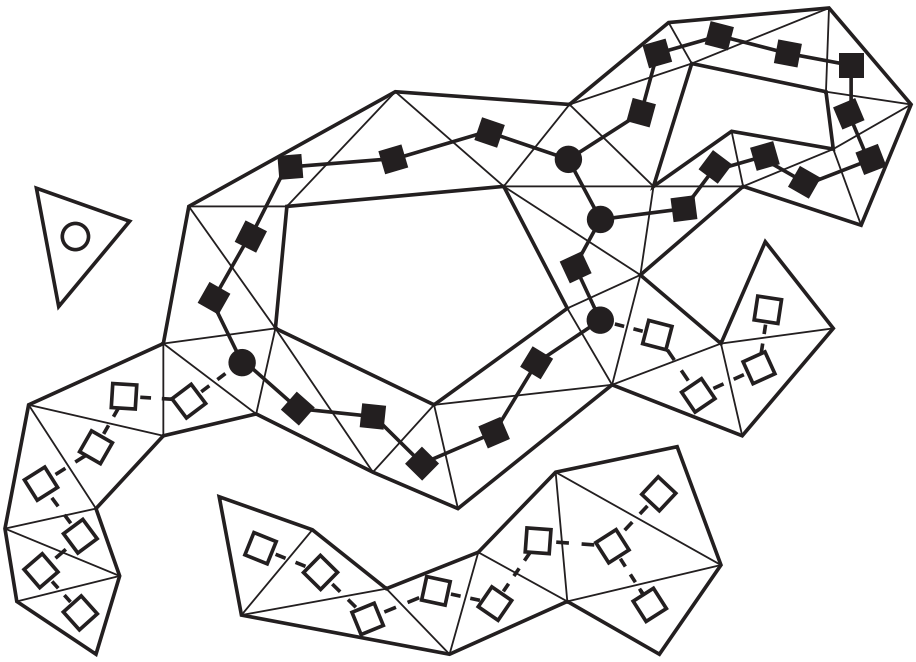


Figure 2.24: Categorized triangulation.

the abstract graph starts by identifying the level 0 nodes and giving them a unique component id. The other nodes will also get a component id assigned, but they will share it with the triangles to which they are connected. This gives the possibility abort the path finding process, if the start and goal point are located in triangles with different component ids.

There are some special cases, where both the start and goal point are located in the same part of the *map*, and the *channel* can be constructed without using graph search algorithms. This could be when the two points are located in the same tree of level 1 nodes, or are in the same corridor or ring composed by level 2 nodes.

If none of the special cases apply we need to use the TRA^* algorithm to find the triangles connecting the points. The first step is to locate the nearest level 3 node for both the start and goal node. When we know the two level 3 nodes, we can use the *most abstract graph* to search for the shortest path between the two nodes. The path between the two level 3 nodes combined with the paths to reach the nodes from the start and end point, will form the final path of triangles. The paths found using this method will always be optimal, as the edges in the abstract graph will reflect the cost of actually moving through the triangles between the level 3 nodes.

The speedup gained by this method varies depending on the path length. Experiments performed in the thesis [23] by Demyan show a 170 times speedup at best.

Chapter 3

Implementation

The implementation is created in Visual C++, using Microsoft Visual Studio 2010 as IDE and compiler.
The following algorithms and data structures have been implemented in order to tests the different representations:

- **Grid Representation:**
  $O(r^2)$, r = resolution of the grid (see Section 2.1)
- **Visibility Graph:**
  $O(n^3)$, n = number of vertices to be inserted (see Section 2.2)
- **Navigation Mesh:**
  $O(n^2)$, n = number of vertices to be inserted (see Section 2.3)
  - **Trivial Point Location:**
    $O(n)$ (see Section 2.3.4)
  - **Sectors Based Jump-and-Walk Point Location:**
    $O(n/s)$, $s = |\text{sectors}|$ (see Section 2.3.4)
- **A*:**
  $O((m+n)\log n)$, $m = |E|$, $n = |V|$ (see Section 2.4.2)
- **Funnel Algorithm:**
  $O(n)$, n = number of triangles in channel (see Section 2.4.3)

The rest of this chapter contains a short description of each of the algorithms and data structures used.

3.1 Common Data Structures

To represent a floating-point number, a wrapper called *Real* is used. This wrapper contains a double precision floating-point number, and implements various comparator and arithmetic operations. Each time two *Reals* are compared, the

absolute difference between the two doubles are calculated. They are deemed equal if the difference is less than a predefined *epsilon* value.

A point in the plane is represented by the *Point* class. A *Point* contains two *Reals*, x and y.

To represent the *constraints*, a *Line* class is used. A line has two pointers to *Points*, one for each end point.

3.2 Grid Representation

The grid is represented by an array that contains pointers to the grid nodes. The size of the array s is determined by an input parameter which contains the resolution of the sides of the square grid r: $s = r^2$. The array entries are initialized to zero and only when a value is set, a node is created and a pointer to it is placed in the entry. The value can be set either by adding a *constraint* or by the graph search algorithm, when it progresses across the *map* to find a path.

3.3 Visibility Graph

The *visibility graph* constructed by a simple $O(n^3)$ algorithm, which checks every possible connection between the points to see if they intersect the constrains. Because the *constraints* do not have any width, the nodes in the graph can contain edges from both sides of the contained edge. This will result in invalid paths if graph search algorithms are applied (see Figure 3.1a). To resolve this, one or more graph nodes are created for each point. The number of created graph nodes corresponds to the number of *constraints* meeting up in the point. The number of created graph nodes will always be twice the number of the *constraints*, as each end of a *constraint* is contributing to a new graph node.
When the graph nodes have been constructed, the edges from the points have to be divided between graph nodes. The edges are sorted by angle with respect to the point that they are connected to, and each graph node contains information about the visibility span that it has. Using this information, each edge will be connected to the right nodes, and the paths will not be able to cross the *constraints* (see Figure 3.1b).

3.4 Navigation Mesh

The implementation of the *Delaunay Triangulation* is based on the pseudo-code in the Computational Geometry book [6]. In order to use the triangulation as a navigation mesh, the *SymEdge* structure is used to represent the edges of the

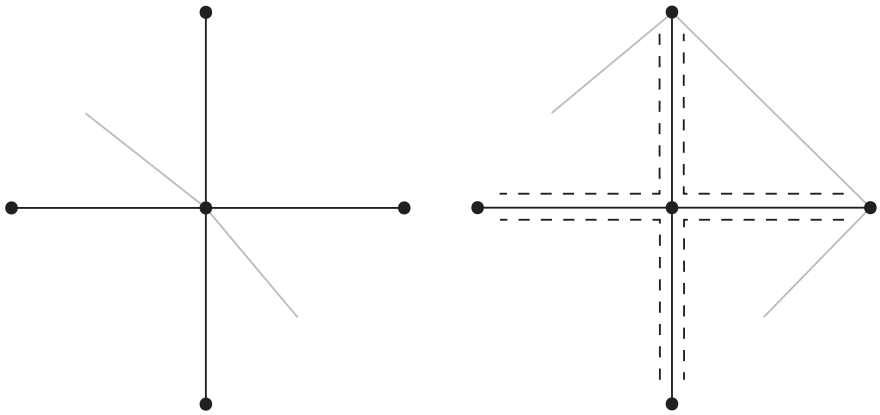


Figure 3.1: Two different paths (gray lines) found in the original visibility graph (a), and on the computed connectivity graph (b). The dashed lines outline the visibility span for each of the four graph nodes created from the center point.

triangles. The edge flips that the implementation of the *Delaunay Triangulation* relies on are also dependent on the *SymEdges*. When an edge e is flipped the algorithm needs to find the 2 triangles that share e, which is easy when all *SymEdges* store a pointer to the triangle they are a part of. After finding the two triangles, it is possible to get the points of the diagonal crossing e, and then flip e, such that it uses the points of the diagonal as endpoints.

Two different methods of handling insertions of constrained edges have been mentioned in the survey (Section 2.3.2). Neither of the two articles related to constraining edges [10][11] describe how to manage the *SymEdge* structure, while handling edges that cross the constrained edges. The first method [10] erases edges and retriangulates the *pseudo-polygon* using a recursive function. This made management of the data structure pretty complex and caused the implementation never to work properly. As an alternative, the second approach [11] was considered easier to implement, as it relies on the same edge flips as the *Delaunay Triangulation*, which was already implemented and known to work properly.

As for the point location, which is used both when building the mesh and afterwards when using the mesh for path finding, two different methods have been implemented. The first is the trivial one, which just loops through all the triangles in the triangulation, until it finds the triangle containing the point. The second one is based on *Sector Based Jump-and-Walk* method described in Section 2.3.4.

3.4.1 Data Structures

An edge in a triangulation is described by the *SymEdge* class as described previously (Section 2.3.2). A *SymEdge* is a *half-edge* spanned by two *Points*. It has a number of pointers to other *SymEdges*, which makes it possible to get from one edge to any other edge in the graph. It is possible to mark a *SymEdge* to be constrained, such that that it will not be flipped while triangulating. Making an edge as constrained also tells the graph search algorithm that any path intersecting the *constraint* will be invalid.

A *Triangle* is constructed from 3 *SymEdges*. The *Triangle* class does not contain any information about which other *Triangles* it is connected to. This is handled by the individual *SymEdges*, which have a pointer to the *Triangle* incident to them. To find a neighboring *Triangle*, t_2, from t_1, the twin edge of one the *SymEdges* in t_1 is found, and its incident *Triangle* will be t_2.

