# 16

# Theta* for Any-Angle Pathfinding

*Alex Nash and Sven Koenig*



## 16.1 Introduction

One of the central problems in game AI is finding short and realistic-looking paths. Pathfinding is typically divided into two steps: discretize and search. First, the **discretize** step simplifies a continuous environment into a graph. Second, the **search** step propagates information along this graph to find a path from a given start vertex to a given goal vertex. Video game developers (and roboticists) have developed several methods for discretizing continuous environments into graphs, such as 2D regular grids composed of squares (square grids), hexagons or triangles, 3D regular grids composed of cubes, visibility graphs, circle-based waypoint graphs, space-filling volumes, navigation meshes, framed quad trees, probabilistic road maps, and rapidly exploring random trees [Björnsson 03, Choset 04, Tozour 04].

Due to its simplicity and optimality guarantees, A* is almost always the search method of choice. A* is guaranteed to find shortest paths on graphs, but shortest paths on graphs are not equivalent to shortest paths in the continuous environments. A* propagates information along graph edges and constrains paths to graph edges, which artificially constrains path headings. Consider Figures 16.1 and 16.2, in which two different continuous environments have been discretized into a square grid and a navigation mesh, respectively. The shortest paths on the square grid and the navigation mesh (Figure 16.1) are longer than the shortest paths in the continuous environment (Figure 16.2) and are unrealistic looking due to either a heading change in free space or a heading change that does not *hug* a blocked cell.

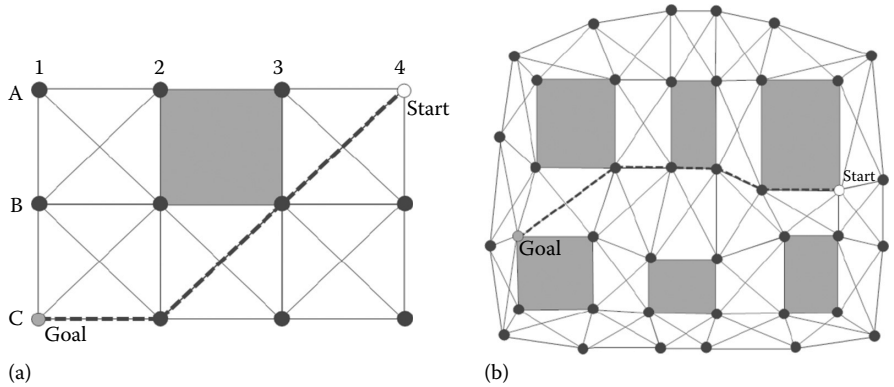



Figure 16.1

Grid paths: square grid (a) and navigation mesh (b). (Adapted from Patel, A., Amit's game programming information, 2000, Retrieved from http://www-cs-students.stanford.edu/~amitp/gameprog.html, accessed September 10, 2014.)

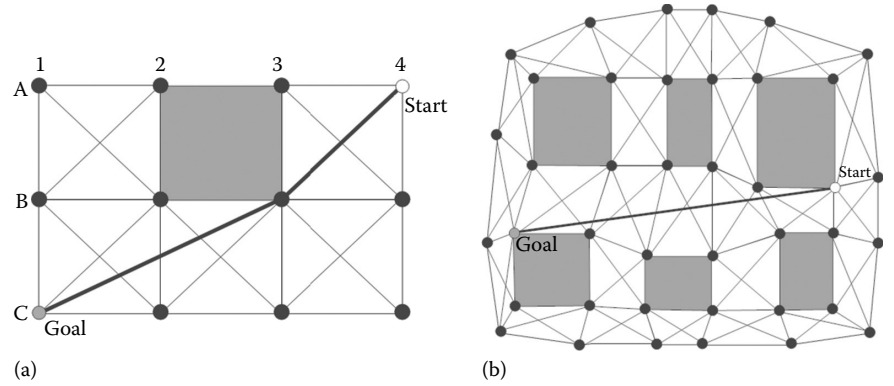



Figure 16.2

Any-angle paths: square grid (a) and navigation mesh (b). (Adapted from Patel, A., Amit's game programming information, 2000, Retrieved from http://www-cs-students.stanford.edu/~amitp/gameprog.html, accessed September 10, 2014.)

