Improving spatial structures for realtime pathfinding

Gerard Meier
Amsterdam University of Applied Sciences
gerjo.meier@hva.nl

February 24th, 2013

Abstract

There are many complete pathfinding algorithms available. These algorithms are supplemented by data structures representing the virtual world. Data structures are distinguished by either being precomputed or realtime generated. Much has been written about the former, however, little has been written about realtime data structures. Real time data structures are highly suited for procedurally generated worlds or worlds occupied by dynamic entities.

In this paper I propose a general purpose realtime spatial partitioning technique which is pathfinding aware. I demonstrate by empirical means that the performance of these structures can be on par with the set requirements.

1 Introduction

Guerrilla Tactics is an applied research project in which techniques are explored for procedurally generating worlds for computer games. Before starting the actual world generation research, we first developed PhantomCPP, a C++ game engine which could harness the dynamics of our to be generated world. One of the problems to be solved was pathfinding through large open worlds. Many papers have been written with regards to large world pathfinding, including breaking the world in manageable sections [7], manually placed waypoints and precomputed navigation meshes. While each technique has its merits, they are all optimisations for specific scenarios - we did not have a world at the time, so a general purpose large world pathfinding technique had to be discovered.

This paper is structured as follows. In section 2 I discuss several existing and commonly employed techniques. In section 3 I introduce our proposed technique using A* and dynamically growing spatial trees. Section 4 demonstrates our implementation. I conclude in section 5.

1.1 Benchmark world

While a rough draft of the game design document for Guerrilla Tactics was created, many unknown factors were still present. We were unsure if there would be a large open world or a world composed out of many smaller areas. Areas could be generated just-in-time or seconds before the game starts. Using agile methodologies such problems would be tackled down the line. With so many unknown variables, we created a test world which would serve as a benchmark for PhantomCPP. This initial test world measured 40.000 pixels squared and contained both manually and, randomly placed trees and enemy tanks. The soldiers controlled by the player should be able to walk through other friendly soldiers, but they should not be able to walk through tanks and trees. Image 1 shows the test world with soldiers, tanks and trees.

Figure 1: Guerrilla Tactic's initial forrest-like test world, having both open areas and narrow corridors. Tanks are white, soldiers are red and trees are green.

2 Existing techniques

In this section I discuss existing techniques that were evaluated for Guerrilla Tactics pathfinding. Most techniques in this section are just variations of representing spatial data of a game world, A* is then used to prune these structures in an effort

to find the quickest path.

2.1 Grids

A technique commonly used in games (The Sims, Age of Empire, Baldurs Gate), the world is decomposed into a uniform grid of buckets, each entity is associated with the bucket(s) it touches. Using spatial hashing techniques, insertion in the correct bucket(s) is done in constant time. Yap [1] discusses grids in detail. A* is then used to prune all buckets in search for the shorted route.

A remarkable caveat with grids is its handling of open areas. If an area contains no entities, buckets are still created for those empty areas. Thus, the eventual algorithm will pointlessly traverse empty areas. To counter this problem, buckets could be made larger - but this at a cost of reducing the quality of the generated path. Though of lesser concern with modern day computers, decomposing a large world in a high resolution grid may impose high memory requirements.

Path symmetry Grids also introduce a high level of path symmetry, where multiple unique routes of the same length share the same start and end coordinate (image 2). When multiple potential routes are equally valid, A* will explode them all. In addition to that, varying per A* heuristic, when the start position changes slightly, a whole new route might be returned by A*.

Rectangular Symmetry Reduction (RSR) has been proposed by Harabor, Botea and Kilby [2] to solve this issue. Through a pre-process phase adjacent buckets are merged, subsequently, paths are searched from bordering buckets, rather than from the center. The pre-process state makes RSR unsuitable for Guerrilla Tactics' dynamic world.

Jump Point search (JPS) is a variant of A*, which addresses path symmetry [3]. Unlike A* which explores the node with the lowest "f"-score from the "open"-collection first, JPS explores the lowest scoring neighbour-node from the recent most explored node, first. In short, JPS will keep exploring in the same direction. JPS is both complete and optimal, however, it assumes a uniform cost for each node. At the time, it would not be unthinkable for Guerrilla Tactics to have different costs for terrain, hard rock should be preferred over pools of mud, and mine fields should only be negotiated as a last resource. As JPS is not a data structure, it could be combined with RSR or other spatial representations.