3.5 A*

To compare the different representations in terms of path finding, an A^* algorithm has been implemented. It uses a *binary heap*, as a queue, to keep track of the nodes to be evaluated. The heap performs all operations in worst-case $O(\log n)$ time, where n is the number of nodes inserted in the heap. In order to get the optimal worst-case theoretical bound on the graph search algorithm, we would have to use a *Fibonacci heap*, which has a decrease key operation that runs in a amortized constant time.
The nodes used in the algorithm are all inherited from a class, called *Node*, which contains information used while finding the path. The information are the g, h and f values (see Section 2.4.2, and a state determining if the node has been evaluated or not.

As described in Section 2.4.3 we have to use a *funnel algorithm* to get the path. The implementation uses a STL deque to represent the *funnel*, and left-of / right-of calculations to determine the visibility span of the points in the *funnel*.

3.5.1 Data Structures

Each of the representations has their own type of node used for the algorithm. The nodes used for the grid representation are called *GridNodes*, and they have information about where in the grid they are located, by two integers x and y. They also know if the cell that they represent is traversable or not, and they have a pointer to the grid array such that they can locate the neighboring cells.

The nodes for the *visibility graph* store the location of the node by using two *Reals*. The visibility span of the node is also stored by using two *Reals*. The first is used to store the angle of the start of the span, and second is used to store the angle of the end of the span. As the nodes can have up to n neighbors, the pointers to these are stored in a vector.

The *navigation mesh* nodes also stores their position using two *Reals*. They have a pointer to the triangle that they represent, and have a pointer to the edge that the algorithm crossed to enter the triangle. The number of neighbors can be at most three, as there can be only one neighbor for each side of the triangle. When the triangle has constrained edges, the adjacent triangles separated by these edges will not be counted as neighbors.

3.6 Memory Consumption

In this section, we will calculate the memory usage of each of the data structures used. This is done in order to compare the different representations in terms of memory consumption.

All the classes have a constructor and a number of member functions. This means that they each have a pointer to a *Virtual Method Table* (VMT), adding 4 bytes to the size of the objects.

3.6.1 Real

The size of a *Real* is 12 bytes (8 bytes for the double).

3.6.2 Point

The raw size of a point is 12 bytes (4 bytes for each of the pointers to the *Reals*), but it requires two *Real* objects, which are not shared with other objects. In total, this gives us a size of 36 bytes.

3.6.3 Line

A *Line* has two pointers to points, one for each end point. This gives a size of 12 bytes.

3.6.4 SymEdge

The size of a *SymEdge* is 36 bytes (8 pointers and 1 boolean value).

3.6.5 Triangle

The size of a *Triangle* is 16 bytes (3 pointers to *SymEdges*). If the *Sector Based Jump-and-Walk* method is used for point location the size is 20 bytes. The 4 additional bytes are for the integer used to store an id, to determine if the *Triangle* has been visited or not.

3.6.6 Node

It has 3 *Reals* (48 bytes), one unsigned integer (4 bytes) and one char (1 byte). The total size is 57 bytes.

3.6.7 Grid Node

A grid node has 1 char, 3 integers and 2 pointers. Counting the size of inherited variables from *Node*, the total size is 82 bytes

3.6.8 Visibility Graph Node

It has 4 *Reals*, 1 pointer and a STL vector. The total size of the node is $(145 + 4 \cdot \text{neighbor count})$ bytes. This includes the inherited variables.

3.6.9 Triangle Node

The size of a triangle node is 105 bytes (2 *Reals*, 3 pointers and inherited variables).

3.6.10 Navigation Mesh

The total size of triangulation will be $36 \cdot |V| + 36 \cdot 2 \cdot |E| + 20 \cdot |F|$ bytes, where $|V|$ is the number of vertices, $|E|$ is the number of edges, and $|F|$ is the number of faces. The reason for the number of edges being multiplied with 2 is that we have two *SymEdges* for each edge in the triangulation. Using Euler's formula $|V| - |E| + |F| = 2$, we can calculate the size just based on the number of points and triangles in the triangulation. We first rewrite the formula to calculate the number of edges $|E| = |V| + |F| - 2$. Then we insert the expression into the size equation: $36 \cdot |V| + 36 \cdot 2 \cdot (|V| + |F| - 2) + 20 \cdot |F|$. Reducing the equation gives us $108 \cdot |V| + 92 \cdot |F| - 144$ bytes.

3.6.11 Visibility Graph

The size of a visibility graph is $145 \cdot |V| + 4 \cdot |E|$ bytes, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

3.6.12 Grid Representation

The total size of a grid is $4 \cdot r^2 + 82 \cdot |GN|$ bytes, where $|GN|$ is the number of created grid nodes.

3.7 Debugging

To ensure correctness of the code a number of asserts are used. The asserts will be enabled when the program is compiled in debugging mode, and helps discovering unintended behavior. When an assertion is triggered, a file is saved, containing the inserted *constraints* . The file can be loaded later to help identifying the problem while debugging.

Another way to ensure correctness has been to step through the code, using the build-in debugger in Visual Studio. This helped making sure that the flow of the code and the variables was correct.

The implementation enables the user to visualize representations being built, while this is a nice feature; it also works as an excellent tool for visual debugging.

Chapter 4

Experiments

This chapter covers the experiments that were made to compare the different representations. First, the different types of environments will be introduced and then the results from the experiments will be presented and discussed shortly. An overall discussion will be available in Chapter 5.

Because the implementation does not use exact arithmetic, some of the representations might have flaws. This usually happens when constraints lie close to each other. In order to avoid experimenting on these flawed representations, the *navigation meshes* were build first as they are most prone to have flaws. If it constrained flaws, the test was skipped and another environment was used.

4.1 Setup

The experiments were done using a PC with an Intel Core 2 Duo E8400 CPU clocked at 3.16 GHz and 4 GB PC3200 DDR2 RAM running Windows 7 32 bit.

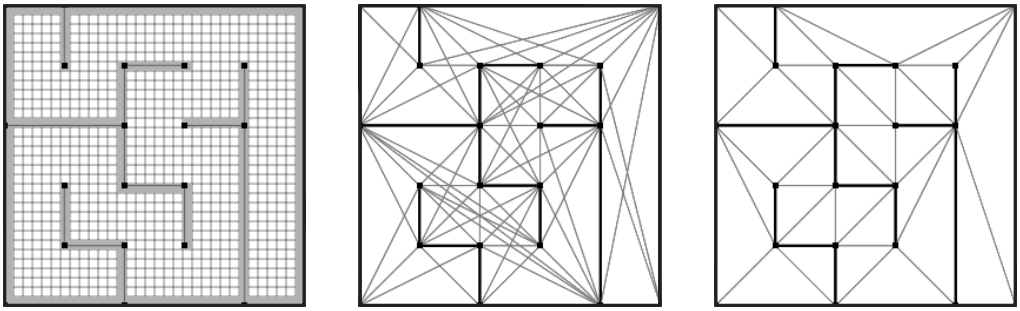
4.2 Environments

Three different types of environments were used for the experiments. A short description of each of them follows in the subsections below.

4.2.1 Maze

The mazes are axis-aligned and are randomly generated using a *Depth-First Search* (DFS) algorithm. The algorithm uses a grid to keep track of the cells in the maze and the walls between them. The first step is to locate a random cell c in the grid. Then a random neighbor n, which has all walls intact, is selected and the wall between the two cells are removed. The position of c is now pushed to a queue q, and n is assigned to c. The algorithm progresses

evaluating the cells until all cells have been visited. If at some point there are no more neighbors which have all walls intact, the next c is determined by popping a position from q. After the routine is finished, we have a perfect maze where all cells are connected, and there is a unique path between every pair of cells in the maze. Figure 4.1 shows the different representations of a maze.

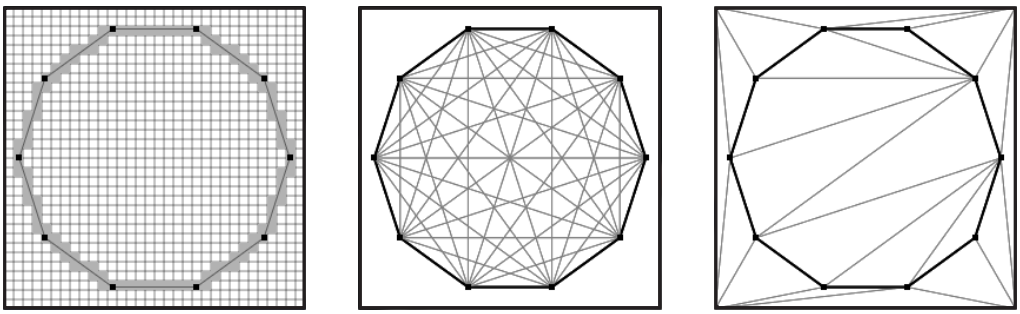


(a) (b) (c)

Figure 4.1: (a) *Grid representation* of the maze. (b) *Visibility graph* based on the maze. (c) *Navigation mesh* based on the maze.

4.2.2 Convex Polygon

To produce a worst-case scenario for the *visibility graph*, we generate a convex polygon. Because every point is visible from every other point in the polygon, we get $O(n^2)$ edges in the *visibility graph*. The *navigation mesh* will produce $O(n)$ triangles and the *grid representation* will not be affected by the specific placement of the *constraints*. Figure 4.2 shows the different representations with a polygon inserted.