The fact that A* paths can be long and unrealistic looking is well known in the video game community [Rabin 00]. The paths found by A* on eight-neighbor square grids can be approximately 8% longer than the shortest paths in the continuous environment [Nash 12]. The typical solution to this problem is to use a postprocessing technique to shorten the A* paths. One such technique is to remove vertices from the path, such that they look like *rubber bands* around obstacles (A* PS). This technique shortens paths such that all of the heading changes on the path *hug* a blocked cell.

However, choosing a postprocessing technique that consistently shortens paths is difficult because there are often several shortest paths on a given graph, and a

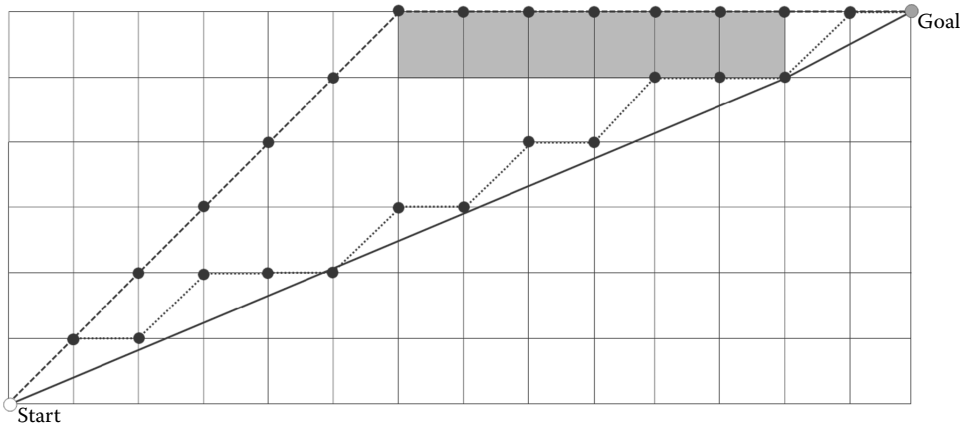



Figure 16.3

A* paths with different postprocessing techniques.

postprocessing technique typically shortens some of them more effectively than others. For example, A* with the octile distance heuristic finds paths very efficiently on eight-neighbor square grids. (The octile distance is the shortest distance between two vertices on an eight-neighbor square grid with no blocked cells.) However, these paths are often difficult to shorten because the search tends to find paths in which moves in the diagonal directions appear before moves in the cardinal directions (for the tie-breaking scheme explained in the next section). The dashed path in Figure 16.3 depicts this behavior.

On the other hand, A* with the straight-line distance heuristic finds paths that are of the same lengths as those found by A* with the octile distance heuristic, albeit more slowly due to additional vertex expansions. However, paths found using the straight-line distance heuristic are often shortened more effectively because they tend to follow the shortest paths in the continuous environment. The dotted path in Figure 16.3 depicts this behavior. In general, postprocessing techniques do improve paths, but they provide trade-offs between path length and runtime that are difficult to chose between, and they do not address the fundamental issue, namely, that the search only considers paths that are constrained to graph edges [Nash 07, Ferguson 06].

We address this issue by describing a different approach to the search problem, called any-angle pathfinding. Specifically, we describe Theta*, a popular any-angle pathfinding algorithm. The development of Theta* was motivated by combining the desirable properties of A* on two different discretizations of the continuous environment:

- *Visibility graphs*: Visibility graphs contain the start vertex, the goal vertex, and the convex corners of all blocked cells [Lorzano-Perez 79]. A vertex is connected via a straight line to another vertex if and only if it has line of sight to the other vertex, that is, the straight line from it to the other vertex does not pass through a blocked cell or between blocked cells that share a side. Shortest paths on visibility graphs are also

shortest paths in the continuous environment. However, pathfinding is slow on large visibility graphs since the number of edges can be quadratic in the number of vertices.

- *Grids*: Pathfinding is faster on grids than visibility graphs since the number of edges is linear in the number of cells. However, shortest paths on grids can be longer than shortest paths in the continuous environment and unrealistic looking since the path headings are artificially constrained to grid edges [Nash 07].

