A Sparse Grid Representation for Dynamic Three-Dimensional Worlds

Nathan R. Sturtevant
Department of Computer Science
University of Denver
Denver, CO, 80208
sturtevant@cs.du.edu

Abstract

Grid representations offer many advantages for path planning. Lookups in grids are fast, due to the uniform memory layout, and it is easy to modify grids. But, grids often have significant memory requirements, they cannot directly represent more complex surfaces, and path planning is slower due to their high granularity representation of the world. The speed of path planning on grids has been addressed using abstract representations, such as has been documented in work on Dragon Age: Origins. The abstract representation used in this game was compact, preventing permanent changes to the grid. In this paper we introduce a sparse grid representation, where grid cells are only stored where necessary. From this sparse representation we incrementally build an abstract graph which represents possible movement in the world at a high-level of granularity. This sparse representation also allows the representation of three-dimensional worlds. This representation allows the world to be incrementally changed in under a millisecond, reducing the maximum memory required to store a map and abstraction from Dragon Age: Origins by nearly one megabyte. Fundamentally, the representation allows previously allocated but unused memory to be used in ways that result in higher-quality planning and more intelligent agents.

Introduction

The choice of a world representation is a fundamental decision that influences the features that can be put into the path planning engine for a game. Most representations offer trade-offs with some tasks being extremely easy, while other tasks are more difficult. It is also important to include within the choice of representation the time required to implement the representation. While Blizzard had years to build a new path planning system for Starcraft 2[1], not every company has the time to invest in such a system. On his AiGameDev.com web site, Alex Champandard recommends waypoint graphs[2] “since they’re a simple and effective approach to navigation”, even though navigation meshes offer a richer feature set.

Grids have also been a popular representation (eg (Sturtevant 2007)) because they are easy to build, only requiring an array of blocked/unblocked values. But, because grids uniformly represent all space, they are not suitable for many games, particularly large worlds. This results from a combination of planning costs, which can be reduced via abstraction, and storage costs, because grid information is allocated for all cells whether or not they are passable. In addition, grids have usually been restricted to two-dimensional terrain. A grid (or voxel) representation of a full three-dimensional world would be even more expensive to maintain.

This paper introduces an alternate sparse grid representation. The hypothesis behind this design was that a grid representation could be enhanced to achieve four purposes. First, to avoid storing areas of the map which are not traversable. Second, to represent three-dimensional worlds. Third, to easily allow points to be added to and removed from the world at runtime. Fourth, and finally, to quickly automatically build and update an abstract world representation. The representation described here achieves all of these metrics.

Post-mortem analysis of the maps that shipped with Dragon Age: Origins suggests that the original efforts focused on reducing the memory used by the abstraction layer were misplaced. The underlying grid uses approximately 10 times more memory than the abstraction layer, and thus the grid should have been the focus of optimization efforts.

The remainder of the paper is as follows. First, background material on path planning approaches are covered, as well as the tasks that a path planning system would be expected to handle. Then, the new design is described in detail. Finally, detailed experimental results show that this system is suitable for being deployed in a commercial system.

Background and Problem Definition

There are a number of general tasks related to path planning that any representation must be able to handle. When a planning task arrives, the location of the relevant agent within the representation is known, but a target location will often be in arbitrary coordinates that must then be converted into the representation format. This process is called localization. It is simple on a grid, as real-valued x and y coordinates must only be divided by the grid resolution to find the integer grid coordinates.

Most games feature dynamic worlds, with creatures or other objects modifying the underlying representation. Ide-

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]Described in detail in a 2011 GDC AI Summit talk

[2]http://aigamedev.com/open/reviews/alienswarm-node-graph/

ally, a path planning representation can be easily changed to block or unblock locations within the world. Again, this is simple on a grid, as grid cells in the representation can be quickly localized and modified.

In addition to creatures, the cost of moving in the world can also be modified by a number of influences. This might include area effects, such as a spell being cast in a role-playing game, or lines of fire which should be avoided in a first-person shooter. Locations with such modifications are still passable, but are more expensive to pass than the regular terrain. Thus, routes without such effects applied would be preferred. If planning occurs over multiple levels of abstraction, such costs should be taken account in all levels of abstraction, otherwise high-level planning will not properly take into account the costs of crossing low-level terrain. Finally, any representation must be amenable to fast planning and re-planning.

