27

Tactical Pathfinding on a NavMesh

Daniel Brewer

- 27.1 Introduction
- 27.2 Other Methods
- 27.3 Tactical Pathfinding Method
- 27.4 Extending the Technique to 3D27.5 Conclusion

27.1 Introduction

Traditional pathfinding has been focused on finding the *shortest* route from A to B. However, as gamers demand more realism, this is no longer sufficient. Agents should instead find the most *appropriate* route from A to B. In action-shooter or strategy games, this usually means the most tactically sound route—the route that provides the most concealment from the enemy and avoids friendly lines of fire, rather than the shortest, most direct path.

Many tactical pathfinding solutions require a regular waypoint grid and numerous line-of-sight raycast checks to determine the safer, more concealed route between two points. However, regular waypoint grids are known for poor memory efficiency and the large number of waypoints to check increases the run-time pathfinding load, especially with the numerous visibility checks required for tactical pathfinding.

Navigation meshes (NavMeshes) are an alternative approach to grids and have become a widely used, efficient representation of navigable space. This article will present a method of cover representation and modification to the A* algorithm to perform tactical pathfinding directly on a NavMesh.

27.2 Other Methods

At its core, tactical A* pathfinding can be achieved by modifying the cost of nodes in your navigation graph [van der Sterren 02]. A node that is visible or exposed to the enemy should have a higher cost, while a node that is concealed from the enemy should have a lower cost. This way, the A* algorithm will favor the lower cost, more concealed nodes over the high cost, exposed nodes. An agent following such a path will seem more cautious, preferring to keep out of line-of-sight of his enemy as much as possible.

The common practice in tactical pathfinding is to use a grid or a regular waypoint graph. The high resolution and regular spacing of nodes allows the A* algorithm to better take into account the differing costs for exposed versus concealed nodes over the distance traveled. The accuracy of these techniques depends on the density of the grid; the tighter the grid spacing, the greater the number of nodes and the better the accuracy of the paths generated. A high grid density has significant costs, not only in memory for storing all the nodes, but also in processing at run-time when the A* algorithm has many more nodes to compute.

There are a number of possible methods to determine how exposed a particular node is to an enemy threat. The crucial question is "Can an enemy standing at position A see me at position B?" Performing visibility raycast checks at run-time during pathfinding can drastically hamper performance. Exhaustive visibility calculations can be performed offline and stored for use in game via look-up tables [Lidén 02]. As the environment increases in size and the number of nodes grows, this $O(n^2)$ approach requires exponentially larger amounts of memory.

An alternative to relying on exact line-of-sight raycast checks is to use approximations. A possible approximation is to store a radial distance field for each node [Straatman 05]. This approach can be extended to 3D with depth-buffer cube-maps [van der Leeuw 09]. This allows a quick look-up to approximate how far an agent can see in a particular direction from a specified point in the world. By comparing the distances in each direction between two points, it is possible to determine if these points are visible to each other. Another form of approximation is to rasterize cover objects into a grid [Jurney 07]. The enemy's view cone can then be rasterized into the grid to add a weighting factor against waypoints that are exposed to his line-of-sight. This rasterizing approach can be used to deal with dynamic, destructible cover, too. After an object is destroyed, simply perform the rasterization again with the remaining objects, and the navigation grid will reflect the current run-time changes.

These approximations actually enhance the robustness of the solution against small movements of the enemy threat. In an environment filled with complex geometry it is possible for a raycast to pass through a tiny gap in an object and so report a clear line-of-sight, when in fact the location would be concealed. The opposite can also occur, reporting an obscured line-of-sight when the location would in fact be visible. Even with simple geometry, a small adjustment of the raycast origin could return different results. The low resolution of the distance fields, or rasterized cover grids, makes these approximations much less susceptible to these slight variations in position.