Theta* combines the desirable properties of these pathfinding techniques by propagating information along graph edges (to achieve short runtimes) without constraining paths to graph edges (to find short *any-angle* paths).

Theta* is easy to understand, quick to implement and provides a good trade-off between path length and runtime. It quickly finds short and realistic-looking paths and can be used to search any Euclidean graph. The pseudocode for Theta* is very similar to the pseudocode for A*, and both pathfinding algorithms have similar runtimes. Despite this, Theta* finds paths that have nearly the same length as the shortest paths in the continuous environments without the need for postprocessing techniques.

## 16.2 Problem Formalization

For simplicity, this article focuses on eight-neighbor square grids in which a 2D continuous environment is discretized into square cells that are either blocked (gray) or unblocked (white). Vertices are placed at cell corners rather than cell centers, and paths are allowed to pass between diagonally touching blocked cells. Neither of these two assumptions are required for Theta* to function correctly. Our goal is to find a short and realistic-looking path from a given start vertex to a given goal vertex (both at the corners of cells) that does not pass through blocked cells or between blocked cells that share a side.

In the following pseudocode, $s_{start}$ is the start vertex of the search, and $s_{goal}$ is the goal vertex of the search. $c(s, s')$ is the straight-line distance between vertices $s$ and $s'$, and `lineofsight(s, s')` is true if and only if they have line of sight or, synonymously, they are visible from each other. *open.Insert*($s$, $x$) inserts vertex $s$ with key $x$ into the priority queue *open*. *open.Remove*($s$) removes vertex $s$ from *open*. *open.Pop*() removes a vertex with the smallest key from *open* and returns it. For A*, we break ties among vertices with the smallest key in the open list in favor of vertices with the largest $g$-value. This tiebreaking scheme can reduce the runtimes of A*, especially when used with the octile distance heuristic. Finally, $neighbor_{vis}(s)$ is the set of neighbors of vertex $s$ that have line of sight to $s$.

## 16.3 A* Algorithm

Theta* builds upon A* [Nilsson 68], and thus we introduce it here. The pseudocode for A* can be found in Figure 16.4. A* uses heuristic values ($h$-values) $h(s)$ for each vertex $s$ that approximate the goal distances and focus the search. A* maintains two values for every vertex $s$: the $g$-value and the parent. The $g$-value $g(s)$ is the length of the shortest path from the start vertex to $s$ found so far. The parent *parent*($s$) is used to extract the path after the search terminates. Path extraction is performed by repeatedly following parents from the goal vertex to the start vertex. A* also maintains two global data structures: the open list and the closed list. The open list *open* is a priority queue that contains the vertices to be considered

```
1  Main()
2      open := closed := ∅;
3      g(S_start) := 0;
4      parent(S_start) := S_start;
5      open.Insert(S_start, S_start) + h(S_start));
6      While open ≠ ∅ do
7          s := open.Pop();
8          if s = s_goal then
9              return "path found";
10         closed := closed ∪ {s};
11         foreach s' ∈ neighbor_vis(s) do
12             if s' ∉ closed then
13                 if s' ∉ open then
14                     g(s') := ∞;
15                     parent(s') := NULL;
16                 UpdateVertex(s,s');
17     return "no path found";
18 end
19 Update Vertex(s,s')
20     g_old := g(s');
21     ComputeCost(s,s');
22     if g(s') < g_old then
23         if s' ∈ open then
24             open.Remove(s');
25         open.Insert(s',g(s')+h(s'));
26 end
27 ComputeCost(s,s')
28     /* Path 1 */
29     if g(s) + c(s,s') < g(s') then
30         parent(s') := s;
31         g(s') := g(s) + c(s,s');
32 end
```

Figure 16.4

Pseudocode for A*.

for expansion. For A*, we break ties among vertices with the smallest key in the open list in favor of vertices with the larger $g$-value. This tiebreaking scheme can reduce the runtimes of A*, especially when used with the octile distance heuristic. The closed list `closed` is a set that contains the vertices that have already been expanded. A* updates the $g$-value and parent of an unexpanded visible neighbor $s'$ of vertex $s$ (procedure **ComputeCost**) by considering the path from the start vertex to $s$ [= $g(s)$] and from $s$ to $s'$ in a straight line [= $c(s, s')$], resulting in a length of $g(s) + c(s, s')$ (Line 29). It updates the $g$-value and parent of $s'$ if this path is shorter than the shortest path from the start vertex to $s'$ found so far [= $g(s')$].