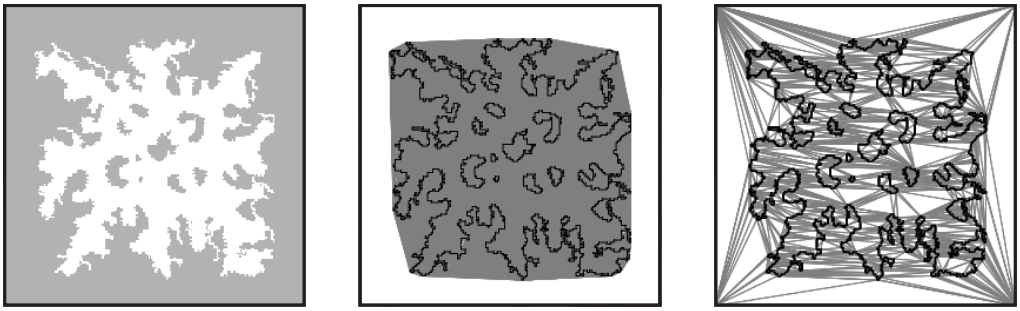


(a) (b) (c)

Figure 4.2: (a) *Grid representation* of the polygon. (b) *Visibility graph* based on the polygon. (c) *Navigation mesh* based on the polygon.

4.2.3 Game Maps

In order to tests the representations on "real" data, some experiments was done using maps from WarCraft III, which is a RTS game. The map environment is described by a 512×512 grid, where each cell has a value describing the area on the map that is covered by that cell. The *grid representation* will just use the raw data, while the other two representations have to use a number of *constraints*, which are created based on the map data. Figure 4.3 shows the different representations of a game map.



(a) (b) (c)

Figure 4.3: (a) *Grid representation* of the game map. The white area is traversable. (b) *Visibility graph* based on the game map. The gray area is all the edges of the graph. (c) *Navigation mesh* based on the game map.

4.3 Results

All the experiments either measure the time it takes for an operation, a node or edge count or a ratio. The aim of all the experiments is to get as low values as possible. In the experiments below the variable n denotes the number of vertices inserted in the representations.

4.3.1 Insertion of a Point

The experiment were done using a 100×100 maze with 6938 *constraints* and the same amount of points. The measurements were done for every 100 inserted point. For each measurement, 100 random points were generated and inserted, and the average time of the insertions was plotted. Figure 4.4 shows the measurements from the different representations plotted against each other. Figure 4.4a shows all the representations compared, while Figure 4.4b only shows the measurements from the grid and the *navigation mesh*. There are two different *navigation mesh* implementations: One where the trivial point location is used and another where the *Sector Based Jump-and-Walk* method is used. Also,

note that the unit difference on the y-axis on the two charts. The measurements of Figure 4.4a are in milliseconds, while they are in microseconds in Figure 4.4b.

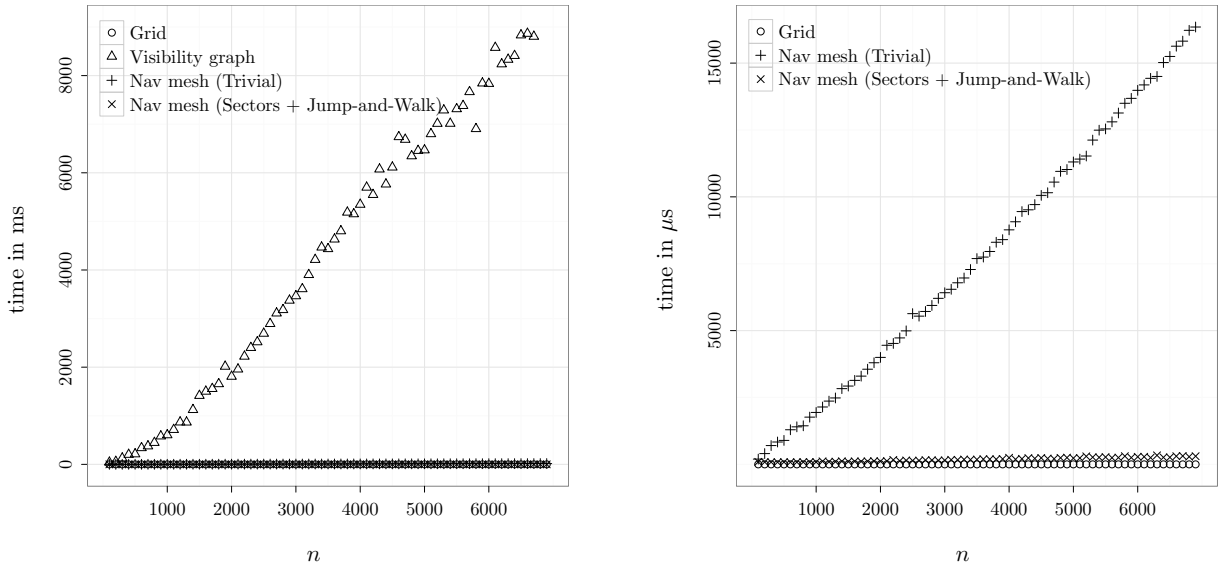


Figure 4.4: Insertion of points in a maze: $n = \{100, 200, \ldots, 6900\}$.

The measurements for the *visibility graph* show that time it takes to insert at point, increases constantly with the number of inserted points. This makes sense, as we always have to check for intersections with all *constraints*, but the number of checks depends on how the number of points in the graph at the time the new point is inserted.

The times spend for insertions in the *grid representation* are always the same. This is expected, as all that needs to be done is to locate the cell to update, and this can be done in constant time.

Insertions of points in the *navigation mesh* using the trivial approach, use more and more time as the number of inserted points increases. As we insert more points, the number of triangles in the mesh will increase, and it will take a longer time to look through them.

For the second approach where *sectors* and the *jump-and-walk* method are used, the measurements are quite different. At first, the time for inserting a point is decreasing, but after a while, it begins to increase (see Figure 4.5). The

reason for the decrease in time in the beginning is that the static data structure has to be updated. Every time a point is inserted, new triangles are created and the *sectors* have to be updated. Because of the edge flips, nearby triangles might need to be updated as well, and when a triangle is updated, the *sectors* have to be updated as well. When the first point is inserted, all of the *sectors* have to be updated, as the changed triangles will cover the whole of the *map* (see Figure A.6 in the appendix). As more points are inserted, and more triangles are created, the area that the changed triangles cover will be reduced. This means that fewer sectors have to be updated, and we spend less time inserting a point (see Figure A.7 in the appendix). When the time of the measurements increases, it is because the time spend on the updates are lower than the time it takes to locate the point.

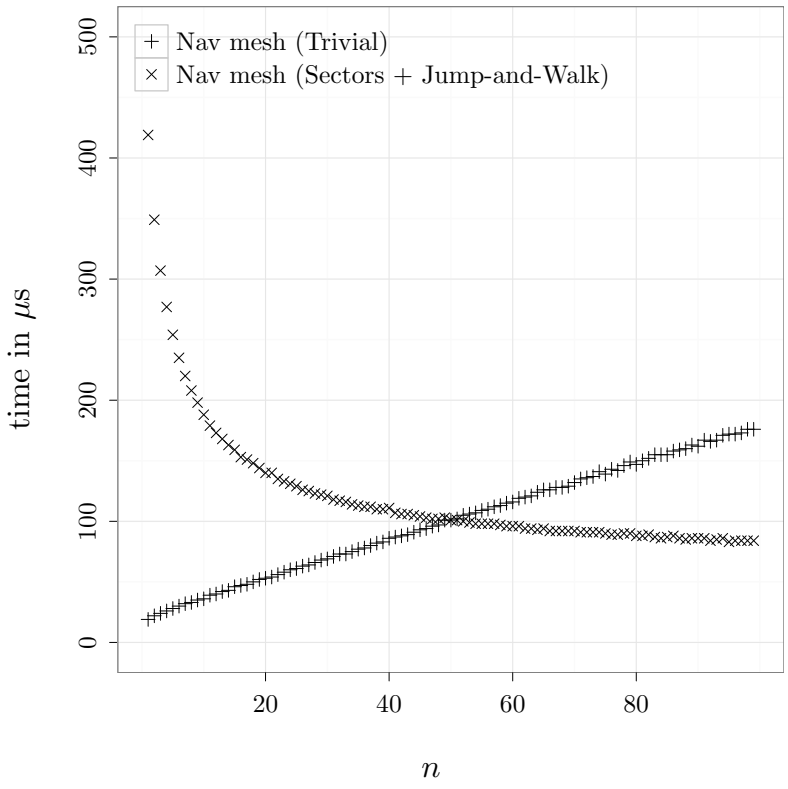


Figure 4.5: Insertion of points in a maze: $n = [0, 100[$.

4.3.2 Point Location

Only the *grid representation* and *navigation mesh* were included in this experiment. The *visibility graph* cannot represent areas, and we have to insert points into the graph in order to find the connections to the rest of the nodes.
The experiment was done by using the 100×100 maze from the previous experiment and performing point location of random points for every 100 inserted points.

Both of the point location methods for the *navigation mesh* have their times plotted in Figure 4.6a, which also have the times for the *grid representation*. In Figure 4.6b only has the times for the *Sector Based Jump-and-Walk* and the *grid representation*. The reason for having a second chart, is that the plotted values of the trivial point location method is so much higher, that it is hard to tell the other plots from each other.