Navigation meshes are a popular, efficient method of representing navigable regions of a virtual environment [Tozour 04]; however, it is not as straightforward to perform tactical pathfinding on these navigation representations. The irregular spacing and large area covered by the navigation polygons results in poor overall visibility accuracy from raycast tests. To overcome this drawback, it is possible to dynamically calculate a regular waypoint graph at run-time by sampling positions from a static NavMesh [Bamford 12]. This tactical graph need only be generated locally for the areas of the world where combat encounters take place. However, this does lead to a duplication of navigation information and at worst case may even require a separate, independent pathfinding code for the two different representations.

The following section presents an approach to tactical pathfinding that can work directly on a NavMesh without requiring the duplication of navigation information.

27.3 Tactical Pathfinding Method

The technique is split up into three parts. The first part will deal with ways to partition and annotate the NavMesh. The second part will deal with cover representation, and the final part will deal with calculating the cost of navigation polygon traversal in A* in order to calculate a more appropriate route.

27.3.1 Tessellating and Annotating the NavMesh

There has been a lot of work on creating an optimal NavMesh representation of a virtual environment [Tozour 02, Farnstrom 06], though this is not necessarily required for the purposes of tactical pathfinding. The NavMesh can be more finely tessellated to provide increased detail in areas of different movement properties or tactical significance.

Polygons in which it is more difficult, or more tactically unsafe, to move can be flagged with additional costs, just as in a regular grid representation. Areas of water could be flagged as "slow movement" or "only passable by amphibious units." Regions of high grass could be flagged with "provides concealment." Dense undergrowth could be flagged as "slow movement" and "provides concealment." The center of a courtyard with overlooking balconies can be flagged as "vulnerable" or "unsafe" so that agents will move through the area while staying near the edges. These flags and costs modify the traversal costs for the pathfinding algorithm to get agents to follow more appropriate paths rather than simply the shortest route.

An optimal NavMesh will be made up of large, convex polygons in order to provide the most coverage of the navigable space with the least number of polygons. The aforementioned courtyard could be represented by a single polygon as shown in Figure 27.1. This would prevent the differentiation between the exposed center and the covered edges under the balconies. Additional polygons are required in order to represent the tactical difference between these areas.

A quick and easy way to achieve these benefits is to provide tools allowing level designers to mark-up areas of the NavMesh with this extra detail. The designers need to be able to specify the contour boundaries of the marked up areas, and then these boundaries need to be kept fixed in the polygonal tessellation of the map. It is possible to use terrain analysis algorithms during NavMesh generation to automate some of the tedium of manual mark-ups, but this is beyond the scope of this article. In most cases, this manual mark-up may only be necessary in very specific game-play instances and the general cover approach below will suffice for most tactical pathing needs during combat.



Optimal NavMesh for a courtyard with an overhanging balcony (a). The same courtyard is shown with an exposed designer region flagged as "unsafe" (b).

27.3.2 Cover Representation

Cover spots in game levels can be represented as discrete points. This is often the case with waypoint graph navigation representations. Connected areas of cover are represented by links between these cover waypoints. It can be beneficial to instead consider cover as connected, linear segments rather than discrete points. These line segments follow the contour of the cover-providing obstacle and a height property can differentiate between tall, standing cover and waist-high, crouching cover. This annotated line segment is called a "cover segment" and provides a simple polygon approximation of the line-of-sight blocking obstacle.

It is important to quickly be able to look up which pieces of cover can provide concealment to an avatar at a given location in the world. Each polygon in the NavMesh can store a small table with the indices to the most significant cover segments affecting that polygon. This table of cover segments per polygon is called the Cover Map (CoverMap). This allows a quick look-up of the objects that could provide cover or block line-of-sight to a character on a particular polygon in the NavMesh (Figure 27.2).

This data is only an approximation of the line-of-sight blockers, since only the most relevant cover for each NavMesh polygon is stored. A heuristic for selecting cover segments for a navigation polygon should take into account the direction of cover provided, the distance of the cover segment from the polygon, and whether that cover is already occluded by other cover segments. For the purposes of allowing agents to make better tactical choices during combat, this approximation is sufficient. It is possible to improve the accuracy by increasing the number of cover segments stored per polygon and by tweaking the selection criteria.