As noted by Rabin [Rabin 00], A* paths often appear as though they were constructed by someone who was drunk. This is both because the paths are longer than the shortest paths in the continuous environment and because the path headings are artificially constrained by the grid. As we mentioned earlier, postprocessing techniques can shorten A* paths but are often ineffective. This is because A* only considers paths that are constrained to grid edges during the search and thus cannot make informed decisions about other paths. Theta*, on the other hand, also considers paths that are not constrained to grid edges during the search and thus can make more informed decisions during the search.

## 16.4 Theta* Algorithm

The key difference between Theta* and A* is that Theta* allows the parent of a vertex to be any visible vertex, whereas A* requires the parent to be a visible neighbor. So, the pseudocode for Theta* is nearly identical to the pseudocode for A*. Only the procedure ComputeCost is changed; the new code can be found in Figure 16.5. We use the straight-line distance heuristic $h(s) = c(s, s_{goal})$ to focus the search. Theta* is identical to A* except that Theta* updates the g-value and parent of an unexpanded visible neighbor $s'$ of vertex $s$ by considering the following two paths in procedure ComputeCost, as shown in Figure 16.5.

- *Path 1*: As done by A*, Theta* considers the path from the start vertex to $s$ [= $g(s)$] and from $s$ to $s'$ in a straight line [= $c(s, s')$], resulting in a length of $g(s) + c(s, s')$ (Line 41).
- *Path 2*: To allow for any-angle paths, Theta* also considers the path from the start vertex to parent(s) [= $g(parent(s))$] and from $parent(s)$ to $s'$ in a straight line [= $c(parent(s),s')$], resulting in a length of $g(parent(s)) + c(parent(s),s')$, if $s'$ has line of sight to $parent(s)$ (Line 36). The idea behind considering Path 2 is that Path 2 is guaranteed to be no longer than Path 1 due to the triangle inequality if $s'$ has line of sight to $parent(s)$.

Theta* updates the $g$-value and parent of $s'$ if either path is shorter than the shortest path from the start vertex to $s'$ found so far [= $g(s')$]. For example, consider Figure 16.6, where B3

```
33 ComputeCost(s,s')
34   if lineofsight(parent(s),s')then
35     /* Path 2 */
36     if g(parent(s)) + c(parent(s),s')< g(s')then
37       parent(s') := parent(s);
38       g(s') := g(parent(s)) + c(parent(s),s');
39   else
40     / * Path 1 */
41     if g(s) + c(s,s')< g(s')then
42       parent(s'):= s;
43       g(s'):= g(s) + c(s,s');
44 end
```

Figure 16.5

Pseudocode for Theta*.

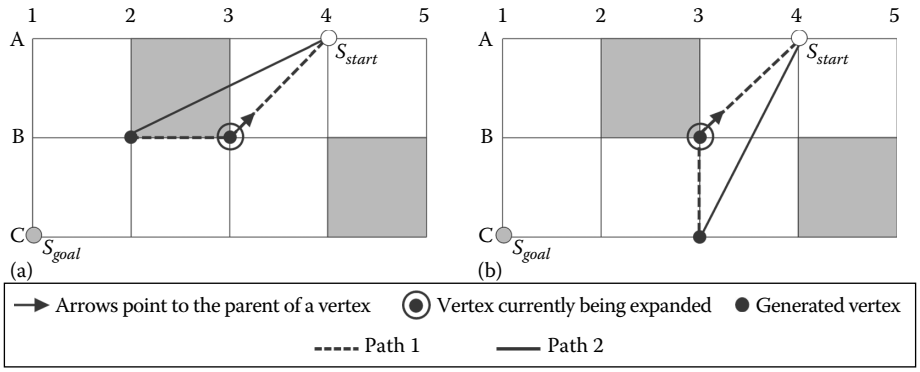



Figure 16.6

Theta* updates a vertex according to Path 1 (a) and Path 2 (b).