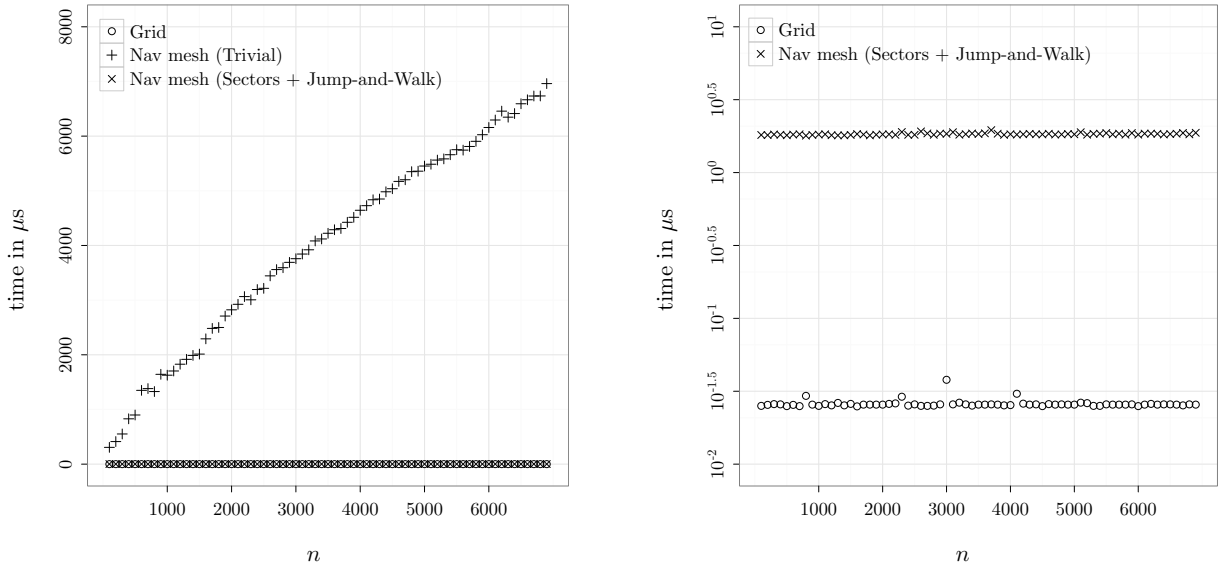


Figure 4.6: Point location: $n = \{100, 200, \ldots, 6900\}$.

The theoretical worst case running time for the trivial point location in the *navigation mesh* is $O(n)$. This is in accordance with the empirical results, which shows that the running time is increasing at an almost constant rate as n grows. A control chart where the time is divided by n can be seen in Figurefig:pointLocactionTest1 in the appendix. The curve of the plotted points in the control chart is not constant, but is negatively sloped, which means that

the actual running time is lower than the upper bound. This makes sense when we locate random points and the algorithm in most cases does not need to loop through all the triangles.

The *Sector Based Jump-and-Walk* method has a expected theoretical running time of $O(n/s)$, where $s = |\text{sectors}|$. The actual running times show, that the time it takes to locate a triangle is constant for all the measurements. The constant running time might be due to the oriented walk, which can be very efficient, especially if it is started in a triangle close to the query point.

As expected, the running time for point location in the *grid representation* is constant. It is a lot faster than the point location methods for the *navigation mesh*, as only a few operations is required to find the cell, in which the query point resides.

4.3.3 Construction Time

These experiments were done using mazes of resolution from 2×2 to 50×50. For each resolution, 10 different mazes were generated. Each test was performed 10 times, and the average times were used. The results are plotted in the two charts in Figure 4.7. Because the construction time for the *visibility graph* is so much higher than for the other representations, it is hard to tell the difference between them. Therefore, we use two different charts, one with and one without the *visibility graph*.

In the chart in Figure 4.7a, clearly shows the effect of the $O(n^3)$ algorithm for constructing the *visibility graph*. When the number of inserted points increases, the time grows cubically. To verify this, we can look at the control chart in the appendix (Figure A.3). The measurements in the control chart, were divided by the expected running time, and the graph converges to a constant as n increases. This confirms the construction time to be $O(n^3)$. The expected running time is $\frac{1}{2}n^2(n-1)$ because the implementation only check for *constraint* intersections once for each pair of points, instead of twice. This is done by giving each point an id. If we want to check a connection between two points p_1 and p_2 we only check for intersections if the id of p_1 is greater than the id of p_2. If we find no intersections between the line p_1p_2 and the *constraints*, we add two edges, one for p_1 and one for p_2, one in each direction. The total number of intersection checks now becomes an arithmetic series, $n-1+n-2+\ldots+1$, which multiplied with the number of *constraints* n, gives us the expected running time.

In the second chart in Figure 4.7b, we can see the difference between the other representations. The *grid representations* all start at specific offset, which is due to the time it takes to initialize the grid array. The additional time for the *grid representations* is due to the *constraints* being rasterized, and the construction of grid nodes. As the maze resolution increases, the average length of the *constraints* is reduced, but the number of *constraints* is increased, which

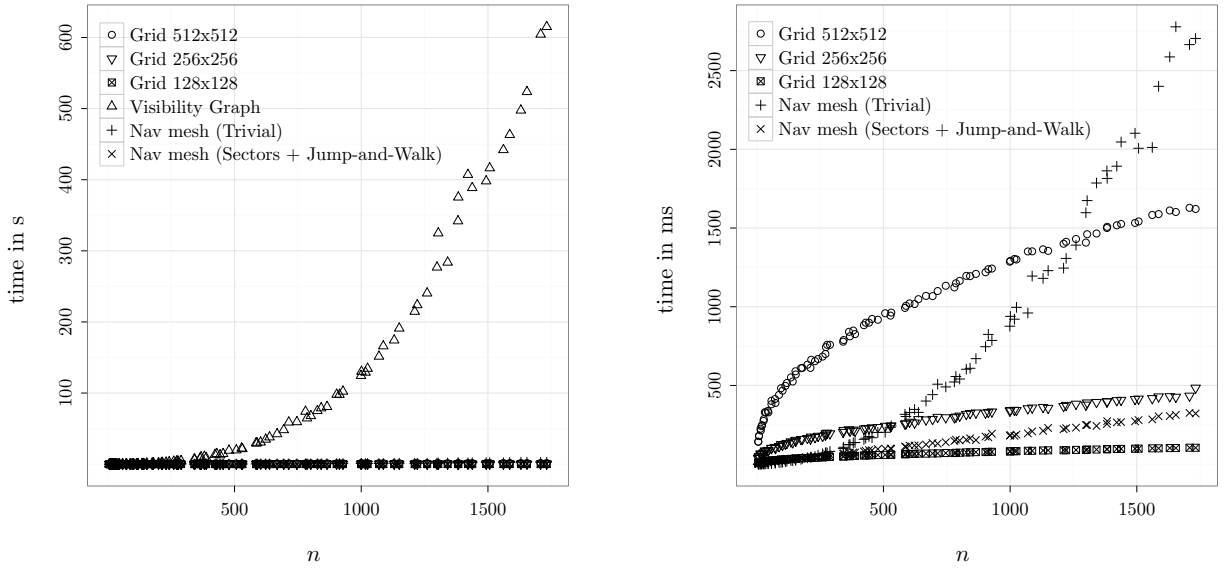


Figure 4.7: Construction time for the different representations.

means that more grid cells have to be constructed.

Two different times are recorded for the construction of the *navigation meshes*. The first uses the trivial point location approach, which results in a construction time of $O(n^2)$. This is in accordance with the plotted points where the time measurements are growing at a quadratic rate. Again, to test this, a control chart has been made. In Figure A.4 the measurements were divided by n^2, and the resulting graph converges to constant, which supports the claim.

When using the *sectors* and the *jump-and-walk* method, the expected running time is $O(n^2/sectors)$. The chart shows, that the actual time for the point location is neglectable, as the curve is linear. Once again a control chart was made, this time dividing the measurements with n. The chart can be seen in Figure A.5, and like the previous control chart, it converges to be constant.

4.3.4 Nodes

Like in the previous experiment, this one was done using the mazes of variable resolution. Again there are two charts, as the number of nodes in the representations are vastly different, which makes it difficult to compare them all in one chart.

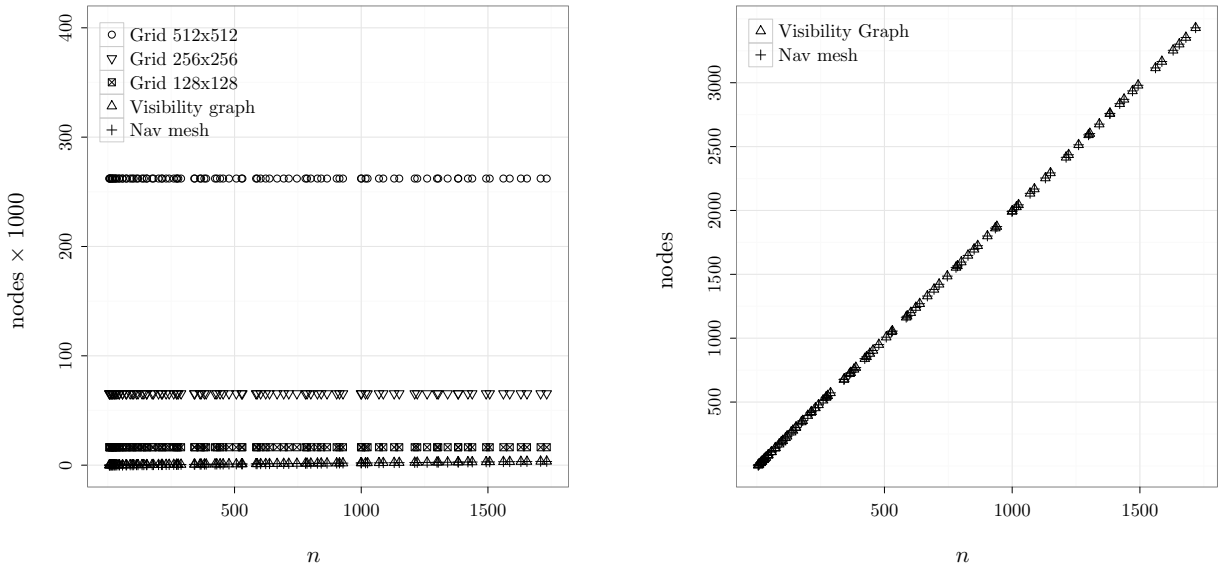


Figure 4.8: The number of nodes in the different representations based on mazes.

