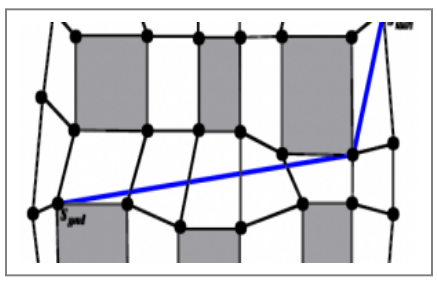
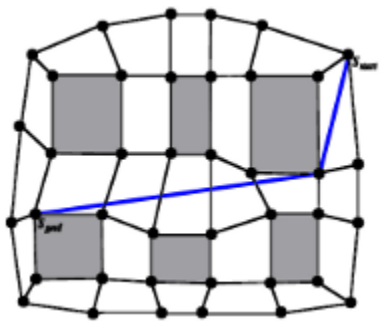
OPEN TUTORIAL



Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments

Alex Nash on September 8, 2010

This article was written by **Alex Nash**, a Ph.D. candidate at the University of Southern California, specializing in path planning for video games and robotics. You can contact him by email <anash at usc.edu>.



One of the central problems in game AI is finding short and realistic looking paths. Path planning is typically divided into two parts: *discretize* simplifies a continuous environment into a graph and *search* propagates information along this graph to find path from a given start location to a given goal location. Video game developers (and roboticists) have developed several methods for solving the *discretize* problem: two dimensional regular grids composed of squares (square grid), hexagons or triangles, three dimensional regular grids composed of cubes, visibility graphs, circle based waypoint graphs, space filling volumes, navigation meshes, hierarchical methods such as framed quad trees, probabilistic road maps (PRMs) and rapidly exploring random trees (RRTs) [2], [3], [4].

However, due to its simplicity and optimality guarantees, A* is almost always the *search* method of choice. This is because A* is guaranteed to find a shortest path on the graph. The problem with A* is that a shortest path on the graph is *not* equivalent to a shortest path in the continuous environment. A* propagates information on the graph *and* constrains paths to be formed by edges of the graph which artificially constrains path headings. Consider Figures 1 and 2, in which a continuous environment has been discretized into a square grid and a navigation mesh, respectively. The shortest path on both the square grid and the navigation mesh (Figures 1 and 2 left) is much longer and more unrealistic looking (i.e has a heading change in free space) than the shortest path in the continuous environment (Figures 1 and 2 right).