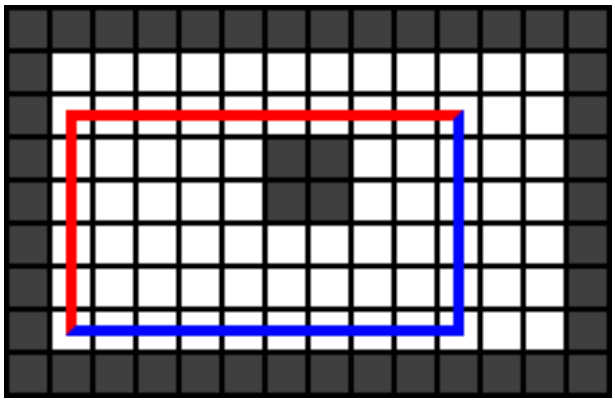

Figure 2: Path symmetry, both routes (blue and red) are equally valid. This potentially causes A to explore both directions, causing more overhead. Depending on how deterministic the A* heuristic is, a different path may be generated each time - heuristics without a tiebreaker are especially susceptible.*

2.2 Navigation meshes

Through convex polygons, navigation meshes describe areas which are traversable by an entity [6]. Convex polygons are required to ensure an unobtrusive path exists between the edges of a polygon. Many games already employ this for AI-bot movement: Uncharted: Drake's Fortune, Left 4 Dead, Halo 3 etc. Nativation meshes solve the issue of empty buckets in a grid, and solve the granularity problem demonstrated in image 3, only 4 polygons would be required to describe the walkable area.

Navigation meshes can be generated procedurally [5] [6], or by the game designer via a level editor. Manually generating meshes is not applicable with procedurally generated game worlds. Procedurally generating meshes requires a process-phase in which the world is decomposed, the later being inefficient for just-in-time generated worlds. In addition to that, unless the mesh is generated/adapted continuously, they require local object avoidance to supplement the pathfinding efforts [4].

Deriving a path from meshes Having the world decomposed into polygons alone does not solve the pathfinding issue, we still need to derive waypoints from the meshes. In its simplest form, the entity could walk from the center of each polygon to the next adjacent center. The later giving an unnatural result. Several techniques have been proposed, most notably;

- *Path smoothing* Constructing a path by connecting all centres and smoothing e.g., using Bezier curves.

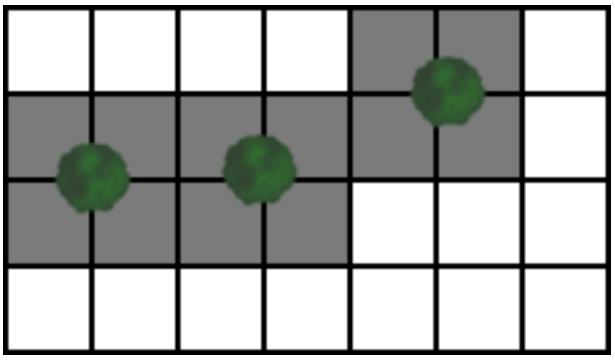

Figure 3: Granularity problem, as the tile size increases, huge chunks of a world become inaccessible. This image shows how 3 awkwardly placed trees can occupy 12 tiles (marked grey).

- *Path optimization* Constructing a path by connecting all centres and removing unnecessary waypoints by line of sight calculations [4] [7].
- *Reactive path following* Constructing a path by connecting all centres. The entity will continuously look ahead, and walk towards the intersection of the path and his range of view [4]. This works particular well with other flocking behaviours such as squad formation movements[8].

Each method boosts its own merits, it would not be unthinkable to combine several methods. Do note that these techniques can apply to grid based representations, too.

2.3 Space partitioning trees

Space partitioning trees work by decomposing an area in subareas until said area is either empty or a limit is reached (e.g., there is little reason in subdividing 1 physical pixel). In the classical sense either binarytrees or quadtrees[9] are used for this.

Quad and binary space partitioning trees can solve the granularity issue of grids, individual spaces adapt their resolution to the local environment. Unlike grids, in average case, trees contain a lesser amount of adjacent empty areas (grid buckets), this however at the cost of no longer allowing A* to generate an optimal path (image 4).