In Figure 4.8a, all the representations are shown. As described earlier, the node count in the *grid representations* only rely on the resolution of the grid, and is not affected by the number of inserted vertices. It is only the other two representation that are shown in Figure 4.8b, and both their node counts grow with a constant rate as the number of inserted vertices increases, as expected.

4.3.5 Edges

Both convex polygons and mazes were used in this experiment. Only the *visibility graph* and *navigation mesh* were tested in this experiment, as the number of edges in the *grid representation* is constant, and only rely on the resolution.

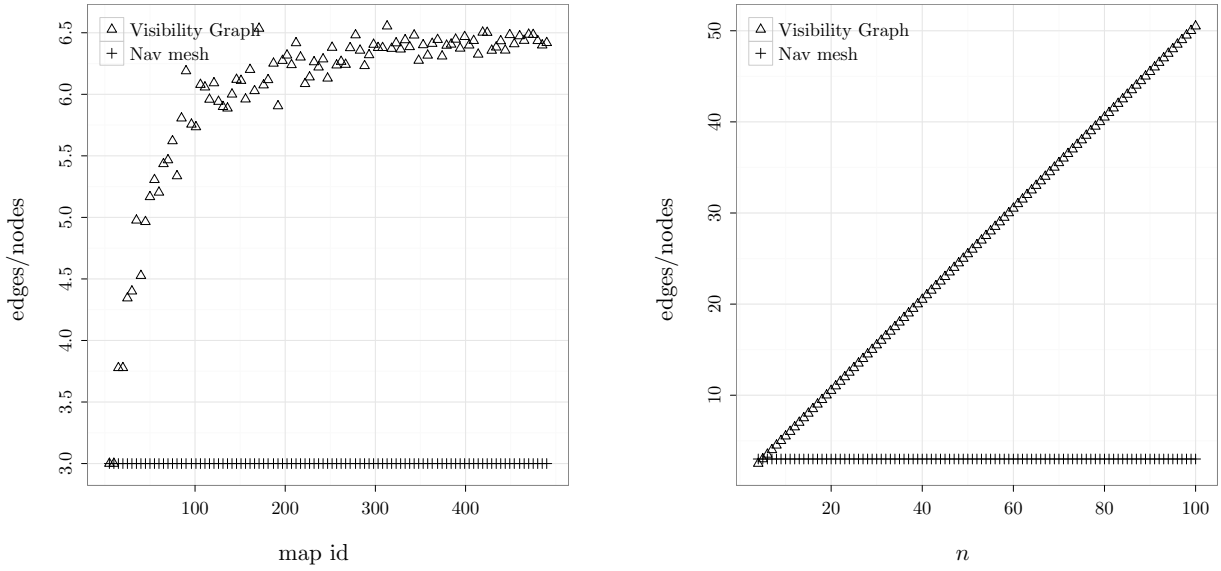


Figure 4.9: Number of edges per node. (a) Mazes. (b) Convex polygons.

In Figure 4.9a, we look at the number of edges per node in the representation for mazes. Not surprisingly the ratio for the *navigation mesh* is 3, as there are three edges for each triangle. For the *visibility graph*, the average number of visible neighbors per node is around 6.

The convex polygons are supposed to generate a worst case scenario for the *visibility graph*, resulting in $O(n^2)$ edges. In Figure 4.9b we see that this is in fact the case, as the number of edges per node is n. The number of edges in the *navigation mesh* is still 3.

4.3.6 Memory Consumption

Again, we used the mazes, and this time we calculate the expected memory consumption for the different representations based on the number of *constraints* inserted. The calculations use the sizes of the data structures from Section 3.6.

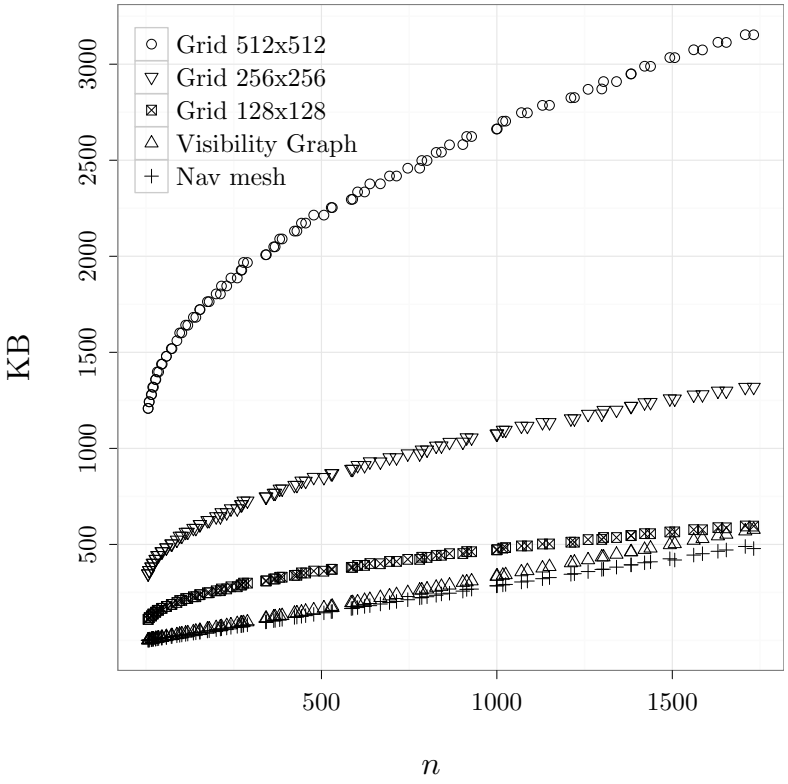


Figure 4.10: Memory consumption in KB.

Figure 4.10 has a chart with all memory sizes in KB. All the *grid representations* start at a fixed offset, which is due to sizes of the arrays that contain the grid nodes. As *constraints* are inserted, new nodes have to be created and more memory is used.

The amounts memory used by the *visibility graph* and *navigation mesh* are almost the same. The *navigation mesh* requires less space per triangle than the *visibility graph* spends per node. This makes the curve for the *visibility graph* a bit steeper than the curve for the *navigation mesh*.

4.3.7 Game Maps

A few experiments were made on the game maps, just to show the difference between the representations in a more realistic scenario. The number of nodes and edges were measured and plotted in the charts shown in Figure 4.11. One important difference between these experiments and the ones on the mazes, is that not all parts of the *map* are reachable. This means that a lot of nodes in the *grid representations* never will be accessed during the *path planning*. Because of this, these nodes were not included in the measurements.

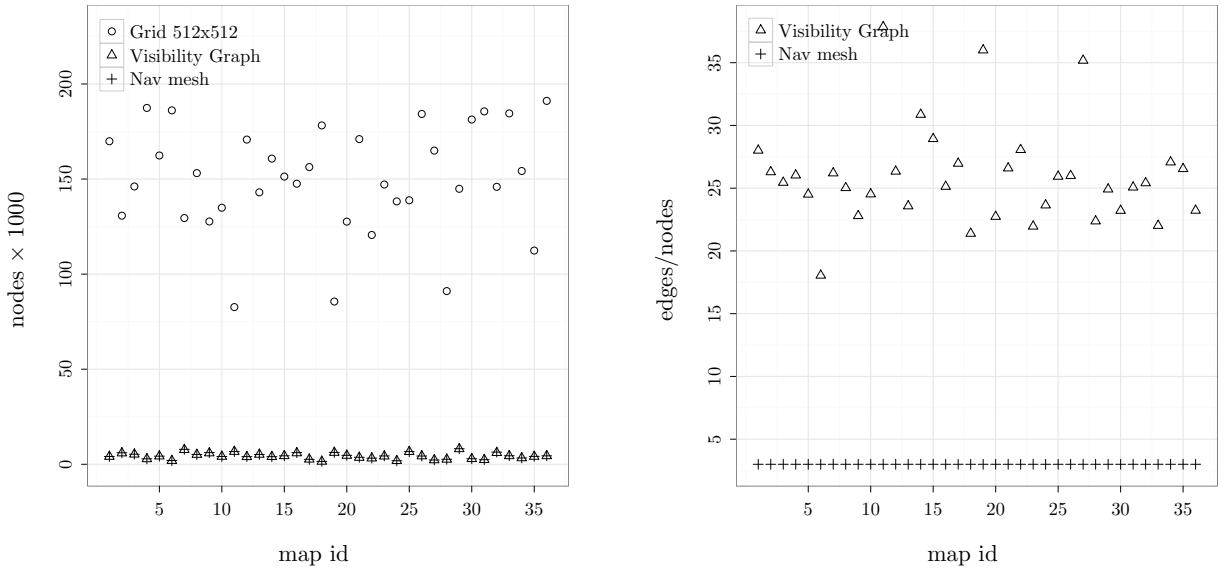


Figure 4.11: (a) The number of nodes in the different representations based on game maps. (b) Number of edges per node in game maps.