(with parent A4) gets expanded. The Path 1 rule is used when generating B2 because it *does not* have line of sight to A4. This is depicted in Figure 16.6a. The Path 2 rule is used when generating C3 because it *does* have line of sight to A4. This is depicted in Figure 16.6b.

Figure 16.7 shows a complete trace of Theta*. Arrows point from vertices to their parents. The concentric circles indicate the vertex that is currently being expanded and solid circles indicate vertices that have already been generated. The start vertex A4 is expanded first, followed by B3, B2, and C1.

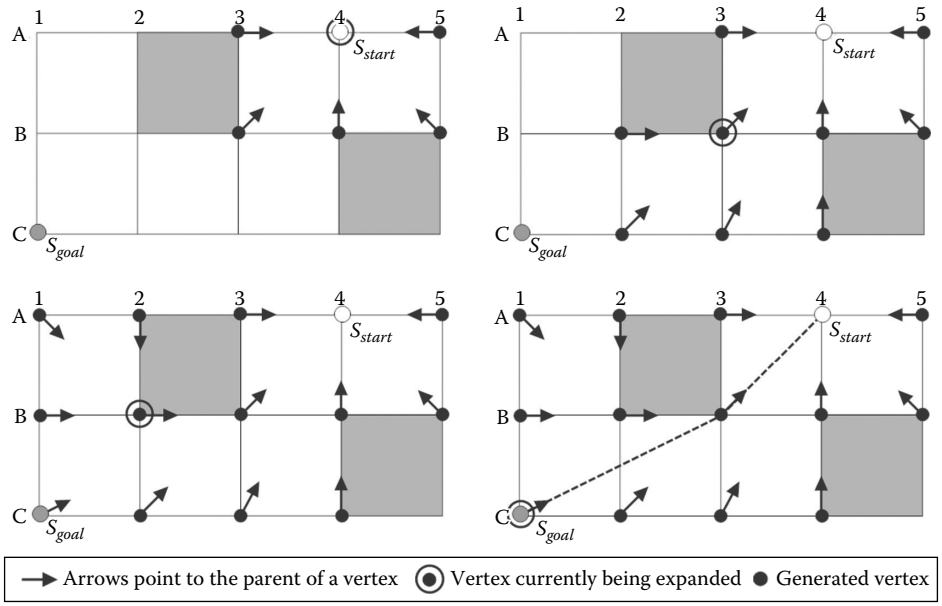



Figure 16.7

Example trace of Theta*.

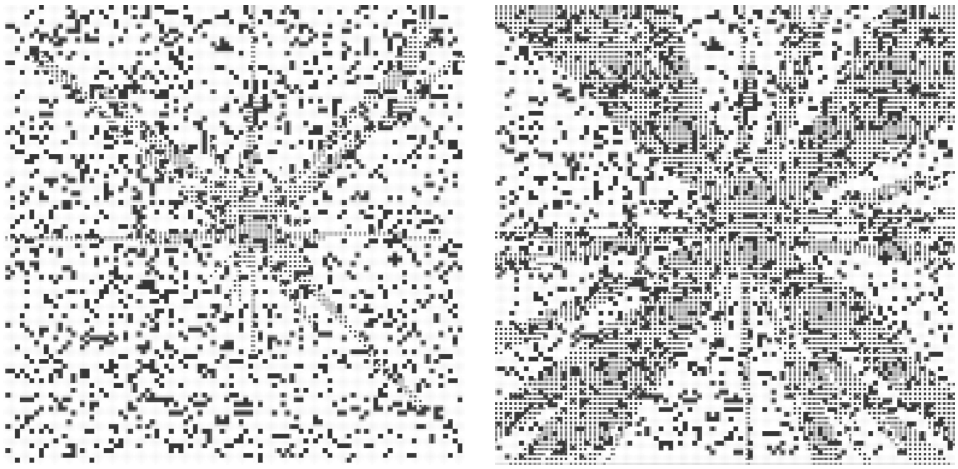
## 16.5 Theta* Paths

While Theta* is not guaranteed to find shortest paths in the continuous environment (for the reasons explained in the work of Nash et al. [Nash 07]), it finds shortest paths in the continuous environment quite frequently, as demonstrated in Figure 16.8. We performed a search from the center of the grid to all of the vertices in the grid. In Figure 16.8a, A* PS found the shortest path in the continuous environment from the center of the grid to each shaded dot. Similarly, in Figure 16.8b, Theta* found the shortest path in the continuous environment from the center of the grid to each shaded dot. There are far more shaded dots in Figure 16.8b than Figure 16.8a.

