INTEGRATED ON- AND OFF-LINE COVER FINDING AND EXPLOITATION

Gregory H. Paull
Secret Level Inc.
123 Townsend St., Suite 300
San Francisco, CA 94107
greg@secretlevel.com

Christian J. Darken
MOVES Institute and
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
cjdarken@nps.edu

KEYWORDS

AI, First-Person Shooter (FPS), Finding Cover

ABSTRACT

Most first-person shooter game AI's are poor at quickly getting out of lines of fire. AI agents that pass up obvious opportunities to hide or take cover can ruin a game's immersiveness. We will present a system that combines the sensor grid algorithm (Darken 2004) with pathnode-based information. This system relies on cover information stored in the path nodes placed throughout the level and performs a focused run-time search in the immediate vicinity of the agent if the node based information is insufficient. This allows it to be both fast and able to react to changes in the environment.

BACKGROUND

Taking cover is a universal human response to threat. However, it is not innate; children must learn to hide. It is also not totally understood; psychologists are still investigating a critical part of hiding, which is what we know of what other people can or cannot see (Kelly et. al.). Nonetheless, nearly everyone is able to quickly and effectively duck to safety when threatened. The use of cover is also not purely defensive in nature. A person can be taught to take advantage of cover when moving to make invisible shifts in their position and to minimize their exposure to danger when shooting.

The ability to use cover effectively is one of the skills that separate the best real players in first-person shooters from the average players. Unfortunately in the current state of gaming it is also one of the ways to distinguish between live players and game agents. Game agents do not use cover effectively. Typical problems include running right by good sources of cover, and failing to consistently take the most direct route to safety.

This paper describes an approach that relies on a combination of data stored in waypoints throughout the level, and a focused dynamic (i.e. run-time) search in the immediate vicinity of the agent when the node data is insufficient. Waypoints used throughout the level for pathfinding contain information that is used by the system to help make more "intelligent" decisions regarding concealment and cover. This information includes data such as appropriate stance to assume, direction in which the cover provides protection, and types of weapons this cover provides protection from. Pre-computed visibility information is also stored in each node that greatly increases run-time performance. If an adequate waypoint is not immediately found the sensor grid algorithm is run to find a safe destination for the agent. This system is both fast and able to react to changes in the geometry of the environment that occur during play. We first describe some related techniques already in the literature. Then, we give a brief overview of the sensor grid algorithm which is described in detail in Darken et. al. 2004. Next, we describe the various types of waypoints used in the system. Finally, we describe a few extensions that could be made to the system in the future.

RELATED WORK

Previous approaches to the hiding problem involve searching a fixed set of potential hiding locations. Often this is the set of waypoints used to plot movement.

Typically, navigation is accomplished in high-resolution shooter games by the use of a set of locations we call "waypoints". An agent gets from point A to point B by moving from A to a nearby waypoint. Then the agent moves from waypoint to waypoint until a waypoint close to B is reached. The waypoint set may be selected by hand, as is typical of games based on the Unreal engine, or the set may be selected by various algorithms (Stout 2000)(Snook 2000). It was early recognized that one key to keeping the computational requirements of searching the waypoint set manageable was to keep it as small as possible (Rabin 2000). Since waypoint infrastructure is so commonly available, it seems only natural to reuse it for determining places to hide (Reece 2003)(Reece 2000)(van der Sterren 2002).

The primary advantage of waypoint-based techniques is ease of implementation and low run-time computational complexity. Unfortunately, the latter benefit is only gained when the set of waypoints is small, and when it is small, the likelihood that the best place to quickly duck out of fire is a waypoint is also small. To see why this is so, consider a map consisting of an open plane with one tall rock sitting in the center. By appropriately placing the observer and the hiding agent, one can make virtually any point the nearest place to hide! An additional difficulty, and one that will become more important in the future, is that a sparse set of potential hiding places fixed in advance is especially vulnerable to becoming invalid in dynamic environments

because of vehicle motion, destruction of buildings, and creation of new hiding places such as piles of rubble, to name a few examples. Thus waypoint-based techniques typically result in agents that can behave very counter-intuitively when searching for cover.

In military simulations, space is typically represented by a fine rectangular grid (Reece 2003) (Reece 2000) (Richbourg and Olson 1996). This avoids the difficulties caused by a sparse spatial representation as described above, but at the cost of computational complexity that may be beyond the budget of many games. The memory required to store the grid may also be an issue for very constrained computational platforms, like game consoles.