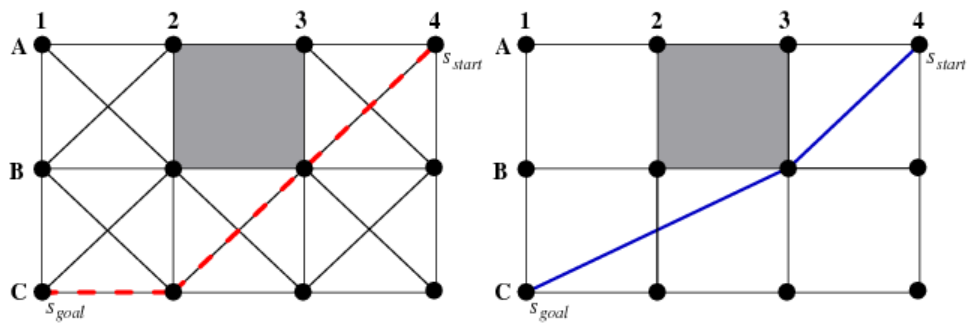


Figure 1: Square Grid: Grid Path versus Shortest Path

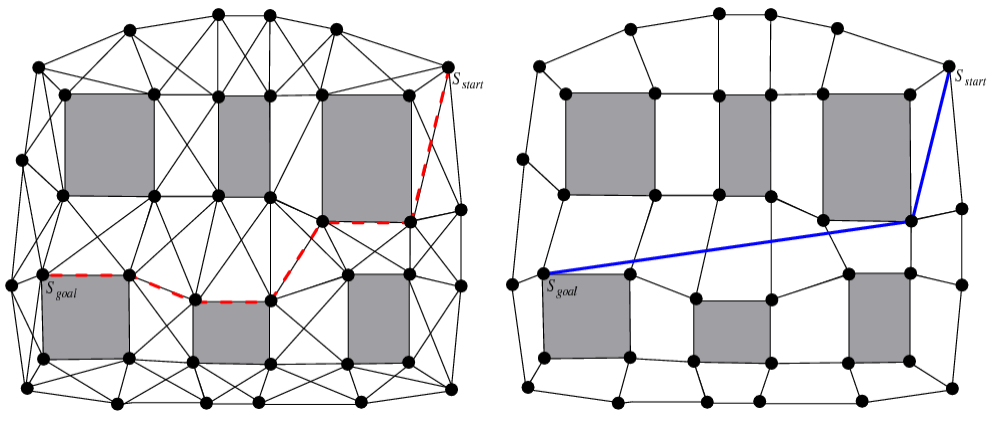


Figure 2: Navigation Mesh: Grid Path versus Shortest Path (adapted from [10])

The fact that A* searches find long and unrealistic looking paths is well known in the video game community [5]. The typical solution to this problem is to apply a post processing technique to smooth the paths found by an A* search. However, choosing the post processing technique that efficiently finds a realistic looking path can be difficult. One reason for this is that an A* search (with certain types of heuristics) is only guaranteed to find one of many shortest paths on the graph, some of which can be smoothed more efficiently than others. For example A* with the octile heuristic finds paths very efficiently on square grids, but it finds paths which are both very difficult to smooth and very unrealistic looking (a bad combination) because it tends to find paths in which all diagonal moves appear before all unit length moves. This is depicted by the dashed red path in Figure 3.

On the other hand A* with the straight-line heuristic on square grids finds paths that are the same length as A* with the octile heuristic, albeit significantly slower due to an increase in the number of vertex expansions, however these paths are smoothed very effectively because the paths it finds tend to follow the shortest path. This is depicted by the dotted green path in Figure 3. In general, these aesthetic optimizations do improve paths, but they provide difficult tradeoffs and they do not address the fundamental issue; namely that the A* search only considers paths that are constrained to graph edges. Therefore these post processing techniques are often not effective [1] [6] because they have no knowledge of the post processed paths. Thus, there is a need for smarter solutions to the *search* problem.

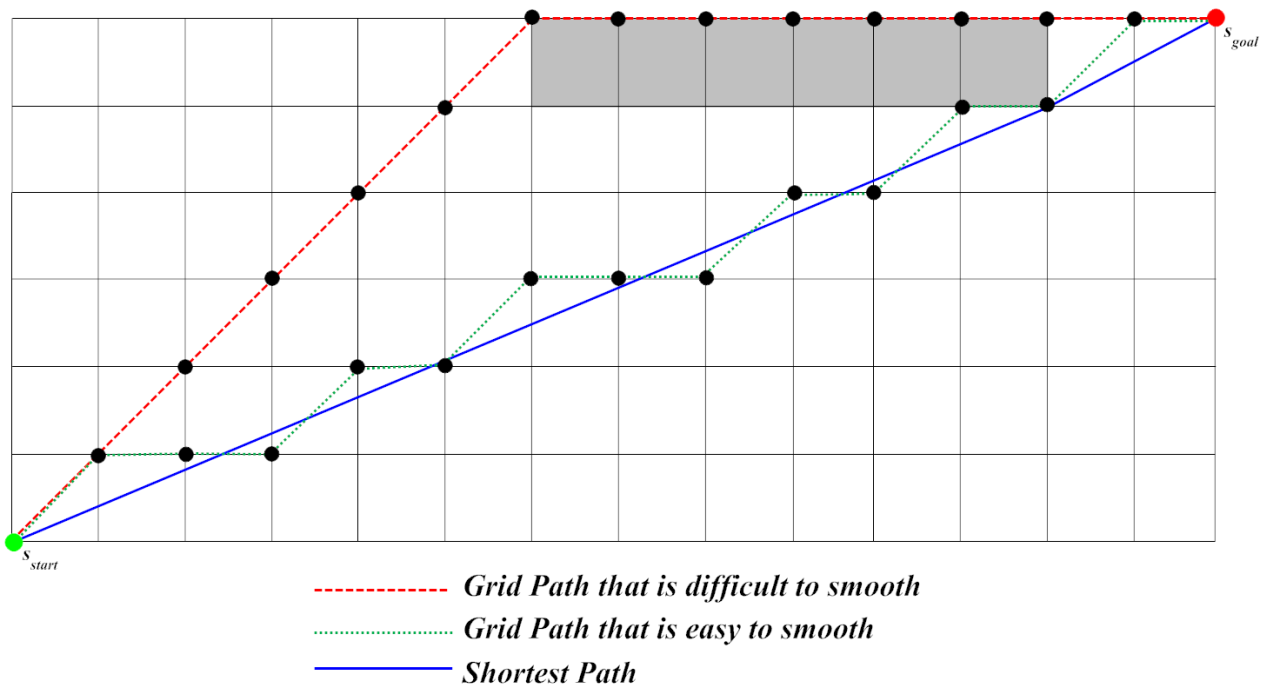


Figure 3: Paths found by A* with different Post Smoothing techniques(A* PS)