There are three dominant representations used in the industry (Tozour 2002b). These include:

- *Grids*, which are covered in detail in this paper. Localization and modifying grids is simple, but some sort of abstraction mechanism is usually needed to speed path planning in grids. Paths resulting from planning in grids usually need an extra smoothing pass.
- *Waypoint graphs*, which are a graph representation of the world. Modifying waypoint graphs is relatively straightforward, but there isn't a tight coupling between the graph and each individual point in the world, which can make reasoning about terrain and the relationship to the waypoint graph more difficult. Agents on waypoint graphs usually stay on the graph and its edges, resulting in lower quality movement.
- *Navigation meshes*, or a nav mesh for short, which can be represented by triangles (Demyen and Buro 2006) or by polygons (Tozour 2002a). Meshes implicitly form graphs, but each point in the graph represents a distinct area of terrain in the mesh. Localization on meshes can be made efficient with an underlying grid (Demyen 2006). Meshes can be modified at runtime, but the process can be difficult to implement. Paths through meshes are smoothed using the funnel algorithm.

Nav meshes and waypoint graphs are high-level terrain representations that greatly reduce planning costs. But, as a result, they may not be able to easily account for small areas of terrain with higher cost, such as a trap that has been discovered and should be avoided. Modifying nav meshes can also be difficult as geometric algorithms that process and reason about meshes do not always handle special cases like parallel lines or congruent points easily[3]. It is possible to represent the walkable surface of complex terrain using nav meshes and waypoint graphs, as they are not tied to a single plane in any way.

A high-level representation can be built from grids (Sturtevant 2007) resulting in something similar to a nav mesh, except that the underlying regions in the graph are not always convex. We will describe this approach in more detail in the next section, as our approach is based on this previous work.

[3]See work and blog entries by Mikko Mononen on the Recast tool http://code.google.com/p/recastnavigation/

Sparse Grid Representation

We begin by describing previous work done in building an abstract representation for Dragon Age: Origins (DAO) and then show how it must be extended to introduce the properties we desire in our modified representation. There are many variations on the design choices described here which could be changed to meet specific game constraints.

Overall, the world is divided into two levels of granularity. The underlying grid is a fine-grained representation suitable for local movement planning, and the representation of small obstacles. From this grid, a high-level graph is built which is suitable for longer range planning. The high-level graph is built from connected regions in the underlying grid. Even higher levels of abstraction can be used; this paper focuses on a single level of abstraction, as additional levels of abstraction can be built with either the same principles or using other techniques (Sturtevant and Geisberger 2010).

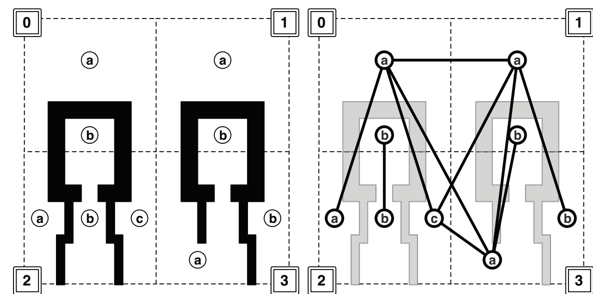


Figure 1: The abstraction used in Dragon Age: Origins.

Dividing the World into Sectors and Regions

The low-level map representation in DAO is a grid, but planning full paths on the grid was too expensive, so a high-level representation was built by dividing the world up into sectors and regions. This is shown in the left portion of Figure 1, which is taken from (Sturtevant 2007). This map is divided into four large squares, or sectors, which overlay the map. The corner of each sector is labeled with an index. The sectors are then sub-divided into regions, where all points in a region are reachable from each other without leaving the sector. Dark areas of the map are uncrossable walls, so sector 2 has three regions, while the remaining sectors have two regions apiece.

The right portion of this figure shows the induced high-level graph. Edges are added between regions if there is an edge in the low-level grid which connects two regions in different sectors. This representation guarantees that any high-level path can be refined into a low-level path, and that every low-level path has a corresponding high-level path.

The focus of the design of this abstraction for DAO was to minimize the memory overhead used by the abstraction. Hence, the abstract edges and almost all of the data for the

abstraction is stored compactly in a single vector. This creates problems, however, if the grid is changed. Removing abstract edges could be done easily, but if new edges are needed, there may not be space to store them directly. This could require re-organizing the entire data structure, an expensive operation. As a result, the DAO grid world is left static, except for changes in traversal costs as a result of area effects and moving NPCs.