Partition techniques There are many partitioning techniques available, Samet has described most of them in his book[12]. With highly dynamic worlds, the tree must be reconstructed frequently, as such, a cheap insertion algorithm must be used. The cheapest method is subdividing the areas in 2 or 4 equal parts, and inserting the entities at the root node, the time complexity characteristics will be similar to that of a binary search algorithm. Other methods create a more balanced tree, but this at a computational cost. Expensive insertion methods are very sensible for static worlds or real live scenarios (e.g., a highway would rarely change its geographical position).

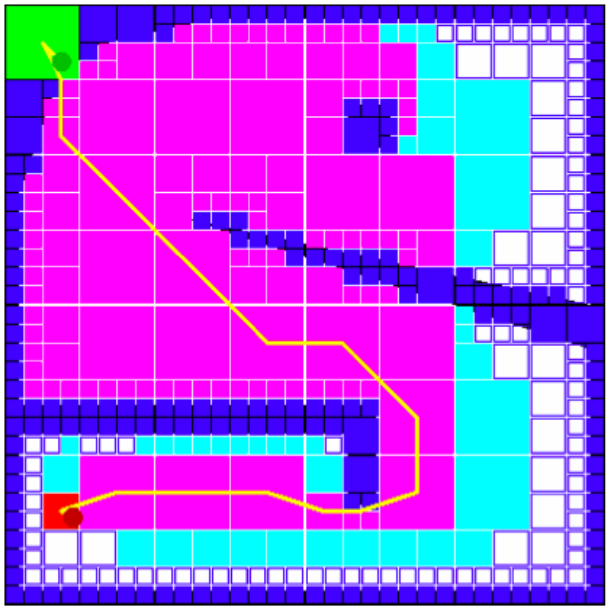

Figure 4: A world partitioned using a quad tree. Blue areas are inaccessible by the entity, the purple areas are explored by A leading to the creation of the yellow path[11].*

Framed-quadtrees Similar to navigation meshes, a path must still be derived from the remaining areas. Alternatively to the path smoothing techniques described in section 2.3, Yahja, et al propose the use of framed-quadtrees[10]. The space is decomposed as usual, however the empty leaf nodes are surrounded by smaller tiles. The pathfinding algorithm will search paths from these smaller tiles, to other smaller tiles. This approach is similar to the aforementioned RSR[2], only the means of generating tiles varies.

2.4 Hierarchical techniques

The use of Hierarchical Pathfinding A* (HPA*) has been proposed by Botea et al[7]. HPA* decomposes the world into manageable subsections. These subsections can still contain obstacles, this unlike the aforementioned RSR[2] and framed-quadtrees[10]. The subsections are then processed into a graph by connecting adjacent subsections with an edge. Knowing the intersections, a rough path could easily be calculated, subsequently only the subsections encountered by the path have to

have thoroughly examined in order to derive a path. The proposed technique highly depends on precomputing optimally-shaped subsections, making it less suitable for just-in-time generated worlds.

Hierarchical A* (HA*) has been introduced by Holte et al[13]. Eventhough it uses a form of hierarchy, it is not to be confused with HPA*. HA* uses a multilevel representation in order to calculate a heuristic. In its simplest terms; a heuristic evaluation will query a higher level abstraction for heuristical estimates, this process repeats until the highest level of abstraction is reached. Image 5 shows an example how this could be used in Guerrilla Tactics. Using grids, a step in "level 1" would query "level 2", which in turn queries "level 3" for an heuristic estimate. The highest level would increase the heuristic score when a tree is found, suggesting that A* should explore other areas, first. HA* requires a lot of tuning before it calculates optimal paths, depending on the type of world, different representation techniques and heuristics must be used per level.

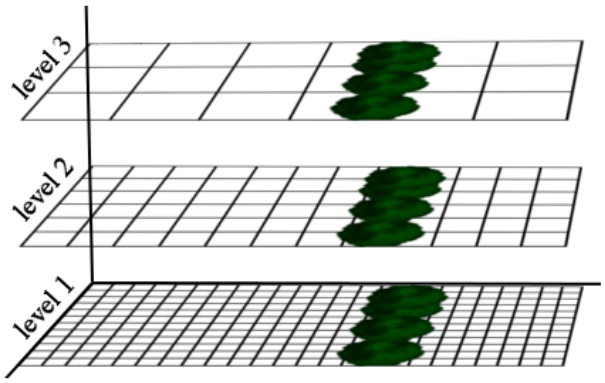

Figure 5: A hierarchical representation of a game world with 4 trees, using grids. Higher level representations are more coarse.