A* Aesthetic Optimizations

This was the topic of a recent paper [1] that I co-wrote with Sven Koenig, Kenny Daniel and Ariel Felner and which was presented at AAAI'07. In the paper we presented a new solution to the *search* problem, called Theta*. Theta* is a variant of A* that propagates information along graph edges (to achieve a short runtime) without constraining the paths to graph edges (to find "any-angle" paths). Theta* is simple (to understand and to implement), fast and finds short and realistic looking paths. The pseudo code for Theta* has only four more lines than the pseudo code for A* and has a similar runtime, but finds paths that are nearly identical to the shortest paths (without post processing the paths found by Theta*!).

For simplicity, this article will focus on square grids in which a two-dimensional continuous environment is discretized into square cells that are either blocked (grey) or unblocked (white). Furthermore, we map vertices to the corners of grid cells as opposed to the centers of unblocked cells. Neither of these two assumptions is required for Theta* to function correctly. Our goal is to find a short and realistic looking path from the start location to the goal location (both at the corners of cells) that does not pass through blocked cells, as shown in Figure 1. Theta* was motivated by combining the desirable properties of two existing path planning techniques.

- **Visibility Graphs:** Visibility graphs contain the start vertex, the goal vertex and the corners of all blocked cells [7]. 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. Shortest paths on visibility graphs are also

Content Index

shortest paths in the continuous terrain, as shown in Figure 1 (right). However, path planning is slow on large visibility graphs since the number of edges can be quadratic in the number of cells.

- **Grids:** Path planning is faster on grids than visibility graphs since the number of edges is linear in the number of cells. However, paths formed by grid edges can be sub-optimal and unrealistic looking since the path headings are artificially constrained [8], as shown in Figure 1 (left)

We assume an eight-neighbor grid throughout this article, where *V* is the set of all grid vertices, s_{start} *in V* is the start vertex of the search, and s_{goal} *in V* 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. $nghbr_{vis}(s)$ *in V* is the set of neighbors of vertex *s in V* that have line-of-sight to *s*.

A* Algorithm

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start) := s_start;
5      open.Insert(s_start, g(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' ∈ nghbr_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 UpdateVertex(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 4: Pseudo Code for A*

Theta* builds upon A* [9](Figure 4)[*] and it's worth quickly introducing it here. A* uses *h*-values *h(s)* that approximate the goal distances of the vertices *s in V* to focus its search. A* maintains two values for every vertex *s*: **(1)** The *g*-value *g(s)* is the length of the shortest path from the start vertex to *s* found so far. **(2)** The parent *parent(s)* is used to extract the path after the search halts. Path extraction is done by repeatedly following the parent pointers from the goal vertex to the start vertex. A* also maintains two global data structures: **(1)** The open list is a priority queue that contains the vertices to be considered for expansion. **(2)** The closed list 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')*. It updates the *g*-value and parent of *s'* if this new path is shorter than the shortest path from the start vertex to *s'* found so far [= *g(s')*].

As noted in [5], the paths found by A* often look like they were constructed by someone who was drunk. This is both because the paths are longer than the shortest paths and because the path headings are artificially constrained by the grid. The most common technique for addressing this problem is to post process the paths found by A*. One such technique is to pull the paths tight, such that they look like "rubber bands" around obstacles. Post processing techniques shorten the paths found by A*, but are cosmetic in the sense that A* *only* considers paths that are constrained to grid edges during the search and thus the search cannot make informed decisions. This follows from the fact that the topology of the path is determined by the A* search, which has no knowledge of the shorter post processed paths. Furthermore, as we saw in

Figure 3, choosing a post smoothing technique that efficiently finds paths with a toplogy that can be easily smoothed can be difficult. In some cases post smoothing cannot reduce the lengths of the paths constrained to grid edges at all, and the paths found by A* on grids can be up to 8% longer than the shortest path and very unrealistic looking. We therefore develop Theta*, which considers paths that are not constrained to grid edges during the search and thus can make more informed decisions during the search.