SENSOR GRID OVERVIEW

The sensor grid approach differs from its predecessors in that the set of possible hiding places is not fixed, but is instead generated dynamically at run-time. This allows it to be relatively dense close to the agent and sparse further out, while keeping the total size of the set small. Thus, this approach has the potential to provide some of the benefit of a large set of potential hiding places while avoiding the computational complexity. Additionally, this approach mirrors the fact that humans can generally perceive nearby opportunities to hide more easily than ones in the distance, and furthermore, the nearer ones are more likely to be useful.

The sensor grid approach takes its name from the fact that the set of potential hiding places that are tested by the algorithm is fixed relative to the agent. It is as if the agent had a collection of observer-detecting sensors fixed with regard to the agent and one another moving wherever the agent moves. A simplified overview of the algorithm is provided in Figure 1.

Figure 1

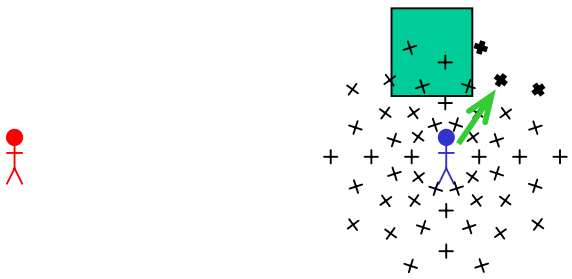

Figure 1: Top-down diagram illustrating the sensor grid approach. The agent (blue) is at right and a single observer (red) is at left. The array of sensors (plus signs) surrounds the agent. A single vision-obstructing object is present (the green square). If a sensor cannot see the enemy, its location is hidden (bold plus signs). The agent chooses the nearest hidden sensor that is accessible (e.g. not inside an object),and moves there (green arrow).

WAYPOINT SYSTEM OVERVIEW

Waypoint systems for pathfinding are quite common in First Person Shooter games. They provide fairly good results when using A*, the usual game industry pathfinding algorithm. However, waypoints can be used for much more than just pathfinding.

We have extended the standard Unreal waypoint system to include not only standard nodes used for pathfinding, but the following as well:

1. Cover nodes
2. Formation nodes
3. Peek-out nodes
4. Tree nodes

Figure 2 depicts a typical cover node setup.

Figure 2

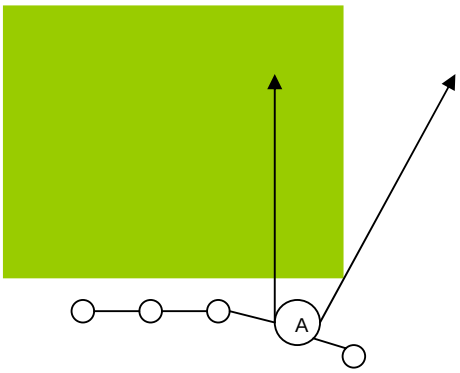

Figure 2: Cover node A is shown with 3 formation nodes to the left and a peek-out node to the right. The 2 handles used to create the angle of coverage are depicted as vectors connected tangentially to the cover node. The angle between these 2 vectors would be the computed angle of coverage for cover node A.

We have also added an initial step to the Unreal waypoint system that is run when paths are built in the Unreal editor. This step pre-computes the visibility amongst all nodes in the level with the exception of formation and peek-out nodes. This provides each node with an easily accessible list of nodes it can "see" at runtime. This optimization greatly reduces the number of rays that need to be cast for line of sight checks which can make standard node-based systems unusable on consoles such as the PlayStation 2.

Cover Nodes

Cover nodes are placed at points in the level which provide adequate cover against some set of weapons. They contain data which provides the agent with an idea of which direction the nodes provides cover towards, what stance to assume for maximum coverage, and what types of weapons it provides cover against.

When a designer places a cover point in the editor he is presented with a circular node that has two vectors we call "handles" connected (see diagram below.) These handles are used by the designer to create the angle of coverage for the given node. By rotating the handles to the desired positions the designer creates an arc. This arc, easily computed using 2D vector algebra, represents the angle that

the given node provides cover against. When the designer runs the pre-computed visibility check each node casts a line of sight ray to each other node to see if it is visible. Each node then stores a list of nodes that it can see. Finally, nodes are marked as to whether or not they are in the given nodes angle of coverage. If they do not fall within the angle of coverage they are marked as being dangerous. While this is an N^2 operation it is only run on the entire set of nodes once. After the initial run, only nodes that have been modified will be rebuilt when pre-computed visibility is calculated. It should be noted that cover nodes assume a static environment. They are not designed to work with deformable terrain or other types of dynamic world geometry.