3 Pathfinding aware data structures.

Aggregating the results of the evaluated techniques, they all, one way or another, parse the entire world into a spatial representation. Even areas that are never explored by A* are still included. How is it that A* can use a heuristic to estimate a sense of direction, but the spatial structure remains agnostic in any regard. The latter question would be answered by path-caching and preprocessing phases - they require a generic data structure that is both predictable and consistent. For Guerrilla Tactics' procedural terrain generation, there literally is no time for a preprocessing phase. In this chapter I prepose a data structure which dynamically adapts itself to match the realtime requirements of a pathfinding algorithm such as A*, thus eliminating the need for a preprocessing phase, or parsing the entire world.

3.1 Dynamically growing spatial partitioning trees

In order for a spatial structure to only index relevant areas, it needs close ties to the search algorithm, e.g., when the algorithm searches northbound, the spatial structure would also need expand northbound. This expanding process needs to be efficient for any such structure to prove its value. One such solution would be to use a modified grid, when the algorithm requests the adjacent neighbours, the grid would in realtime calculate which entities are occupying the direct neighbours (tiles). This solution is highly inefficient, per neighbour calculation the entire list of entities has to be traversed.

In our implementation I used a modified quadtree. When inserting an entity into the tree's root node, we would not recurse at all, instead all entities are contained within the root node. When calculating adjacent neighbours of a space[1], we recursively calculate which entities are contained within that space - however, we only recursively partition the tree in the relevant direction. Observe image 6 for a conceptual northbound exploration, the example assumes an empty world.

Query by criterion It should be noted that we did not partition until the first empty space was found, but rather, we partition until a criterion was met. Criterions specify the maximum size and minimal size of a space, whether diagonal steps are permitted, as well as the number of entities allowed to exist in a space. The latter, for pathfinding, is usually zero. Enforcing a minimal size creates smoother paths, however at a cost of forcing A* to explore more spaces. A fine balance must be found between computationally expensive smooth paths and computationally cheaper coarse paths. In addition to our C++ implementation, a Java[2] application was created to tweak criterions and visualise the quadtree in realtime. Image 7 shows the differences between two criterions.

Entity types Guerrilla tactics features different entities. Each entity has different route-

[1]Space has been defined as having both a position and, width and height properties.

[2]Java implementation of the tree to be found here: https://github.com/Gerjo/spatialtree/

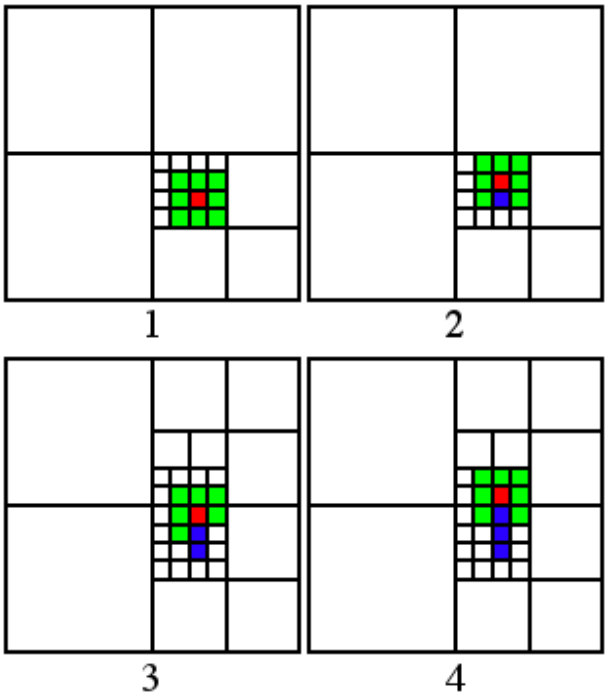

Figure 6: Northbound exploration, the red space indicates A's 'current-' node, the green spaces are adjacent neighbours, the blue tiles are a potential path generated by A*. Note how the quadtree only grows in the relevant direction, the left side of the tree is mostly ignored. In this example we force the tree to always recurse until a certain minimal space-size is reached, this creates a smoother path. Note how the tree does not require any partitioning between steps 1 and 2, nor, 3 and 4.*

requirements, e.g., amphibious vehicles are able to pass through water, whereas a tank cannot. Rather than creating multiple quadtrees, each node kept track of which entity types where present in its children. Via simple bit-wise operations this could be done with little overhead. Water, trees and other static object types are inserted into the tree, too. We made absolutely no distinction between moving (dynamic) and static entities.