Theta* Algorithm

```
1  ComputeCost(s, s')
2      if LineofSight(parent(s), s') then
3          /* Path 2 */
4          if g(parent(s)) + c(parent(s), s') < g(s') then
5              parent(s') := parent(s);
6              g(s') := g(parent(s)) + c(parent(s), s');
7      else
8          /* Path 1 */
9          if g(s) + c(s, s') < g(s') then
10             parent(s') := s;
11             g(s') := g(s) + c(s, s');
12 end
```

Figure 5: Pseudo Code for Theta*

The key difference between Theta* (Figure 5) and A* is that Theta* allows the parent of a vertex to be any vertex, unlike A* where the parent must be a visible neighbor. Procedure Main and UpdateVertex are identical to that of Figure 4 and thus are not shown. We use the consistent straight line distances $h(s) = c(s,s_{goal})$ 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 (procedure ComputeCost):

- **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 9).
- **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 4). The idea behind considering Path 2 is that Path 2 is no longer than Path 1 due to the triangle inequality if *s'* has line-of-sight to *parent(s)*.

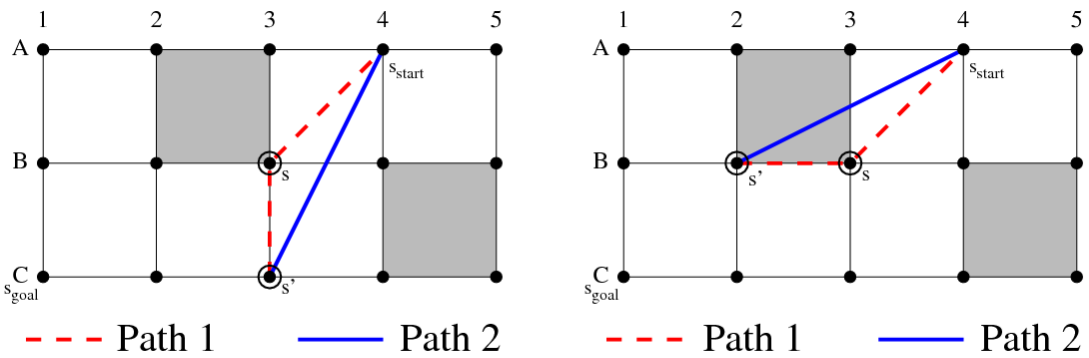
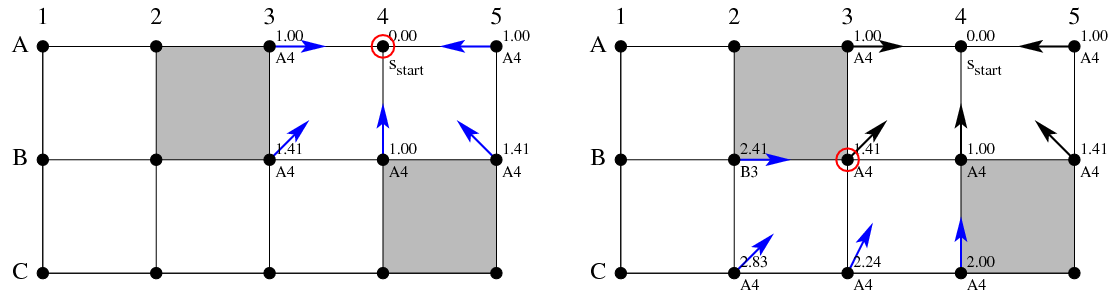


Figure 6: Theta* updates a vertex according to Path 1 (left) and Path 2 (right)

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 6 where *B3* (with parent *A4*) gets expanded. *B2* is an unexpanded visible neighbor of *B3* which does not have line-of-sight to *A4* and thus is updated according to Path 1 (right). *C3* is an unexpanded visible neighbor of *B3* which does have line-of-sight to *A4* and thus is updated according to Path 2 (left).