27.3.3 Pathfinding

When searching for the shortest path, the primary concern is distance or movement time. Tactical pathfinding, on the other hand, is concerned more with finding a safe path. This can be accomplished by taking a standard pathfinding algorithm and biasing the node



POIY I	А, D, E, П, I, J, IV
Poly 2	F, J, K, L, O, P

Example NavMesh showing some cover segments and how they are stored in the CoverMap.

traversal cost by how exposed or concealed that node is to the enemy. By increasing the cost of exposed nodes, the pathfinding algorithm will return a route that avoids more of these exposed nodes and thus one that is more concealed. The method below will not cover how to perform A* on a NavMesh, which can be found elsewhere [Snook 00], but will focus on how to modify the A* costs to take cover into account in order to find a safer path. The core of the algorithm is first presented in a simpler, 2D form in order to clearly illustrate the concept. The next section will explain some extensions to the algorithm to operate in more complex environments.

To start finding a safe path from an enemy, the algorithm requires a list of cover segments that can potentially shield the agent from his enemy. The navigation polygon containing the enemy's position can be used to index into the CoverMap and obtain the list of cover segments. When considering a potential path segment, it is now possible to calculate how much of that path segment is concealed from the enemy.

The next step is to construct a 2D frustum from the cover segment (A–B in Figure 27.3) and the lines joining the enemy's position to each end of the cover segment (C–A and C–B). The normals of all these planes should point inwards to the leeward area behind the cover. The path segment is then clipped by this frustum. The portion of the segment left in front of all three planes is the concealed portion of the path segment.

These steps are repeated for each cover segment in the enemy's CoverMap. By comparing the total clipped length to the original segment length, the proportion of exposed to



Calculating the proportion of the path segment that is concealed from an enemy (a) by first constructing a frustum from the cover segment (b) and then clipping the path segment by this frustum (c) and repeating for each cover segment in the CoverMap (d).

concealed is calculated. This is then multiplied by the cover-bias factor and added to the base cost of the segment, as shown in Listing 27.1. The A* search will thus be biased away from exposed segments. The greater the cover-bias factor, the more it will cost the agent to move along exposed segments and this will result in him going further out of his way to find a safer, more obscured path.

Listing 27.1 also shows how designer mark-ups and agent biases can affect the pathing costs. When dealing with these, it is important to remember that biases above 1.0 will increase the cost of traversing the polygon, while biases below 1.0 decrease it and may cause the cost of traversing the node to drop below the actual straight-line distance. In general, this should be avoided as it is preferable to keep the heuristic admissible and always modify the costs to make nodes more expensive. Care should also be taken not to increase the costs too much, as if the heuristic is drastically different to the cost, the A* algorithm will explore many more nodes than necessary.

27.4 Extending the Technique to 3D

The previously presented algorithm is good for 2D or slightly undulating terrain. This is to present the core algorithm in a simple, digestible manner. However, not all environments are this simple. A character on a tall hill will be able to draw a line-of-sight over some intervening cover. Vertical structures can prove problematic as a character may not be able

Listing 27.1. Pseudocode functions to calculate the exposed portion of a path segment and the cost of traversing that segment for A^* pathfinding.

```
Line ClipLineByPlane(Line, Plane)
   returns the segment of the input Line in front of the Plane
float GetSegmentExposedLength(Line pathSegmentP1P2, Point C)
   float ConcealedLength = 0
   For each Line coverSegmentAB do
       Plane planeAB = CalcPlaneFrom3Points(A, B, A + up axis)
       Plane planeAC = CalcPlaneFrom3Points(A, C, A + up_axis)
       Plane planeCB = CalcPlaneFrom3Points(C, B, C + up axis)
       Line clippedLine = ClipLineByPlane(pathSegmentP1P2, planeAB)
       clippedLine = ClipLineByPlane(clippedLine, planeAC)
       clippedLine = ClipLineByPlane(clippedLine, planeCB)
       ConcealedLength += Length(clippedLine)
       ExposedLength = Length(pathSegmentP1P2) - ConcealedLength
       return ExposedLength
float CostForSegment(Line pathSegmentP1P2, polyFlags)
   float segmentLength = Length(pathSegmentP1P2)
   float exposedLength =
         GetSegmentExposedLength(pathSegmentP1P2, enemyPosition)
   float cost = exposedLength * agentCoverBias +
         (seqmentLength - exposedLength)
   cost += segmentLength * polyFlags.isUnsafe * agentSafetyBias
   return cost
```