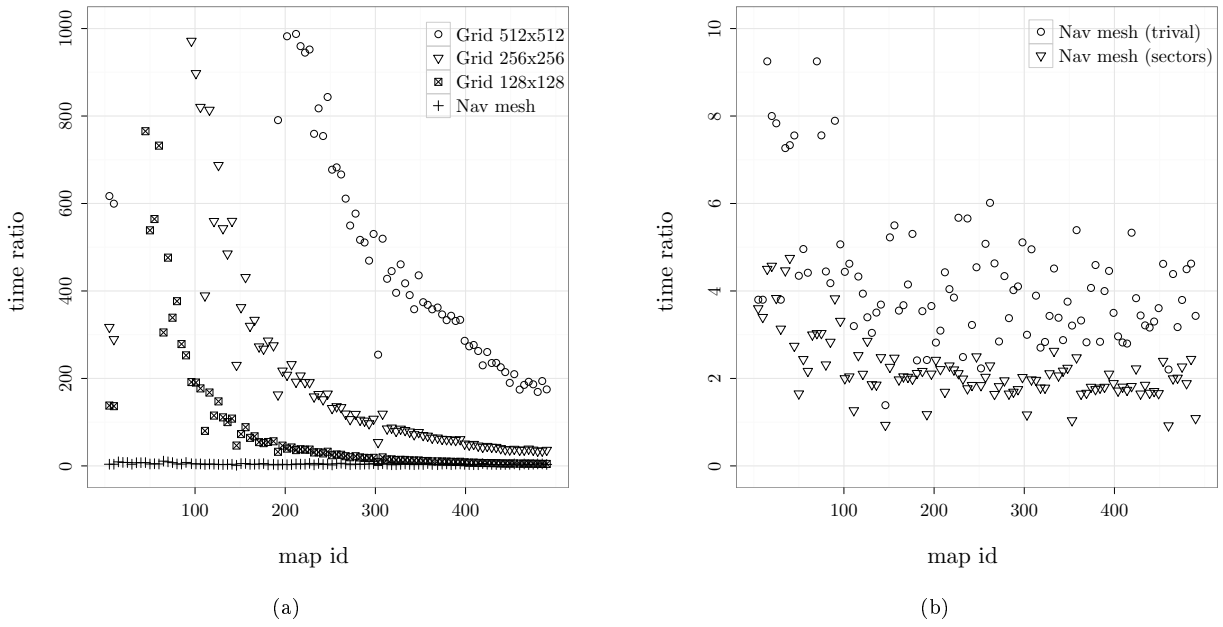
The number of nodes varies a lot in the different maps, as seen in Figure 4.11a, but the number of nodes in the *navigation mesh* and *visibility graph*, is only a fraction of the node count in the *grid representation*, regardless of the number of nodes. As mentioned in Section 1.1.3, it is because the grid representation relies on the extend of the *map*, whereas the other representations only rely on the *constraints*.

As for the number of edges in the representations, Figure 4.11b shows that the *visibility graph* has a around 25 edges per node. This is much higher than for the *visibility graphs* of the mazes, which is because the game maps are more open, and hence more nodes have a clear line of sight between them.

4.3.8 Path Time

In this experiment we measured the time spend for finding paths in the representations. Again, we used the mazes, and since they were randomly generated, we cannot compare the times directly. This is because even though they have the same sizes and the same amount of *obstacles*, the obstacles can be placed differently and can result different paths. Instead, we calculate a ratio between the optimal path time, and the time for finding the path the different representations. For the mazes, the optimal path time are gained by using the visibility graph.

There are two charts. The one in Figure 4.12a have both the *grid representations* and the *navigation mesh* plotted, while the other in Figure 4.12b only displays the plots for the *navigation mesh*. The reason for this is, like in the previous experiments, that the magnitude of the differences in the measurements is too great to get a proper understanding of how the representations perform against each other, if they are all plotted in one chart.



(a) (b)

Figure 4.12: Path time ratio for the different representations. The ratio calculated as: $r = \text{time}_{rep}/\text{time}_{vis}$.

The time ratios for the grids are greatly reduced as the resolution of the mazes increase. This not because we spend less for finding the paths in the grids, but rather that it takes longer time for finding the paths in the *visibility*

graph. The time for finding the paths in the grids are almost the same for all mazes regardless of size, because they have the same amount of nodes. The grids with higher resolution have a greater ratio as more nodes have to expanded, in order to find the path.

In Figure 4.12b, we see that the path time for the *navigation mesh* is lower when using sectors, which seems reasonable, as the point location step is needed in order to find the starting triangle.

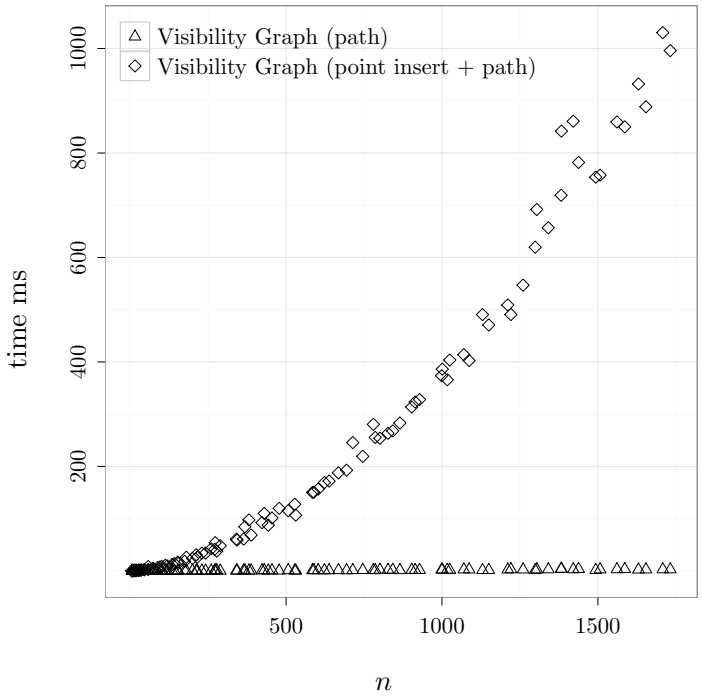


Figure 4.13: Time for path finding using the visibility graph, with and without the time spend for adding points.

We get the fastest path times using the *visibility graph*, but this is only if we do not count the time for inserting the start and goal points. In Figure 4.13, we see the plots of the measured times for finding paths in the *visibility graph*, with and without the time for inserting the points. As mentioned earlier in Section 2.2, the time spend inserting the points is much greater than the time of actually finding the path. The time for inserting a point is $O(n^2)$ in the trivial implementation, but even with a more efficient one, the insertion of the points will still take a lot of time compared to the time of finding the path.

4.3.9 Expanded Nodes

This experiment measures the number of nodes expanded during the path finding using the different representations. Like in the previous experiment, we calculate the ratio between the optimal value, which again is from the *visibility graph*, and the measured value from the representation. Figure 4.14 shows the plotted values, and we can see that there is a connection between this chart and the one in Figure 4.12a, showing the path times. This connection is due to the running times of the graph search algorithm are based on how many nodes it expands, and the number of edges it relaxes.

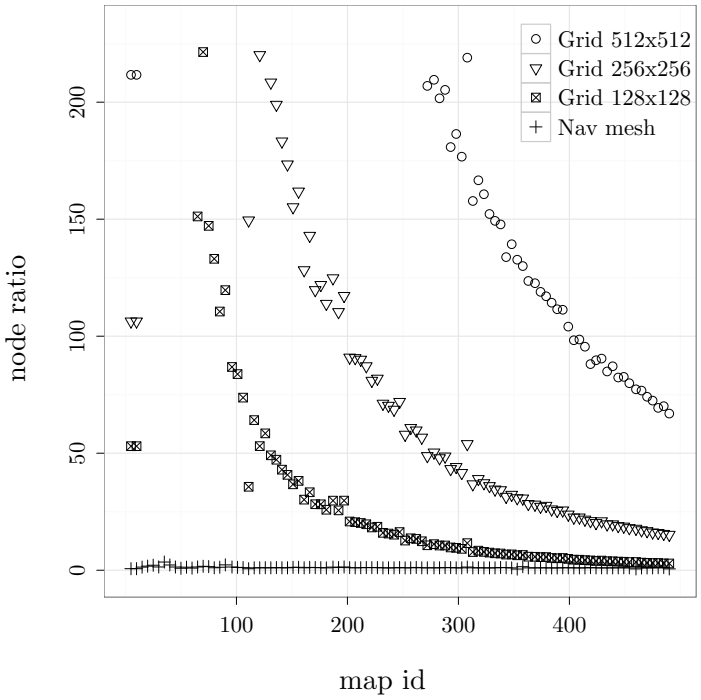


Figure 4.14: Node ratio for the different representations. The ratio calculated as: $r = \text{expNodes}_{rep}/\text{expNodes}_{vis}$.

4.3.10 Path Length

In this experiment, we will look at the lengths of the paths found in the different representations. Because both the *visibility graph* and the *navigation mesh* always find the optimal paths, they will not be a part of this experiment. This leaves the *grid representations*, and we will measure them using three different resolutions, as shown in Table 4.1. The values are length ratios that are calculated by dividing the length of the found paths with the optimal path length.

	10×10	20×20	30×30	40×40	50×50
512×512	104.85%	105.45%	106.26%	107.17%	108.33%
256×256	105.38%	107.75%	109.40%	110.85%	112.91%
128×128	107.28%	113.85%	115.78%	119.34%	124.52%

Table 4.1: Path length ratios for the different *grid representations*. The column headings are the resolutions of the mazes, and the row headings are the resolutions of the grids. The ratio calculated as: $r = \text{length}_{rep}/\text{length}_{optimal}$. The table shows the average ratios for the 10 random mazes of that resolution.

We can see that the higher the resolution of the grid is, the shorter path length we get. The higher the resolution of the grid is, the more accurately we can represent the *constraints*, and thereby get more accurate paths. In the table we can see that the larger the mazes become, the more inaccurate the paths are. This is because there will be more *constraints* in the larger mazes, and the more *constraints* we have to get around the more inaccurate the path becomes.

Chapter 5

Discussion

In this chapter, we will short have a discussion of the results from Chapter 4.