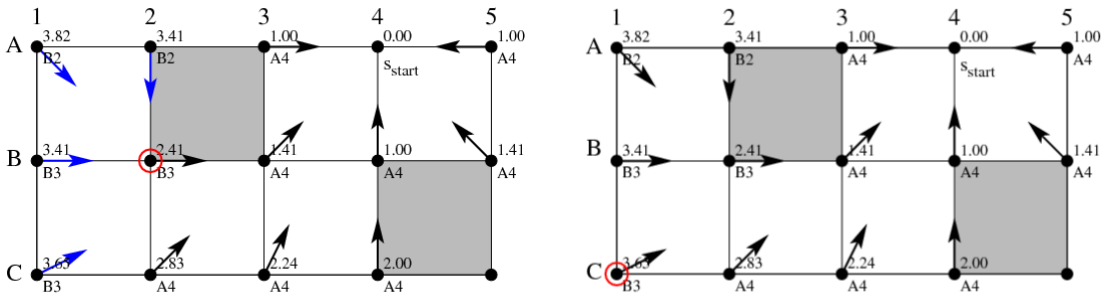


Figure 7: Example Trace of Theta*

Figure 7 shows a complete trace of Theta*. Each vertex is labeled with its *g*-value and an arrow pointing to its parent vertex. The hollow circle indicates which vertex is currently being expanded. The start vertex *A4* is expanded first, followed by *B3*, *B2* and *C1*.

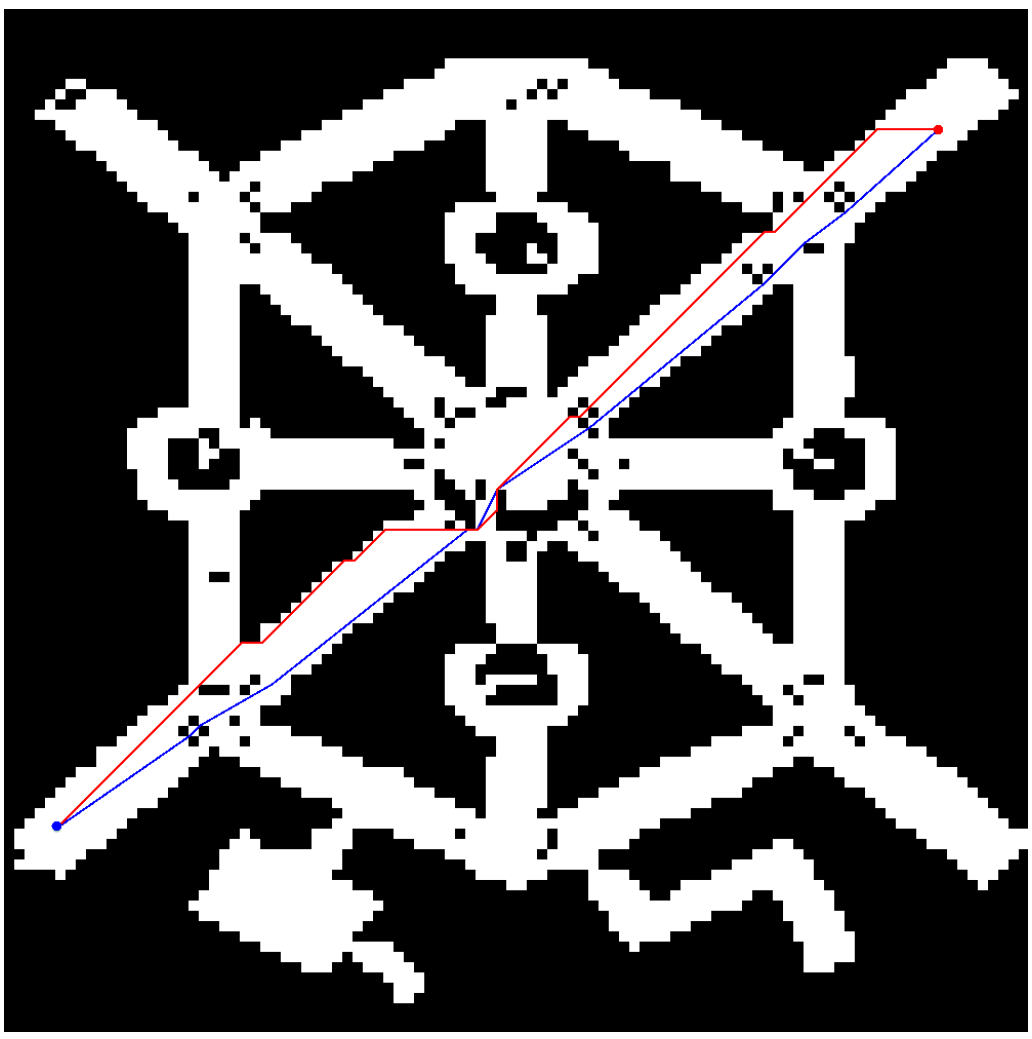


Figure 8: Theta* Path (blue) versus A* Path (red)