to draw a clear line-of-sight through the floor or ceiling, but if the visibility approximation only considers cover segments, it will not take the floor or ceiling into account. Fortunately, the technique can be extended to work with these more complex environments.

If the terrain has large height variance, a more accurate result can be obtained by treating the cover segment as a rectangular polygon representing the length and height of the cover. A frustum can be constructed from the enemy's position and the edges of the cover polygon, as shown in Figure 27.4. This frustum can be used to clip the path segments to determine how much of the segment is obscured, just as in the planar example presented above. If the environment includes more vertical structures, it may be necessary to add extra cover planes for floors and ceilings into the CoverMap.

27.5 Conclusion

Agents that can navigate an environment in a tactically sound manner can greatly enhance a video game experience. A good approximation of cover is crucial for run-time tactical pathfinding. Using linear cover segments is an effective way of reasoning about cover and line-of-sight blocking obstacles. This representation makes it simple to calculate how much of a path segment is obscured by each cover segment. By combining this cover representation with a NavMesh, it is possible to perform fast tactical pathfinding without having to resort to high memory usage grids, regular waypoint graphs, or comprehensive visibility look-up tables.



To extend the algorithm into 3D, a full frustum needs to be created from the edges of the cover segment polygon to clip the path segment.

References

- [Bamford 12] N. Bamford. "Situational Awareness: Terrain Reasoning for Tactical Shooter A.I." AI Summit, GDC 2012. Available online (http://www.gdcvault.com/play/1015443/ Situational-Awareness-Terrain-Reasoning-for).
- [Farnstrom 06] F. Farnstrom, "Improving on near-optimality: More techniques for building navigation meshes." In AI Game Programming Wisdom 3, edited by Steve Rabin. Charles River Media, 2006, pp. 113–128.
- [Jurney 07] C. Jurney and S. Hubrick. "Dealing with Destruction: AI From the Trenches of Company of Heroes." GDC 2007. Available online (http://www.chrisjurney.com/).
- [Lidén 02] L. Lidén. "Strategic and tactical reasoning with waypoints." In AI Game Programming Wisdom, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002, pp. 211–220.
- [Snook 00] G. Snook. "Simplified 3D movement and pathfinding using navigation meshes." In *Game Programming Gems*, edited by Mark DeLoura. Charles River Media, 2000, pp. 288–304.
- [Straatman 05] R. Straatman, W. van der Sterren, and A. Beij. "Killzone's AI: Dynamic Procedural Combat Tactics." GDC 2005. Available online (http://www.cgf-ai.com/ docs/straatman_remco_killzone_ai.pdf).
- [Tozour 02] P. Tozour, "Building a near-optimal navigation mesh." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002, pp 171–185.
- [Tozour 04] P. Tozour, "Search space representations." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Charles River Media, 2004, pp. 85–102.
- [van der Leeuw 09] M. van der Leeuw. "The PlayStation 3's SPU's in the Real World—A KILLZONE 2 Case Study." GDC 2009. Available online (http://www.gdcvault.com/ play/963/The-PlayStation-3-s-SPU).
- [van der Sterren 02] W. van der Sterren. "Tactical path-finding with A*." In *Game Programming Gems 3*, edited by Dante Treglia. Hingham, MA: Charles River Media, 2002, pp. 294–306.