In almost all of the experiments where the *navigation mesh* was involved, it was the best or at least close to being the best of the compared. This was at least true when the *Sector Based Jump-and-Walk* point location method were used. We see that the point location method has a huge impact on the running time. The results for the point insertion (Section 4.3.1) and construction (Section 4.3.3) where the *Sector Based Jump-and-Walk* point location method was used show that it is much faster compared to the trivial point location method.

Paths found using the *visibility graph* or *navigation mesh* are always optimal, and the number of nodes expanded during the search compared to the *grid representation* is relatively low. The paths found in the *grid representation* are suboptimal due to the inaccurate representation of the *constraints* as seen in Section 4.3.10. It almost always takes longer to find a path in the *grid representation*, as the number of expanded nodes usually is higher (see Section 4.3.8).

The time it takes to find a path using the *visibility graph*, also involves inserting both the start and goal point into the graph. This takes a long time, and especially in our case where we use a trivial algorithm. The extra time spend makes the *visibility graph* a less viable representation for planar environments, compared to the other two representations (see Section 4.3.8).

The *grid representations* come out last in the experiment about memory consumption (Section 4.3.6), as they have a constant overhead due to the arrays. The *navigation mesh* has the lowest memory usage. The amount of memory used by the *visibility graph* is nearly as low, but difference will be more noticeable in a worst-case scenario, with $n - 1$ edges for each node.

Overall, the results show that the *navigation mesh* is the most viable representation for planar environments.

Chapter 6

Conclusion

Here we will summarize the findings and results in this thesis. In the sections below, there will be a short conclusion for each of the representations, and an overall conclusion.

6.1 Grid Representation

The *grid representation* is often used for planar environments, as it can represent the whole *map* by dividing it into cells. The grids rely on a predefined resolution, which determines how many cells are used to describe the environment. Because of the grid structure, where all the cells are aligned side by side, the representation can only approximate the *obstacles* in the environment. This will affect the accuracy of the paths found in the representation, and if the resolution is not high enough, valid paths might not be found using the grid.

Because the *grid representation* has a fixed number of cells, the memory usage will always have an overhead due to the array that contains the cells. The number of nodes that have to be explored during a path search will usually be higher than in the other representations, due to the fixed node sizes and the way they are connected.

One of the good things about the representation is the simplicity of it. It makes it easy to implement, and the access time to the nodes is constant.

6.2 Visibility Graph

The *visibility graphs* are not usually used for representing planar environments, and we have found out why that is. The necessity of inserting the query points into the graph before finding a path causes the total query times to be unacceptable for many applications, especially when used in complex environments.

This is confirmed by the experiments in Section 4.3.1 and 4.3.8.

The advantages of this representation are that it is simple to implement, and the paths found will always be optimal.

6.3 Navigation Mesh

There are a number of advantages by using a *navigation mesh* for representing an environment. Compared to the *grid representation*, the number of nodes will be reduced greatly, and as the mesh is based on the *obstacles* in the environment, we will always get optimal paths. Even when the *obstacles* are axis-aligned and the environment is designed to work with grids, the navigation mesh will have a fewer nodes, as it represents any area similarly regardless of size. We see this in the experiments with the game maps, where the number of nodes for the *navigation mesh* only is a fraction of that of the grid (see Section 4.3.7.
Because of the low node count, we are able to find the paths faster, and use less memory for the representation. This can be seen in the experiments in Section 4.3.8 and 4.3.6.

6.4 Overall

We can conclude that *navigation meshes* are the optimal choice of representation, for any planar environment that can be represented using *constraints*. They are not as simple to implement as the other representations, but the advantages of fast query times, reduced memory consumption and optimal paths, outweigh that.

Chapter 7

Extensions

In this chapter, we will look at some possible extensions, which could be interesting implement and experiment on.

7.1 Point Location

We saw in experiments (Chapter 4) that the point location method used by the *navigation mesh* had a huge impact on the performance. It could be interesting to implement the other point location methods, described in Section 2.3.4, to see how they perform in practice.

7.2 Nonpoint Objects

Throughout this thesis, we have only looked at algorithms and representations for point objects, with zero radiuses, to navigate through the environment. In order to support nonpoint objects, we could grow all the obstacles by the radius of the objects and build the representations. The problem is that we would have to make a new representation each time the radius of the object changes, or if an object with a different size is used. Demyan et al. [14] have modified both an A^* algorithm and a *funnel algorithm* to handle arbitrary widths of objects without changing the navigation mesh. It would be interesting to see how the performance is affected by these modifications.

7.3 Line of Sight

Many games use the line of sight between units or objects to determine if they should be rendered. Using the representations, it is relatively simple to implement this feature. For the *grid representation* and *navigation mesh*, you could just make an oriented walk between the two query points, and check that no constraints are intersected [3]. For the *visibility graph*, you would have to insert

the two points and check that they can see each other or just loop through all the constraints and check for an intersection.

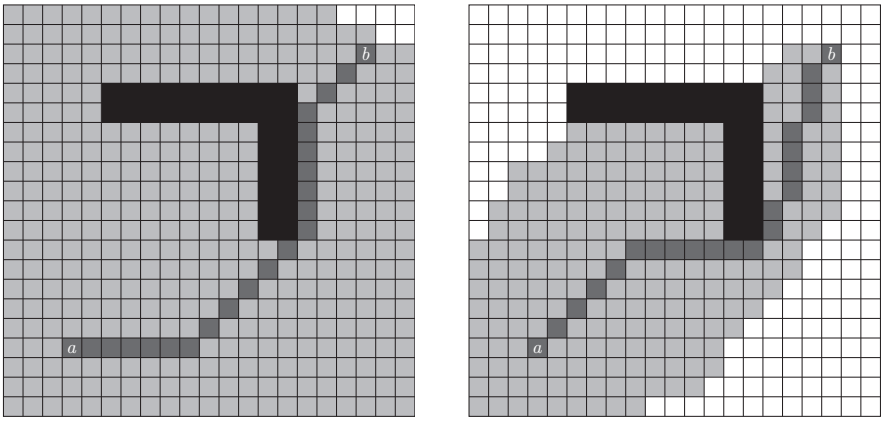
7.4 Dynamic Changes

If the environment is changing, we will have to update the constraints. The implementation of the *navigation mesh* does not currently support removal of constraints, but Kallmann et al. [8] describe how it can be done, such that the mesh is only modified locally. When a constraint is changed, we need to remove it from the mesh, and reinsert it again with updated end points. By applying this, it would possible to avoid collision between moving objects.

7.5 Terrain Traversal Cost

The implementation of the *grid representation* is the only one that currently supports traversal costs. In order to get the other representations to support it as well, we will have to introduce a new kind of *constraints* that is passable. These new *constraints* will encapsulate the areas with changed traversal cost. Each node or triangle will have to store a cost multiplier, which are used when the graph search algorithm calculates the paths.

Appendix A



(a) *Dijkstra's algorithm* (b) *A**

Figure A.1: Comparison of nodes expanded. The dark gray nodes denote the path, the light gray nodes are the nodes expanded, and the black nodes are outlining obstacles.

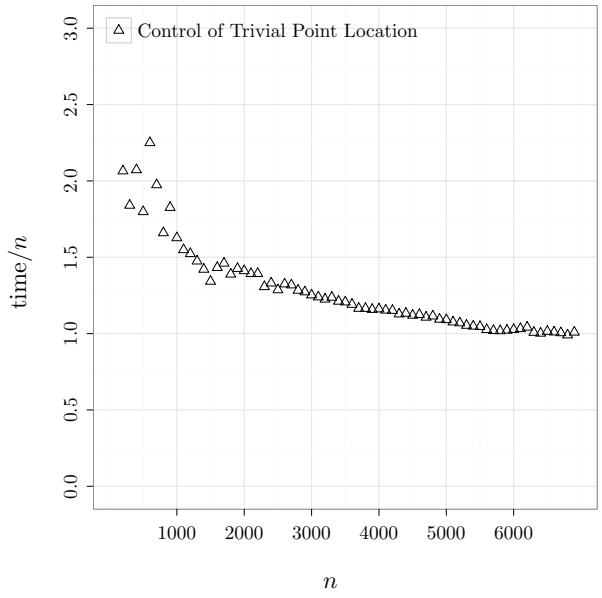


Figure A.2: Trivial point location time for the *navigation mesh* divided by n.

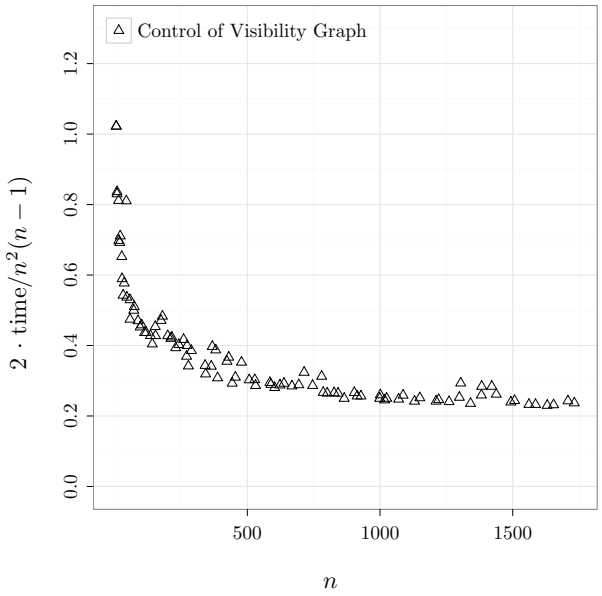


Figure A.3: Construction time for *visibility graphs* divided by $\frac{1}{2}n^2(n-1)$.