Figures 8 compares the path found by Theta* (blue) and the path found by A* (red) on a game map from Bioware's popular RPG Baldurs Gate, which has been discretized into a 100*100 square grid. The Theta* path is significantly shorter and more realistic looking than the path found by A*. Furthermore, notice that a post processing step would not be able to smooth the A* path into the Theta* path because the topologies of the two paths are different.

Line-of-Sight

Line-of-sight checks can be performed very efficiently with only integer operations on square grids. The reason for this is that performing a line-of-sight check is similar to determining which points to plot on a raster display when drawing a straight line between two points. The plotted points correspond to the cells that the straight line passes through. Thus, two vertices have line-of-sight if and only if none of the plotted points correspond to blocked cells. This allows Theta* to perform its line-of-sight checks with a standard line-drawing method from computer graphics [11] that uses only fast logical and integer operations rather than floating-point operations, as shown in Figure 9 where *grid* represents the grid and *grid[x,y]* is true if and only if the corresponding cell is blocked. For simplicity, the pseudo code in Figure 9 returns true for paths that pass between diagonally touching blocked cells. However, Theta* is complete and correct whether or not paths are allowed to pass between diagonally touching blocked cells.

```
LineOfSight(s, s')
    x_0 := s.x;
    y_0 := s.y;
    x_1 := s'.x;
    y_1 := s'.y;
    d_y := y_1 − y_0;
    d_x := x_1 − x_0;
    f := 0;
    if d_y < 0 then
        d_y := −d_y;
        s_y := −1;
    else
        s_y := 1;
    if d_x < 0 then
        d_x := −d_x;
        s_x := −1;
    else
        s_x := 1;
    if d_x ≥ d_y then
        while x_0 ≠ x_1 do
            f := f + d_y;
            if f ≥ d_x then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                y_0 := y_0 + s_y;
                f := f − d_x;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            if d_y = 0 AND grid[x_0 + ((s_x − 1)/2), y_0] AND grid[x_0 + ((s_x − 1)/2), y_0 − 1] then
                return false;
            x_0 := x_0 + s_x;
    else
        while y_0 ≠ y_1 do
            f := f + d_x;
            if f ≥ d_y then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                x_0 := x_0 + s_x;
                f := f − d_y;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            if d_x = 0 AND grid[x_0, y_0 + ((s_y − 1)/2)] AND grid[x_0 − 1, y_0 + ((s_y − 1)/2)] then
                return false;
            y_0 := y_0 + s_y;
    return true;
end
```

Figure 9: Pseudo Code for Line-of-Sight checks on a square grid

Analysis

While Theta* is not guaranteed to find shortest paths in the continuous environment (for reasons explained in [1]), it *does* find shortest paths a large percentage of the time. For example, in Figure 10, Theta* found the shortest path in the continuous environment from the large red dot to each blue dot. We performed an extensive analysis using both small and large square grids from Bioware's popular RPG Baldur's Gate (game maps) and square grids with given percentages of randomly blocked cells (random maps).

We found that on average the ratio of the lengths of the paths found by Theta* and the shortest paths was 1.007 on game maps and 1.002 on random maps. This is significantly better than the 1.04 ratio for paths found by A* (remember that A* paths can be up to 8% longer than the shortest paths). Theta* was only approximately a factor of 2 slower than a version of A* that was optimized for performance using the octile distance heuristic (to significantly reduce the number of expansions and thus the runtime). However, when you incorporate the post smoothing technique the runtimes can far more favorable for Theta*. While Theta* was slower than A* PS with the octile heuristic, it was significantly (approximately a factor of 2) faster than A* PS with the straight-line heuristic. Furthermore it is worth noting that on NavMeshes highly optimized heurisitcs such as the octile heuristic do not exist. Theta* is orders of magnitude faster than standard implementations of A* searches on visibility graphs[*].