Sparse Grid Design

Thus, we begin by re-designing the data structures used to implement this same abstraction in a way that the underlying grid and high-level graph can both be easily modified as the world changes.

Points in our space can be represented both by tuples of integers, (x, y, z), and by internal sector and region information. Localizing a point within a sector is simple, so at the highest level there is an array of sectors, with each sector containing a list of regions in the sector. In the DAO representation a global map stored all the underlying data for the grid, and regions boundaries were implicit from the obstacles in the grid. We modify this by explicitly storing the grid information within each region. In full, each region contains:

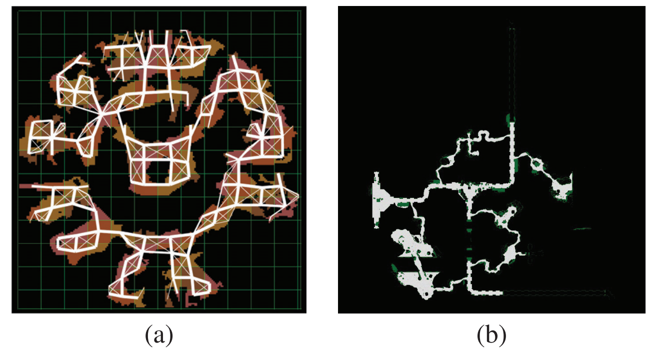
- A *center location* representing a point at the approximate middle of the region.
- A *base height*. The heights of individual grid cells are stored as offsets to this height.
- A count of *free* and *partially blocked* cells. This information is not strictly necessary, but can be used to propagate local travel costs into the abstract graph. If many cells are blocked in the region, the cost of traversing an edge incident to the region should be more expensive.
- A list of *graph edges* incident to the region. Each edge includes the destination sector, region, and the edge *support*, which is the number of underlying grid edges that would have to be removed for the abstract edge to be removed.
- A pointer to the low-level grid data for the region.

The first four items are packed into 32 bits (8 bits each), while the edges and grid data require a pointer. Each edge is stored in 32 bits, and each grid location is stored in 16 bits. The low-level grid contains:

- One bit for each of the 8 directions of movement representing whether movement in that direction is possible. This means that grid edges are stored explicitly instead of implicitly. This simplifies the process of checking if there are passable neighbors, particularly when the neighbors might be in a different sector or region.
- Five bits of height information, meaning that within a region there are 31 possible height values, with the last height value representing that a cell is impassable.
- Three bits representing how many local area effects are currently applied to the cell. These counts are aggregated within the full region.

A map from the DAO benchmark problem set[4] built using this representation is shown in Figure 2(a). The white lines represent the abstract edges in the graph. The thickness of a line represents its underlying support in the grid. Note that if there are multiple regions in the same sector, they are stored separately, effectively doubling the memory required to store the map. For the map in this figure, there are 47 sectors which contain no regions. But, because there are multiple regions in other sectors, the final representation has 169 total sectors and 164 total regions. Thus, the savings by not storing the data in every sector offsets the fact that regions are stored multiple times. The largest savings are found in the map in Figure 2(b), which is mostly empty space. Full storage results are contained in the experimental results. The alternate advantage of using this approach is that modifying the map and abstraction becomes much easier. We will show how this is done after comparing the total memory usage in our new design to the memory used by the DAO representation.

[4]http://movingai.com/benchmarks/dao/



(a) (b)

Figure 2: (a) An example map showing the abstract graph representation and underlying map. (b) The map where the sparse grid representation provides the largest gain.

Memory Usage

The DAO abstraction used 32 bits for each sector, storing an array index for the region data and the number of regions in each sector. Our new representation uses the same amount of memory per sector on a 32-bit system, but the use of a pointer requires more memory on a 64-bit system.

The DAO abstraction stored each edge in a single byte. Each region contained the center location and the number of edges, requiring 16 bits total. Thus, each region is 2 bytes plus 1 byte per edge. The new representation uses 32 bits per region plus two pointers, and edges are stored in four bytes instead of a single byte. Edges could be compressed into less storage, but alignment issues could result, so we aligned data structures to 32-bit boundaries. This means that regions now require 12 bytes plus 4 bytes for each edge. We assume that the underlying grid requires the same number of bytes in both representations (2 bytes per cell), although more memory may be used if other meta-information about the world is stored directly in the grid.