Alignment issues An inherent problem with space partitioning is the fixed alignment of spaces, image 8 demonstrates this problem. The narrow corridor between the two black walls measures 22 pixels, the criterion on the left is set to 10 to 20 pixels, the right criterion to 5 to 20 pixels - however, only one criterion finds a valid path. Through trial and error optimal criterions had to be discovered, both our C++ and Java implementations featured debug modes which interactively visualise the tree and paths.

Broad-phase collision detection After pathfinding has been completed, one major artefact remains, namely, a partially created quadtree! By crafting a criterion that selects spaces based on the number of entities contained therein, the tree can be used for broad-phase collision detection. We did not explore other uses, however it would not be unthinkable to derive AI-influence maps from the tree as well.

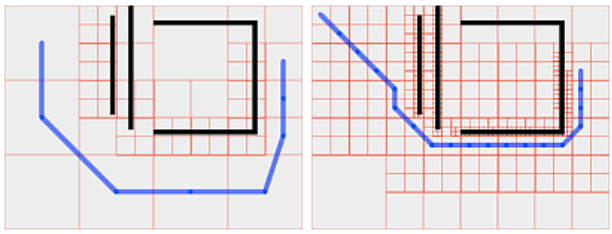

Figure 7: Differences in criterion, left enforces a space size between 50 and 150 pixels, right uses 10 and 40 pixels. The left criterion scans 33 spaces, the right 471 spaces. Screenshots taken from our Java demo application.

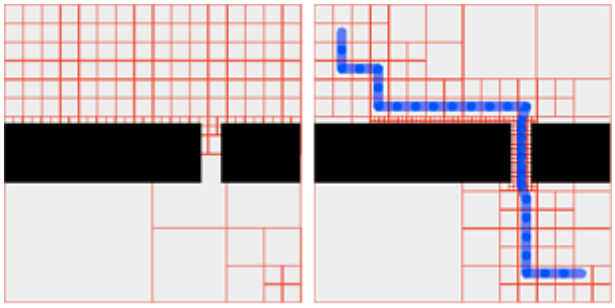

Figure 8: Dealing with narrow corridors. The criterion on the left does not find a path at all. Right shows a tuned criterion that does find a path, however at a cost of exploring more spaces.

4 Empirical results

In this section I show several demonstrations and their relevant statistics. All screenshots are taken from the Java implementation. The Java implementation features more debug information and a zoom feature. Time measurements are done via Java's System.nanoTime() high precision clock. When referring to "pathfinding" I include both building the tree and running A*.

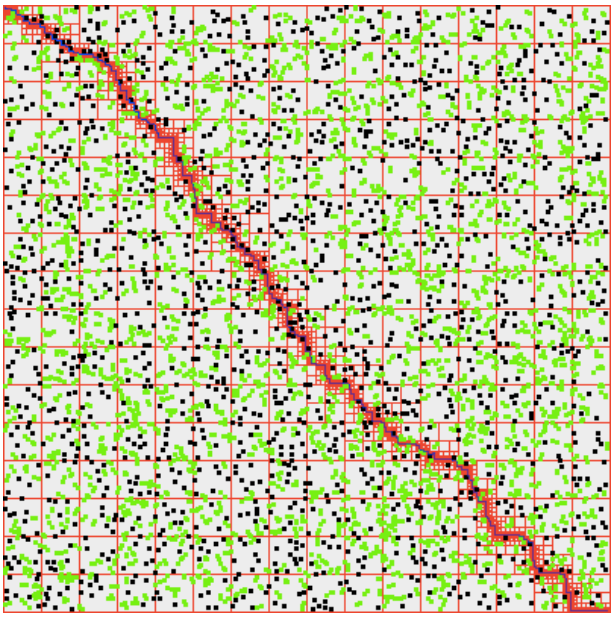

Figure 9: A 40,000 by 40,000 pixel world with 5.000 moving entities measuring 300 by 300 pixels. The size criterion has been set to 10 to 100 pixels. Pathfinding took 0.0054 seconds, collision detection took 0.0059 seconds. The final route contained 1747 waypoints with a length of 54,372 pixels. The quadtree contains 21,697 spaces. Green entities are those colliding with other entities.