Figure 16.9 compares a Theta* path (bottom) and an A* path (top) on a game map from BioWare's popular RPG *Baldur's Gate II*, which has been discretized into a 100 × 100 eight-neighbor square grid. The Theta* path is significantly shorter and appears more realistic than the A* path. Furthermore, most postprocessing techniques are unable to shorten the A* path into the Theta* path since the A* path circumnavigates blocked cells in a different way.

## 16.6 Analysis

Researchers have performed experiments comparing the path lengths and runtimes of A*, A* PS, and Theta* using eight-neighbor square grids that either correspond to game maps or contain a given percentage of randomly blocked cells [Nash 12, Yap 11, Sislak 09]. The relationships between the lengths of the paths found by A*, A* PS, and Theta* are relatively consistent. The Theta* paths are approximately 4% shorter than the A* paths. The A* PS paths are approximately 1%–3% shorter than the A* paths depending on the type of environment and the heuristic (e.g., straight-line distance or octile



(a) (b)

Figure 16.8

Shortest paths found by A* PS (a) and Theta* (b).

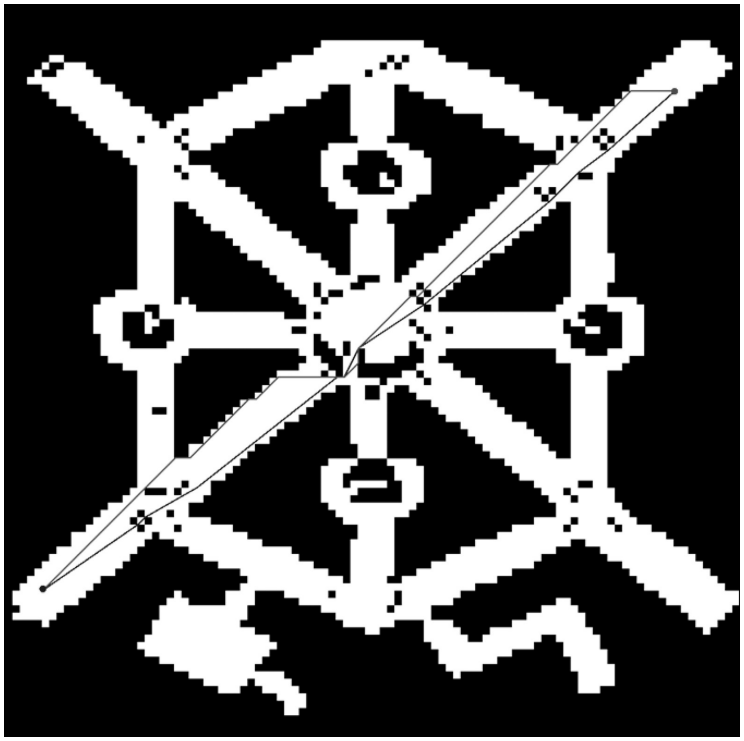


Figure 16.9

Theta* path (bottom path) versus A* path (top path).

distance heuristic). The shortest paths in the continuous environment are approximately 0.1% shorter than the Theta* paths.

The relationships between the runtimes of A*, A* PS, and Theta* are less consistent. This is because the results vary significantly with different experimental setups (such as grid size, placement of blocked cells, locations of start and goal vertices, *h*-values, and tiebreaking scheme for selecting a vertex among those with the smallest key in the open list). Currently, there is no agreement on standard experimental setups in the literature. We therefore broadly average over all reported results to give the reader an approximate idea of the efficiency of Theta*. Theta* is approximately two to three times slower than A* with the octile distance heuristic and approximately two times *faster* than A* PS with the straight-line distance heuristic. The significant difference in runtime between A* with the octile distance heuristic and A* PS with the straight-line distance heuristic is due to a large difference in the number of vertex expansions. However, highly informed heuristics, such as the octile distance heuristic on grids, do not exist for many discretizations of continuous environments, such as navigation meshes. If A*, A* PS, and Theta* all use the straight-line distance heuristic, then Theta* might find shorter paths faster than A*, and A* PS.