The DAO representation and our new representation will both have the same number of regions and edges, but the new representation will not store any grid information for sectors

Algorithm 1 Greedy Point Addition($p = \{x, y, z\}$)

```
1: sector ← getSector(x, y)
2: if sector has no regions then
3:     create new region in sector and add p
4: else
5:     for all regions r in sector do
6:         if p can be added to r then
7:             add p to r
8:             return
9:         end if
10:    end for
11:    create new region in sector and add p
12: end if
```

with no passable points. The new representation requires a two-step lookup for localization, first finding the sector, and then the region for a given point.

Adding Points to the Map

Within each sector, if there are overlapping z coordinates or if the range of z values is too large, points must be partitioned into separate regions. This partitioning could be done offline using an approach like watershed segmentation (Mangan and Whitaker 1999), but as the map is expected to change at runtime, we use a greedy approach, which could be further optimized if necessary.

Maps are built incrementally by adding passable (x, y, z) coordinates to the representation. We assume that diagonal edges cannot be crossed unless both of the adjoining cells are passable. This means the order of adding cells to the map matters. The following algorithm is used to add passable grid cells to the representation. This works in two stages. First, points are added to the representation, and then the edges are added. As stated previously, all edges are explicitly stored in the representation.

Adding Points The code in Algorithm 1 describes how points are added to the world. Given a point, (x, y, z), the sector containing this point is found using simple division. If the sector is empty, a new region is created and the point is added. Otherwise each region is tested in turn to see if the point can be added. This involves testing to see if the point is adjacent to an existing point in the region and whether the point falls within the limits of the heights representable in each region.

Adding a point could enable two adjacent regions to be merged. This requires removing the points from one region and adding them to the other, but for simplicity we currently do not merge split regions.

Adding Edges After a point has been added to a region, the edges must be added. This is done by finding all points in the 8 adjacent cells to the cell that was just added. These points may be in the same region, a different region, or even a different sector. There are then 12 possible edges that can be added to the world, shown in Figure 3. Most of these edges are obvious, as they involve the cell being added, A. But, if cells B, C and D are already part of the world, then adding A allows a diagonal edge to be added between B and D. These extra diagonals account for an additional 4 edges.

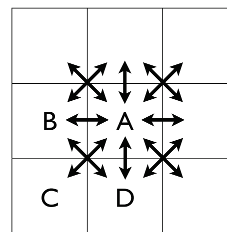


Figure 3: The possible edges which must be checked when adding the grid cell marked “A”.

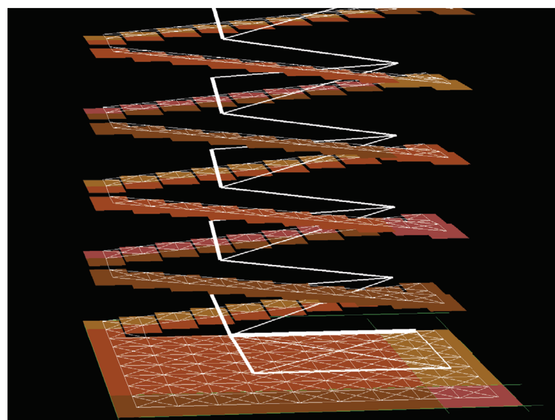


Figure 4: An example staircase built using multiple regions in the same sector.

Each of these edges are added to the low-level grid, even if the neighbor of a cell is in a different sector or region. But, when this occurs, an abstract edge is either added to the region itself or, if the abstract edge already exists, the support of the existing edge is increased.

All edges are stored in both the origin and destination cells, and if applicable, in the origin and destination regions as well. This allows directed edges, such as the ability to jump down from a height that can’t be climbed, however we do not use this representational capability in this implementation.

The addition of edges between regions in the same sector is what allows three-dimensional maps to be created. Suppose a map needs to represent a staircase, like the one shown in Figure 4. In such a situation it is possible for the stairs to all be in the same sector, and for every state in the sector to be reachable without leaving the sector. In the DAO representation this would require putting all points in the same region, which isn’t possible. With edges between regions in the same sector, the points can be put in different regions with edges in between them, still representing the space properly.

Removing Points from the Map

Removing maps from the representation takes place using a similar process to the way points were added. As before, the

points surrounding the cell being removed are assembled, and the 12 edges that were potentially added (Figure 3) are now removed from the low-level grid. Removing these edges also reduces the support of any abstract edges, and if the support reaches 0, then the abstract edge is removed.