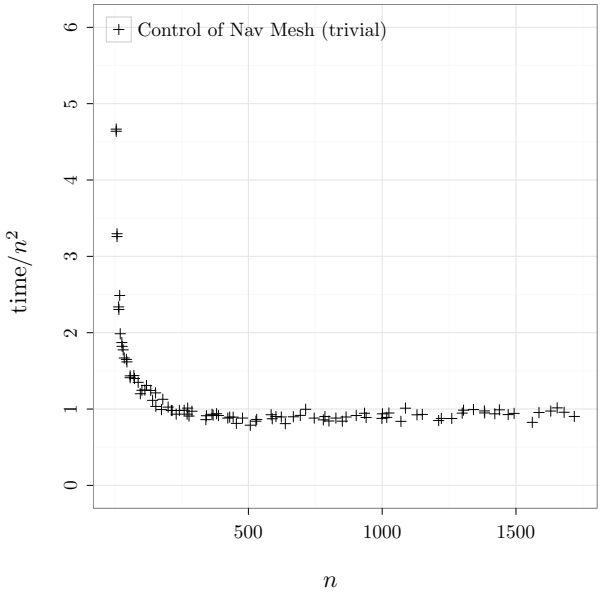


Figure A.4: Construction time for *navigation meshes* using trivial point location divided by n^2.

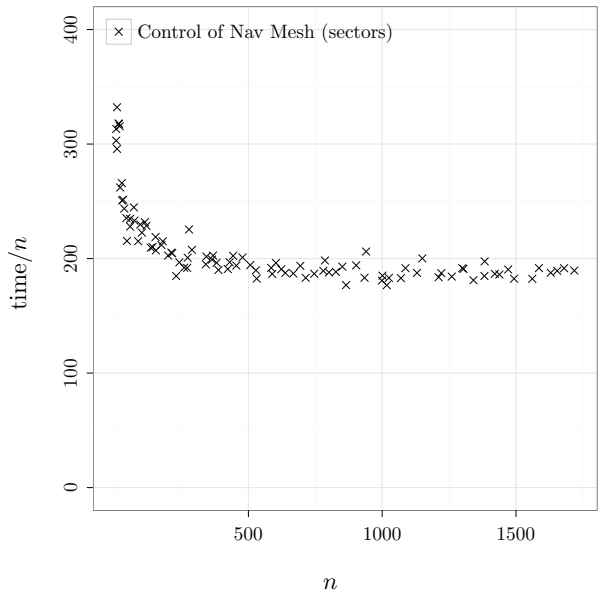
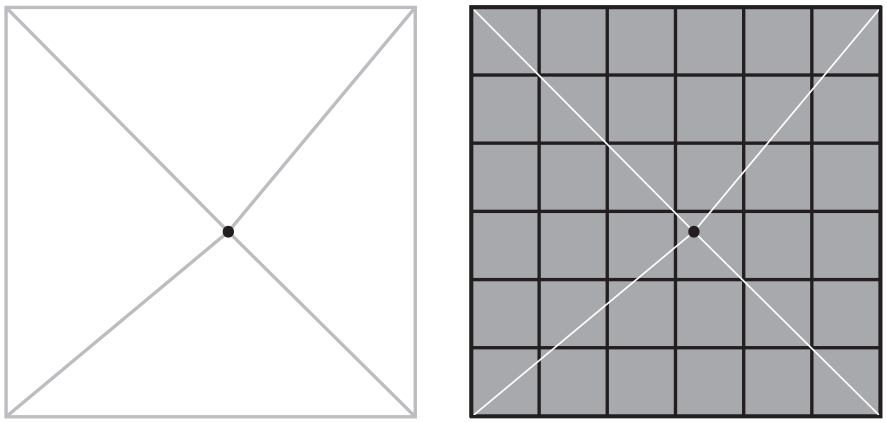
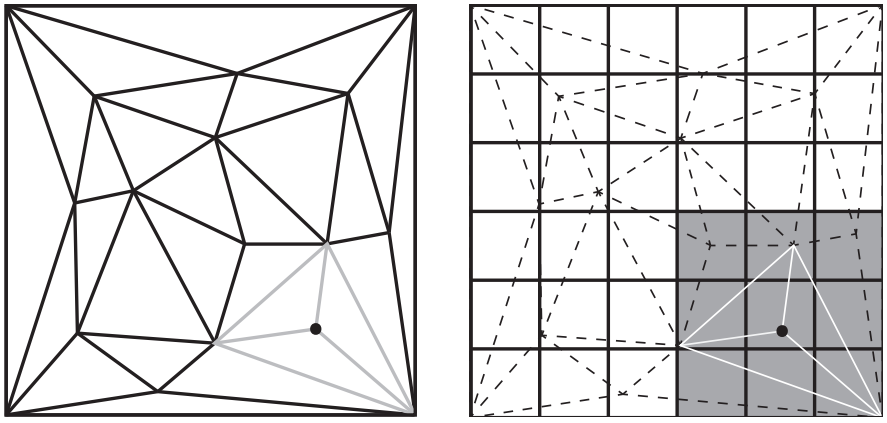


Figure A.5: Construction time for *navigation meshes* using *Sector Based Jump-and-Walk* divided by n.



(a) (b)

Figure A.6: Sector updates after the first point insertion. (a) The triangulation with the updated triangles marked in gray. (b) The sector grid where the sectors to be checked for updates are marked in gray.



(a) (b)

Figure A.7: Sector updates after the thirteenth point insertion. (a) The triangulation with the updated triangles marked in gray. (b) The sector grid where the sectors to be checked for updates are marked in gray.

Bibliography

[1] W. Schickler and A. Thorpe, "Surface Estimation Based on LIDAR **(Introduction)**," in *Proceedings of the ASPRS Annual Conference St*, 2001.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms* **(Section 24.3, Appendix A)**. McGraw-Hill Higher Education, 2nd ed., 2001.

[3] M. Kallmann, "Navigation queries from triangular meshes **(Introduction)**," vol. 6459 of *Lecture Notes in Computer Science*, pp. 230–241, Springer, 2010.

[4] P. Yap, "Grid-Based Path-Finding **(Section 4, 5, 7)**," in *Canadian Conference on AI* (R. Cohen and B. Spencer, eds.), vol. 2338 of *Lecture Notes in Computer Science*, pp. 44–55, Springer, 2002.

[5] Y. Bjornsson, M. Enzenberger, R. Holte, J. Schaejfer, and P. Yap, "Comparison of different grid abstractions for pathfinding on maps," in *Proceedings of the 18th international joint conference on Artificial intelligence*, pp. 1511–1512, Morgan Kaufmann Publishers Inc., 2003.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry Algorithms and Applications* **(Section 2.2, Chapter 15)**. Springer-Verlag, 3 ed., 2008.

[7] S. K. Ghosh and D. M. Mount, "An output sensitive algorithm for computing visibility graphs **(Introduction)**," in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, FOCS '87, pp. 11–19, IEEE Computer Society, 1987.

[8] M. Kallmann, H. Bieri, and D. Thalmann, "Fully Dynamic Constrained Delaunay Triangulations **(Section 4)**," in *Geometric Modelling for Scientific Visualization* (H. M. L. L. G. Brunnett, B. Hamann, ed.), pp. 241–257, Springer-Verlag, first ed., 2003. ISBN 3-540-40116-4.

[9] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams **(Section 3)**," in *ICALP* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*, pp. 414–431, Springer, 1990.

[10] M. V. Anglada, "An improved incremental algorithm for constructing restricted Delaunay triangulations **(Section 4)**," *Computers and Graphics*, vol. 21, no. 2, pp. 215–223, 1997.

[11] J. Bernal, "Inserting Line Segments into Triangulations and Tetrahedralizations **(Introduction)**," vol. 5596, National Institute of Standards and Technology, 1995.

[12] C. L. and Lawson, "Transforming Triangulations **(Introduction)**," *Discrete Mathematics*, vol. 3, no. 4, pp. 365–372, 1972.

[13] P. K. Agarwal, L. Arge, and K. Yi, "I/O-Efficient Construction of Constrained Delaunay Triangulations **(introduction)**," in *ESA* (G. S. Brodal and S. Leonardi, eds.), vol. 3669 of *Lecture Notes in Computer Science*, pp. 355–366, Springer, 2005.

[14] D. Demyen and M. Buro, "Efficient triangulation-based pathfinding," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pp. 942–947, AAAI Press, 2006.

[15] R. Seidel, "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons **(Section 3)**," *Comput. Geom.*, vol. 1, pp. 51–64, 1991.

[16] E. W. Dijkstra, "A Note on Two Problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. 10.1007/BF01386390.

[17] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths **(II.C, III.B)**," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[18] J. Hershberger and J. Snoeyink, "Computing minimum length paths of a given homotopy class **(Section 3)**," *Comput. Geom. Theory Appl.*, vol. 4, pp. 63–97, 1994.

[19] M. Kallmann, "Path Planning in Triangulations **(Sections 3-4)**," in *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005.

[20] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical pathfinding **(Section 1, 3.0, 3.1, 3.3, A.2.2)**," *Journal of Game Development*, vol. 1, pp. 7–28, 2004.

[21] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks **(Introduction)**," in *Experimental Algorithms* (C. McGeoch, ed.), vol. 5038 of *Lecture Notes in Computer Science*, pp. 319–333, Springer Berlin / Heidelberg, 2008.

[22] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm **(Section 1.1 Hierarchical Approaches)**," *J. Exp. Algorithmics*, vol. 15, pp. 2.3:2.1–2.3:2.31, 2010.

[23] D. Demyen, "Efficient Triangulation-Based Pathfinding **(Chapter 6)**," Master's thesis, University of Alberta, Edmonton, Alberta, 2006.

Index