5 Conclusion

Through empirical means I've shown that in a real time environment pathfinding aware data structures can perform on par with the set requirements. In my quadtree implementation I've used A*, however as it's just a data structure, any pathfinding algorithm should be compatible. Especially JPS might be worthwhile candidate.

Figure 10: A map from Baldur's gate measuring 700 by 700 pixels[14]. Using a 10 to 20 size criterion a path was found in 0.0024 seconds. The resulting tree contains 2629 spaces. The path was sought from left to right, permitting diagonal movement. Due to technical limitations, the black walls are broken up into 3195 entities with a 1 pixel width.

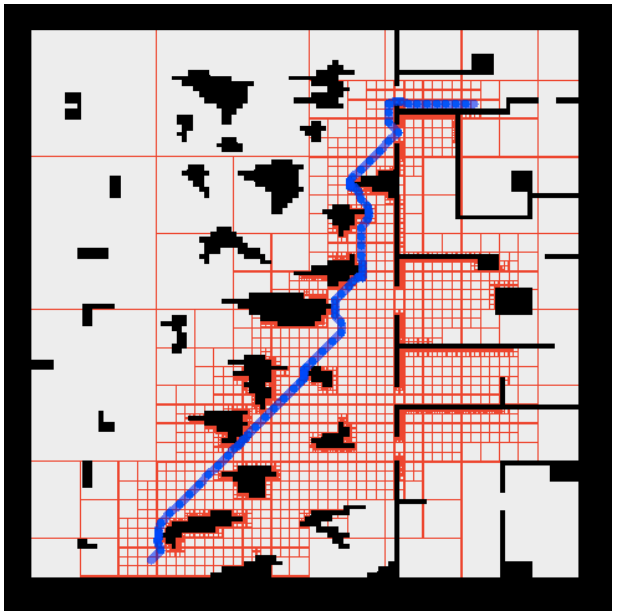

Figure 11: A 700 by 700 pixel map[14], using a 4 to 20 criterion a route was found in 0.0062 seconds. The final tree contains 16,957 spaces. A path was sought starting from the lower left corner, permitting diagonal movement.

References

[1] P. Yap, *Grid-Based Path-Finding.* Department of Computing Science, University of Alberta Edmonton, Canada T6G 2E8, 1999.

[2] D. Harabor, A. Botea and P. Kilby, *Path Symmetries in Undirected Uniform-Cost Grids.* NICTA and The Australian National University, 2011.

[3] D. Harabor and A. Grastien. *Online graph pruning for pathfinding on grid maps.* AAAI Conference on Artificial Intelligence (AAAI), San Fransisco, USA. 2011.

[4] M. Booth. *The AI Systems of Left 4 Dead.* Slides of Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford. 2009.

[5] D. Hale, G. Hunter, M Youngblood and P. Dixit. *Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds.* Artificial Intelligence and Interactive Digital Entertainment (AIIDE) (2008): 173-178.

[6] P. Tozour. *Building a Near-Optimal Navigation Mesh.* IAI Game Programming Wisdom, pages 171185. Charles River Media, Inc., 2002.

[7] A. Botea, M. Mller and J. Schaeffer. *Near optimal hierarchical path-finding.* Journal of game development 1.1 (2004): 7-28.

[8] C. Reynolds *Steering behaviors for autonomous characters.* Game Developers Conference. http://www.red3d.com/cwr/steer/gdc99 1999.

[9] S. Kambhampati and L. Davis. *Multiresolution path planning for mobile robots.* Robotics and Automation, IEEE Journal of 2.3 (1986): 135-145.

[10] A. Yahja, A. Stentz, S. Singh and B. Brumitt *Framed-quadtree path planning for mobile robots operating in sparse environments.* Proceedings of IEEE International Conference on Robotics and Automation. Vol. 1. IEEE, 1998.

[11] P. Tozour. *Search algorithms and search space demo 1.0.* http://www.ai-blog.net/ 2005.

[12] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, ISBN 978-0123694461. 2006.

[13] R. Holte, M. Perez, R. Zimmer and A. MacDonald. *Hierarchical A*: Searching Abstraction Hierarchies Efficiently.* In Proceedings AAAI-96, pages 530535, 1996.

[14] N. Sturtevant. *Benchmarks for Grid-Based Pathfinding* In Transactions on Computational Intelligence and AI in Games. Vol 4, no 2, pages 144-148. 2012.