The key complication in removing a point from the map is that it might split a previously passable region into two separate regions. Thus, after removing a point, the connectivity of a region must be tested. If the region is no longer connected, the smallest $k-1$ of the k resulting regions are removed and re-added to the representation. This is done directly using the point adding procedure described above.

When adding or removing points from the representation, the location of the abstract region center must be recomputed. This is done by finding the point in the region which is closest to the center of the region, although other optimizations (Sturtevant 2007) can reduce the work required for path planning.

Generating Successors in Search

During an A* search, the successors of any state must be generated. In our sparse grid representation the successors of every grid cell are explicitly stored, which eliminates the need to find and test all neighbors for passability. But, because neighbors can be in different sectors or regions, the lookup of known neighbors can still be expensive. This is demonstrated in the experimental results, where several techniques for speeding up this process are evaluated. Generating abstract successors of a region is straightforward, as all of the edges are explicitly stored within the region.

Note that in the DAO representation, a small search is used to identify the current region. Because regions are now stored explicitly, this is no longer necessary. This can also make pathfinding easier, because instead of finding the path to the center of a region, a path can be found to any grid cell within a region, as the region associated with a cell is always known.

Experimental Results

On 2-dimensional maps the representation described in this paper will be identical to the abstraction used in DAO. The speed-up of the DAO approach over A* has already been studied in several contexts (Sturtevant 2007; Sturtevant and Geisberger 2010). So, in our experimental results we focus on other properties. First, we look at the total memory used. Then, we look at the cost of generating successors. Finally, we look at the cost of modifying the map by adding or removing successors.

All experiments were run on a 2.66GHz Intel Core i7 with 8GB of RAM. Only a single core and a fraction of the total RAM were used in the experiments. The code which implements this representation has been released as open source for use by other researchers or those in the game industry[5]. Any comparison to the existing DAO abstraction was done using the code that shipped with AI Wisdom 4 (Rabin 2008). The code base was implemented over a few weeks time, most of which was used for refining the design described here.

[5]See http://code.google.com/p/hog2/

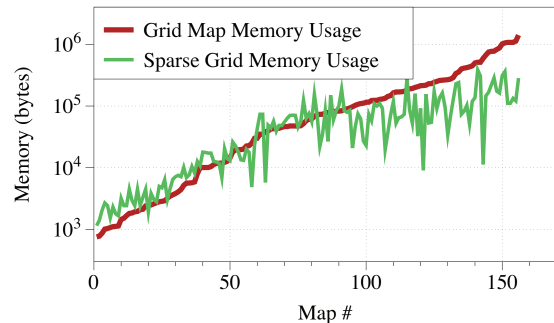


Figure 5: Distribution of memory used per map.

Table 1: Average time time to generate all successors on each legal state of every map.

Sector	Regular	Naive	Smart	1-bit	8-bit
8	0.15 μs	0.24 μs	0.19 μs	0.17 μs	0.12 μs
12	0.15 μs	0.23 μs	0.17 μs	0.14 μs	0.11 μs
16	0.15 μs	0.23 μs	0.17 μs	0.13 μs	0.10 μs

Total Memory Usage

To analyze memory usage we looked at the set of maps that shipped with Dragon Age: Origins, measuring the total memory used by both the abstraction and the underlying grid. We sorted the results according to the memory used by the original representation, showing the results in Figure 5. Note that the y-axis is a logarithmic scale. This is necessary to clearly distinguish the curves across the range of map sizes. These lines show the distribution for sectors size 12, but different sector sizes do not result in a significant difference in the curves.

Over all the maps, the average memory used by the previous approach was 178,390 bytes versus 93,064 bytes for the sparse grids. More importantly the maximum memory required to store any map was reduced from 1,457,090 bytes to 519,292 bytes, almost a 3-fold reduction.

We performed the same analysis on maps from Dragon Age II. This game has a smaller scope, so the largest map only required 600k of memory with the regular representation, which was reduced to 200k with the sparse representation.

Generating successors

Although we could measure the cost of pathfinding, the cost of a search includes many other overheads which are independent of representation. Instead we directly measure the cost of successor generation by measuring the average cost of generating all successors over all legal states in all DAO maps. The results are in Table 1.

A regular grid implementation takes 0.15μs to generate successors, regardless of the sector size. On the sparse

Table 2: Average time to remove and re-add 99 cells from the middle of a map.