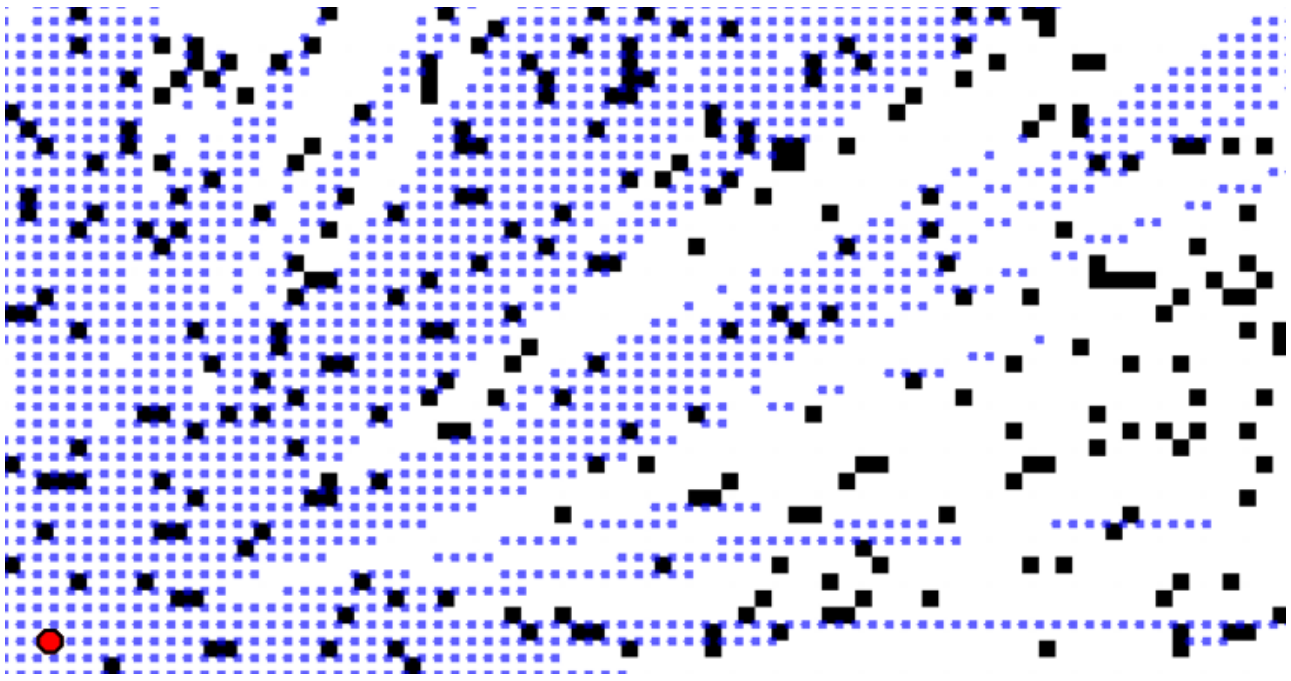


Figure 10: Shortest paths found by Theta*

Concluding Remarks

I hope this article will serve to highlight the usefulness of any-angle path planning methods for efficiently finding short realistic looking paths. For more information on Theta* I suggest taking a look at the original paper [1] or visiting our any-angle path planning web page. If you like Theta*, you may also like Field D* [6], which uses linear interpolation to find any-angle paths, but is restricted to regular grids composed of squares.

If you have specific questions or comments regarding anything described in this article, please feel free to contact me.

Footnotes

[*] *open.Insert(s,x)* inserts vertex *s* with key *x* into *open*. *open.Remove(s)* removes vertex *s* from *open*. *open.Pop()* removes a vertex with the smallest key from *open* and returns it.

[*] Visibility graphs have been studied extensively in robotics. Thus, many optimizations exist such as reduced visibility graphs, plane sweeps and nearest neighbor tricks, which can significantly reduce runtimes, especially in sparse outdoor environments. However, for cluttered indoor environments, the quadratic worst case complexity of visibility graphs significantly degrades their performance. Furthermore, these optimizations are far more complex than A* or Theta*.