At runtime it is a trivial task to query a given nodes list of dangerous nodes. When being fired upon an agent queries the list of dangerous nodes for the node he currently occupies. If the enemy firing at him occupies a node that is not in the dangerous nodes list the agent simply assumes the correct posture for cover at his present node. If the enemy's node is in the dangerous nodes list the agent has to weigh the cost of either moving to a location that provides cover from the enemy, or returning fire and hoping for the best. If either agent does not occupy a node the node they are closest to can be used for determining cover. Alternatively, a line of sight check could be used. Similar techniques have already been implemented and tested (Liden 2002).

Formation Nodes

Formation nodes are used by fireteams to determine where each member of a fireteam should go when the fireteam leader occupies a cover node. Each cover node has a set of formation nodes associated with it by a level designer using the Unreal editor. A fireteam is a group of 4-5 soldiers who are all under the control of a designated fireteam leader. Only the leader uses the pathfinding system, all other fireteam members have a distance and orientation they maintain from the fireteam leader at all times. When a fireteam leader determines the team needs to take cover he moves to a cover node, and the other members move to the formation nodes that have been associated with the given cover node. Formation nodes are not included in the pre-computed visibility step. They share the same visibility as the cover node they are associated with.

Peek-out Nodes

Peek-out nodes are used by agents to move out from behind cover and return fire at the enemy. During level creation the level designer can associate up to two peek-out nodes with each cover node. Whenever an agent wants to return fire from a cover node he would follow these steps:

1. Lean out from the cover node, cast a ray, and see if the enemy is visible. If he is visible fire, if not proceed to step 2.
2. Check to see if at least one peek-out node is unoccupied. If there is an available peek-out node proceed to step 3.
3. Move a pre-set distance along the path between the cover node and the peek-out node.
4. Cast a ray towards the enemy. If the enemy is visible, fire. If the enemy is not visible return to step 3.

These steps would continue until either the agent had a clear line of sight to the enemy, or he had moved all the way to the peek-out node location and still could not see the enemy. As with formation nodes, peek-out nodes are not included in the pre-computed visibility step.

Tree Nodes

Tree nodes are very similar to cover nodes except they are used specifically around trees found in the level. They function exactly the same as a standard cover node except that they never have any formation or peek-out nodes associated with them. All trees in our current game have a small enough diameter that an agent never has to move out from behind them to return fire. He merely has to lean out in a given direction.

INTEGRATING THE SENSOR GRID AND WAYPOINT SYSTEMS

In a previous paper we discussed how the sensor grid approach was motivated as a replacement for navigating to safety on sparse waypoint graphs. We also discussed integration with waypoints as an extension to the sensor grid system. This combined approach is being implemented in a currently unannounced Unreal engine based first person shooter. While the sensor grid approach is highly effective at finding cover, it relies on numerous line of sight checks for each agent. Casting rays to check for line of sight is a very expensive computation, and with the limited resources inherent to consoles such as the Xbox and PlayStation 2 this approach is not feasible. By combining the sensor grid approach with a waypoint system we have created a system that is both fast and fairly inexpensive, but also yields very realistic results.

The major modification to the pure sensor grid approach is that now, when an agent needs to find cover, all cover nodes within some pre-set distance from the agent's location are checked first. If an adequate cover node is found the agent proceeds to that node. If no nodes are found, then the standard sensor grid algorithm is run. By only running the full sensor grid algorithm when no adequate cover nodes are found we greatly reduce the number of rays cast per frame. The beauty of the system is that it can be throttled by adding more nodes to the waypoint graph. If the system is being used on high end PCs with a lot of cheap memory simply reduce the granularity of the waypoint graph to allow the sensor grid algorithm to run more often. Or, if running the system on a console with limited resources, increase the granularity of the graph to reduce the usage of the sensor grid algorithm. An additional advantage is that even when the sensor grid is used to locate cover, the waypoints can be used to improve pathfinding to the covered point. This improvement is particularly significant in highly constrained environments (e.g. helping navigate through doorways).

EXTENSIONS

Disabling Waypoints in a Dynamic Environment

One disadvantage to using a standard waypoint system for cover is that it cannot handle dynamic environments. For example, if you have cover point A behind a wall, and that wall is destroyed by artillery, point A is no longer a valid cover point. Since waypoint graphs are computed at build time, and are static, there is no way to update the graph and notify agents that point A is no longer usable for cover.

The sensor grid deals with this problem by constantly scanning the environment for cover locations via line of sight checks. Thus, it can easily handle dynamic changes to the environment.

An interesting extension to the system we describe in this paper would be a method of disabling nodes when the environment changes. Going back to the example given above, after the wall is destroyed cover point A would be disabled and excluded from any further pathfinding searches. Now, when the agent is in the area near cover point A he uses the sensor grid algorithm to find adequate cover in the rubble of the former wall.