In general, Theta* provides a good trade-off between path length and runtime. On the one hand, Theta* is orders of magnitude faster than standard implementations of

A* on visibility graphs and finds paths that have nearly the same length. On the other hand, Theta* is approximately two to three times slower than versions of A* on eight-neighbor square grids that are optimized for performance but finds much shorter and more realistic-looking paths. The efficiency of Theta* is dependent on efficient line-of-sight checks. For applications in which line-of-sight checks are slow and thus a bottleneck, we suggest taking a look at Lazy Theta* [Nash 10] and Optimized Lazy Theta* [Nash 12]. These are optimized versions of Theta* that are easy to understand, quick to implement, and often provide an even better trade-off between path length and runtime.

## 16.7 Conclusion

We hope that this chapter serves to highlight the usefulness of any-angle pathfinding for efficiently finding short and realistic-looking paths. For more information on Theta*, we suggest visiting our any-angle pathfinding web page [Koenig 14] and reading the publications that this article is derived from [Daniel 10, Nash 07, Nash 12, Nash 13]. If you are interested in Theta*, you may also like other any-angle pathfinding algorithms such as Field D* [Ferguson 06], Accelerated A* [Sislak 09], Block A* [Yap 11], and Anya [Harabour 13].

## Acknowledgment

The research at USC was supported by NSF under grant numbers 1409987 and 1319966. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

## References

[Björnsson 03] Björnsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., and Yap, P. 2003. Comparison of different grid abstractions for pathfinding on maps. *Proceedings of the International Joint Conference on Artificial Intelligence*, 1511–1512.

[Choset 04] Choset, H., Hutchinson, S., Kantor, G., Burgard, W., Lydia, K., and Thrun, S. 2004. *Principles of Robot Motion.* MIT Press, Cambridge, MA.

[Daniel 10] Daniel, K., Nash, A., and Koenig, S. 2010. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39, 533–579.

[Ferguson 06] Ferguson, D. and Stentz, A. 2006. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2), 79–101.

[Harabor 13] Harabor, D. and Grastien, A. 2013. An optimal any-angle pathfinding algorithm. *Proceedings of the International Conference on Automated Planning and Scheduling*, Rome, Italy, 308–344.

[Koenig 14] Koenig, S. 2014. Project "Any-angle path planning". Retrieved from http://idm-lab.org/project-o.html (accessed September 10, 2014).

[Lorzano-Perez 79] Lozano-Perez, T. and Wesley, M. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of the ACM*, 22, 560–570.

[Nash 07] Nash, A., Daniel, K., Koenig, S., and Felner, A. 2007. Theta*: Any-angle path planning on grids. *Proceedings of the AAAI Conference on Artificial Intelligence*, 1177–1183.

[Nash 10] Nash, A., Koenig, S., and Tovey, C. 2010. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 147–154.

[Nash 12] Nash, A. 2012. Any-angle path planning. PhD dissertation, University of Southern California, Los Angeles, CA.

[Nash 13] Nash, A. and Koenig, S. 2013. Any-angle path planning. *Artificial Intelligence Magazine*, 34(3), 85–107.

[Nilsson 68] Nilsson, N., Hart, P., and Raphael, B. 1968. Formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.

[Patel 00] Patel, A. 2000. Amit's game programming information. Retrieved from http://www-cs-students.stanford.edu/~amitp/gameprog.html (accessed September 10, 2014).

[Rabin 00] Rabin, S. 2000. A* aesthetic optimizations. In *Game Programming Gems*, ed. DeLoura, M., 264–271. Charles River Media, Hingham, MA.

[Sislak 09] Sislak, D., Volf, P., Pechoucek, M., Suri, N., Nicholson, D., and Woodhouse, D. 2009. Accelerated A* path planning. *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 375–378.

[Tozour 04] Tozour, P. 2004. Search space representations. In *AI Game Programming Wisdom 2*, ed. Rabin, S., 85–102. Charles River Media, Hingham, MA.

[Yap 11] Yap, P., Burch, N., Holte, R., and Schaeffer, J. 2011. Block A*: Database-driven search with applications in any-angle path-planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 120–125.