Sect. size	Avg. cut	Max. cut	Avg. add	Max add
8	0.081 ms	0.263 ms	0.059 ms	0.129 ms
12	0.120 ms	0.624 ms	0.076 ms	0.162 ms
16	0.172 ms	0.804 ms	0.100 ms	0.207 ms

grid, successor generation isn't completely straightforward. A state is internally represented by a sector, region, and region offset, while successors are based on $x/y/z$ coordinates; moving between these representations isn't free.

A naive implementation, which switches representations to generate successors, and then switches back, takes 0.23-0.24μs on average to generate successors. A smarter implementation only switches between representations at the borders of sectors and regions, where successors might be in a different sector and/or region. This implementation is still slower than a regular grid, taking 0.17-0.19μs.

If extra memory available, even better performance can be achieved. Using 1-bit to cache whether a state has neighbors in a different sector or region results in faster performance, except on sectors of size 8; smaller sectors have a larger percentage of cells on the border between sectors and regions. Using 8-bits to mark every edge as external or internal to the region is up to 33% faster than regular successor generation. This memory overhead could also speed the regular implementation, but may not be affordable without the memory savings made possible by the sparse grid.

Modifying the map

One of the key features of our sparse grids is the ability to modify the map. We demonstrate this by cutting out a strip of 99 cells in a row in the middle of each map and then adding it back in again. We then measure the average time needed to remove and add the 99 cells separately, as well as the maximum time required for each set of operations. The major cost associated with removing points from the map is checking to see if a region has been split. We aggregate the calls together, so regions are only recomputed once per sector. Checking for region splits at each step approximately doubles the average and maximum time required to cut 99 cells from the map.

The overall results are in Table 2. The average cut and add time depends on the sector size. This is because it takes more time to check if regions have been split when there are larger sectors. But, even the maximum cut time is under 1ms, which is sufficiently fast, as this is not a common operation. Note that temporary changes to a map are performed by weighting the relevant cells. Adding cells to the map is much cheaper, taking at most 0.207ms even on the largest sector sizes.

Conclusions

In this paper we have offered an alternate representation for grid-based maps and grid-based map abstractions. This representation eliminates the need to store grid data for portions of a map which can never be traversed. Experiments with the set of maps that shipped with Dragon Age: Origins, shows that this approach can reduce the maximum memory required to store a map from approximately 1.5MB to 500kb.

In addition to reducing the memory overhead, the representation is also able to represent three-dimensional maps, and it is easy to modify the map at runtime. The cost of this approach is that successor generation is slightly slower than before, unless memory is used to offset the speed penalty.

There are a number items of future work. We have only completed a few iterations of optimization on the code, and so additional optimizations are still possible. For instance, we currently store a separate grid for each region. Sharing grids between regions would further reduce memory, but could increase the time for adding and repairing cells. Also, while sparse grids can represent many classes of three-dimensional maps, it is assumed that all creatures in the map are affected by gravity and walk on flat surfaces. It is an open question whether this work could be adapted for vertical surfaces. One possible approach would be to allow vertical connections between cells. These could either model walkable walls, or ladders that can be climbed. Finally, caching schemes could be used to avoid or reduce the allocation of memory at runtime.

Although some have questioned the usefulness of grids in modern games, this work shows that they are still a valid representation to be considered. We understand that they are not suitable for every game, but, given the enhancements described here, they are still suitable for use in many games.

Acknowledgements

We appreciate the feedback on this work from the reviewers, as well as cooperation with Gavin Burt and BioWare.

References

Demyen, D., and Buro, M. 2006. Efficient triangulation-based pathfinding. In *Proceedings of the 21st National Conference on Artificial Intelligence*, 942–947. AAAI Press.

Demyen, D. 2006. Efficient triangulation-based pathfinding. Master's thesis, University of Alberta.

Mangan, A. P., and Whitaker, R. T. 1999. Partitioning 3d surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics* 5:308–321.

Rabin, S., ed. 2008. *AI Game Programming Wisdom 4*. Charles River Media.

Sturtevant, N. R., and Geisberger, R. 2010. A comparison of high-level approaches for speeding up pathfinding. In *AIIDE*.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Tozour, P. 2002a. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom. (S. Rabin, ed.)*, 171–185.

Tozour, P. 2002b. Search space representations. In *AI Game Programming Wisdom 2. (S. Rabin, ed.)*.