EXPERIMENTS AND RESULTS

The algorithm was implemented on top of America's Army version 2.0, which uses the Unreal Warfare engine. The core of the sensor-grid code was written in UnrealScript, and is approximately 500 lines in length. The extensions to the waypoint system were primarily made in C++ code, and added about another 1000 lines of code. All tests were carried out on a desktop PC with a Pentium 4 processor, 1 GB of RAM, and a GeForce 5600FX with 256 MB of RAM.

Running the algorithm provided nearly instantaneous results, and no slowdown in gameplay was noticed. Agents were able to successfully find cover behind various types of objects such as trees, rocks, buildings, and vehicles. In addition, agents now successfully traversed doorways which they were unable to do in the sensor-grid only approach.

The one area that can slow the running time of the algorithm down noticeably is the reliance on ray casting for the line-of-sight checks used by the sensor-grid. Ray casting is a very expensive operation in the Unreal engine. When multiple agents are all casting multiple rays each frame in an effort to find cover there is a noticeable drop in the framerate of the game. This is an even more serious problem on consoles such as the PlayStation 2 where the limited amount of memory and CPU resources make it virtually impossible to perform the sensor-grid part of the algorithm in its current form. We are currently working on a scheduling system for ray-casting which will time-slice the process and spread the casting of rays over multiple frames. We hope this scheduling system, used in conjunction with the pre-computed visibility system, will make this a viable cover finding algorithm for console based first-person shooter games.

CONCLUSIONS

We have presented a system that combines the ease of use and quick access to data of a waypoint system with the sensor grid approach and its robustness in dealing with dynamic environments.

The technique we describe is very fast when only node data needs to be queried to discover an adequate cover point. The efficiency at runtime is achieved by pre-computing node visibility at build time, as well as embedding key data in each node. When an adequate node is not discovered the extremely robust sensor grid algorithm is run which is very effective at finding cover quickly.

The sensor grid algorithm can sometimes make mistakes which are described in our previous paper (Darken 2004). Additionally, the waypoint system is subject to the constraint that in order for it to be completely effective, agents must reside on a waypoint anytime they check for cover. The pre-computed visibility relies on the fact that agents are always on a waypoint. They system will work if agents use their closest waypoint for cover calculations, however errors may creep in. To counteract this problem the sensor grid algorithm could be scheduled to run anytime an agent needs cover and he is not on a node.

Computing lines of sight is already a major component of the computational budget devoted to AI for many computer games. We feel this system greatly reduces the need for a large number of line of sight checks, but when necessary can use them to great effect.

ACKNOWLEDGEMENTS

Portions of this work were supported by funds from the Naval Postgraduate School, the Navy Modeling and Simulation Management Office, and the U.S. Army Training and Doctrine Analysis Center, Monterey.

BIOGRAPHY

GREGORY PAULL received his Masters Degree in Computer Science from Boston University in 2002. He is currently pursuing a PhD in AI through the MOVES Institute at the Naval Postgraduate School. He is also a full-time AI programmer at Secret Level Inc., a small computer game company located in San Francisco, CA. He has previously worked at Electronic Arts and Looking Glass Studios.

REFERENCES

Darken, C., Morgan, D., and Paull, G. **"**Efficient and Dynamic Response to Fire**"**, *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004.

Darken, C. 2004. "Visibility and Concealment Algorithms for 3D Simulations", *Proceedings of Behavior Representation in Modeling and Simulation (BRIMS) 2004.*

Kelly, J., Beall, A., and Loomis, J. To appear. "Perception of Shared Visual Space: Establishing Common Ground in Real and Virtual Environments", to appear in *Presence.*

Liden, L. 2002 “Strategic and Tactical Reasoning with Waypoints”, *AI Game Programming Wisdom*, Charles River Media, pp. 211-220.

Rabin, S. 2000. “A* Speed Optimizations”, *Game Programming Gems*, Charles River Media, pp. 272—287.

Reece, D., Dumanoir, P. 2000. “Tactical Movement Planning for Individual Combatants”, Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation. Available at http://www.sisostds.org.

Reece, D. 2003. “Movement Behavior for Soldier Agents on a Virtual Battlefield.” *Presence*, Vol. 12, No. 4, pp. 387—410, August 2003.

Richbourg, R., and Olson, W. 1996. “A Hybrid Expert System that Combines Technologies to Address the Problem of Military Terrain Analysis,” *Expert Systems with Applications*, Vol. 11, No. 2, pp. 207—225.

Snook, G. 2000. “Simplified 3D Movement and Pathfinding Using Navigation Meshes”, *Game Programming Gems*, Charles River Media, pp. 288—304.

van der Sterren, W. 2002. “Tactical Path-Finding with A*”, *Game Programming Gems 3*, Charles River Media, pp. 294—306.