References

Theta*: Any-Angle Path Planning on Grids
A. Nash, K. Daniel, S. Koenig and A. Felner
(2007) Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1177--1183
[1] Download **(PDF)**

Comparison of Different Grid Abstractions for Pathfinding on Maps
Y. Bjornsson, M. Enzenberger, R. Holte, J. Schaeffer and P. Yap
(2003) Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1511-1512
[2] Download **(PDF)**

Principles of Robot Motion
[3] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun
(2004) MIT Press
(google books entry)

AI Game Programming Wisdom 2: Search Space Representations
[4] P. Tozour
(2004) Charles River Media, pp. 85-102
Purchase

Game Programming Gems: A* Aesthetic Optimizations
[5] S. Rabin
(2000) Charles River Media, pp. 264-271
Purchase

Using Interpolation to Improve Path Planning: The Field D* Algorithm
[6] D. Ferguson and A. Stentz
(2006) Journal of Field Robotics, Volume 23, Issue 2, pp. 79-101
Download **FILE**

An Algorithm for Planning Collision-free Paths among Polyhedral Obstacles
[7] T. Lozano-Perez and M. Wesley
(1979) Communication of the ACM, Volume 22, pp. 560-570
Download **FILE**

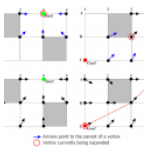
Related Articles

Grid-Based Path-Finding
[8] P. Yap
(2002) Proceedings of the Canadian Conference on Artificial Intelligence, pp. 44-55
Download **FILE**

Formal Basis for the Heuristic Determination of Minimum Cost Paths
[9] P.E. Hart, N.J Nilsson, B. Raphael
(1968) IEEE Transactions on Systems Science and Cybernetics, Volume 4, Issue 2, pp. 100-107
Download **FILE**

Amit's Game Programming Information
[10] (2000) A. Patel **Online**

Algorithm for Computer Control of a Digital Plotter
[11] J. Bresenham
(1965) IBM Systems Journal, Volume 4, Issue 1, pp. 25-30
Download **FILE**



Lazy Theta*: Faster Any-Angle Path Planning

Which is the Yellow Brick Path:

Pathfinding in Static and Dynamic Environments and the Future of Multi-

Discussion 4 Comments

alexjc on September 9th, 2010

Thanks Alex, great article. [B]I'm hoping this stimulates a healthy discussion about path-smoothing techniques and our solutions to them![/B] It took Alex and I many months to refine this article, particularly its motivation and conclusion -- a credit to Alex's dedication there :-) The reason is that it's hard to make a solid argument against replacing the traditional path-smoothing in games, for the following reasons in particular: [LIST] [*]Path-smoothing may be sub-optimal in the world itself, but why does it matter? Sub-optimal paths are fine as long as they are realistic. [*]Hierarchical pathfinding means that smoothing happens locally and incrementally. It may be costlier, but it helps create good enough results. [*]Increasingly, navigation use local steering or animation-driven motion that may not necessarily need fully string-pulled paths, just a corridor? [/LIST] That said, there's a solid case to be made for Theta* style algorithms that address the issue at search-time: [LIST=1] [*]Path smoothing and string-pulling is most often a hack and there are so many ways to do it... and do it wrong (e.g. path corners in the middle of a room). [*]Seems like Theta* changes to A* are self-contained and relatively elegant, so it could even result in less work for someone experienced in both. [/LIST] The only big downside is performance. If Theta* wins on grids under some conditions, that's a good start. It'll be interesting to see if this scales to navigation meshes... Alex

phyzen on September 15th, 2010

How do you handle cases where you deliberately pathfind around an area like a mine field? With a normal A* you are (almost) forced to have a path smoothing and that could honour the chosen triangles. It seems that Theta* would just smooth the path straight over it. The solution I can think of would be to treat a node as un-traversable (fail the test for a new parent) if it had a "worse" value than the one you are testing (could use some threshold, and of course the original parent is always valid). This would prevent the algorithm from smoothing over the mine field, but work on the normal open ground as normal. It should also work if you start in the middle of the mine field as you the other nodes would be as bad, but not "worse". Any thoughts on this?

dutchmark on November 10th, 2010

I think this is a very interesting idea. It certainly simplifies things over 'post-smoothing' so to speak. I find it elegant. So far I still do post smoothing though. I'm a little afraid the line-of-sight needs to be called very often. Also, all the work needs to be done ahead of time. I only smoothen the path until the first obstacle. Then it can do the rest when it gets there, which balances the work-load over time. But it's nice to know this technique, it will come in handy some day.

demilp on August 13th, 2014

On figure 7, C1 is being expanded from B1, right? Because if it would be expanded from C2, its parent should be C2 itself as it doesn't have a line of sight to C2's parent (A4).

less than a minute!