Strategic and Tactical Reasoning with Waypoints

Lars Lidén – Valve Software
lars@valvesoftware.com

For the behavior of computer controlled characters to become more sophisticated, efficient algorithms are required for generating intelligent tactical decisions. Non-player characters (or NPCs) commonly use waypoints for navigation through their virtual world. This article will demonstrate how pre-processing the relationships between these waypoints can be used to dynamically generate combat tactics for NPCs in a first person shooter or action adventure game. By pre-calculating and storing tactical information about the relationship between waypoints in a bit string class, NPCs can quickly find valuable tactical positions and exploit their environment.

Beyond Pathfinding

It is common practice in 3D games for level designers to place waypoints in their levels for NPCs to navigate the environment [Lidén00]. These waypoints serve as nodes in a node-graph that represents all the ways in which an NPC can navigate through the world. Connections between nodes in the node graph are either generated automatically in a pre-processing step, or placed manually by a level designer. Pathfinding algorithms such as A* are then used to generate routes through the node-graph [Stout00].

As the waypoints used for navigation inherently contain information about the relationship between positions in the world, they can be exploited to efficiently calculate information about the strategic value of particular world locations [Lidén01] and [vanderSterren01]. A closer look at a node-graph reveals that in addition to pathfinding information it also contains data about the approachability of each waypoint, whether a waypoint is a dead end, and even if a waypoint is the sole approach to a particular region. Furthermore, with a little extra work, the node-graph can be used to generate static visibility information between nodes. Pre-calculating whether one waypoint has line-of-sight to another allows for fast run-time evaluation of map locations.

Fast Map Analysis

For the real-time evaluation of a map to be effective, an economical method for assessing the danger and strategic value of each location in the map must be employed. This is of particular importance when an NPC has to contend with multiple enemies, as any assessment must take the visibility and location of each enemy into account. The computational cost of such a method must not become prohibitive as the number of nodes and enemies increases.

An effective technique is to store connectivity and visibility information in a *bit string* class that consists of a string of bits of arbitrary length with operators for Boolean operations such as *<and>, <or>* and *<not>*. For a node-graph $\boldsymbol{N_n}$, consistingof $\boldsymbol{n}$ nodes, connectivity is represented by a set of n bit strings, $\boldsymbol{C_n}$, where the length of each bit string corresponds to the number of nodes in the node graph and each bit represents the connectivity from node $\boldsymbol{n}$ to every other node in the node-graph.

Network visibility is represented by a set of bit strings, $\boldsymbol{\Lambda_n}$, where the length of each bit string corresponds to the number of nodes in the node graph and each bit represents the visibility from node $\boldsymbol{n}$ to every other node in the node-graph. If $\boldsymbol{\beta\ (E_i, t)}$ is a function that returns the nearest waypoint for enemy $\boldsymbol{i}$ at time $\boldsymbol{t}$ then the visibility of an enemy $\boldsymbol{i}$ is given by the bit string:

1. $$V_{it} = \Lambda_{\beta(E_i\, t,)}$$

Each bit in the bit string represents whether or not each waypoint is visible to enemy, $\boldsymbol{i}$. For notational simplification, the time component will not be included in subsequent equations, as it is understood that formulas apply to a specific time.

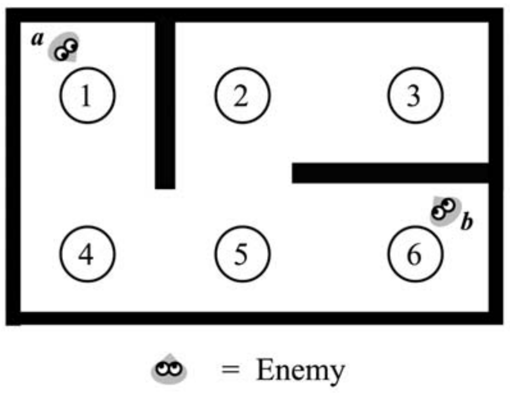
Using Visibility

The most straightforward use of visibility is to determine which locations are potentially *safe* and those that are likely to be *dangerous*. For an NPC with a set of $\boldsymbol{k}$ enemies $\boldsymbol{E_k}$, the set of dangerous waypoints is determined by *<or>*ing the visibility bit strings for each enemy's nearest waypoint:

2. $$V = \bigcup_{j=0}^{j=k} V_j$$

$\boldsymbol{V}$ is then the set of waypoints from which the NPC might have a line-of-sight (LOS) to shoot at an enemy. To determine LOS for certain, each enemy must still be checked explicitly as visibility bit strings tell us only about the visibility of each enemy's nearest waypoint not their actual positions.

Safe nodes are given by the inverse, $\overline{V}$. These nodes are good candidates for safe locations to which an NPC can flee or reload safely (Figure 1).



$$\begin{array}{r|cccccc|}
 & \multicolumn{6}{c}{\text{Node}} \\
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
(a)\ V_1 = & 1 & 0 & 0 & 1 & 0 & 0 \\
V_2 = & 0 & 1 & 1 & 0 & 1 & 0 \\
V_3 = & 0 & 1 & 1 & 0 & 0 & 0 \\
V_4 = & 1 & 0 & 0 & 1 & 1 & 1 \\
V_5 = & 0 & 1 & 0 & 1 & 1 & 1 \\
(b)\ V_6 = & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
\end{array}$$

DANGER NODES:

$$V = V_a \cup V_b = 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1$$

Nodes 1, 4, 5 and 6 are dangerous

SAFE NODES:

$$\overline{V} = 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0$$

Nodes 2 and 3 are safe

Figure 1. Dangerous and safe nodes.

Safe Pathfinding

Another straightforward use of visibility information is to find safe paths for an NPC to traverse the environment. Most pathfinding algorithms use a cost function to determine the shortest path through the node-graph [Stout96]. If a penalty is added to transitions that pass through nodes that are designated as not being safe, the pathfinding algorithm will be biased towards finding safe paths for the NPC. Depending on the size of the penalty given to dangerous waypoints, the NPC can be made to favor either: (1) safe paths when they aren't particularly longer than a regular path, or (2) safe paths at any cost, even when they take the NPC a long way out of it's way.

Intelligent Attack Positioning

Although an all-out frontal assault may have its place, it isn't always the most intelligent form of attack. The set of dangerous waypoints, $\boldsymbol{V}$, may give us locations from which an NPC can attack its enemies; it does nothing to protect the NPC from attack. A more sophisticated strategy would be to find a location from which the NPC can attack a particular enemy that also protects it from simultaneous attack by other enemies. Fortunately, determining such locations is straightforward and can be computed quickly.

As before, the set of potential waypoints from which a particular enemy, $\boldsymbol{E_a}$, can be attacked is given by the set of nodes that are visible to that enemy (Figure 2.1):

3. $$V_a = \Lambda_{ß(E_a)}$$

Nodes that are visible to the NPCs other enemies are determined by <*or*>ing their individual visibilities (Figure 2.2):

4. $$V_{\bar{a}} = \bigcup_{j=0}^{j=k} V_j, j \neq a$$

These waypoints are then eliminated from the set of potential attack positions, by taking the intersection with its inverse, namely, the set of waypoints that are safe from the other enemies, $\overline{V_{\bar{a}}}$ (Figure 2.3):

5. $$V'_a = V_a \bigcap \overline{V_{\bar{a}}}$$

The result is a set of locations from which a particular enemy can be attacked that are safe from all other enemies.

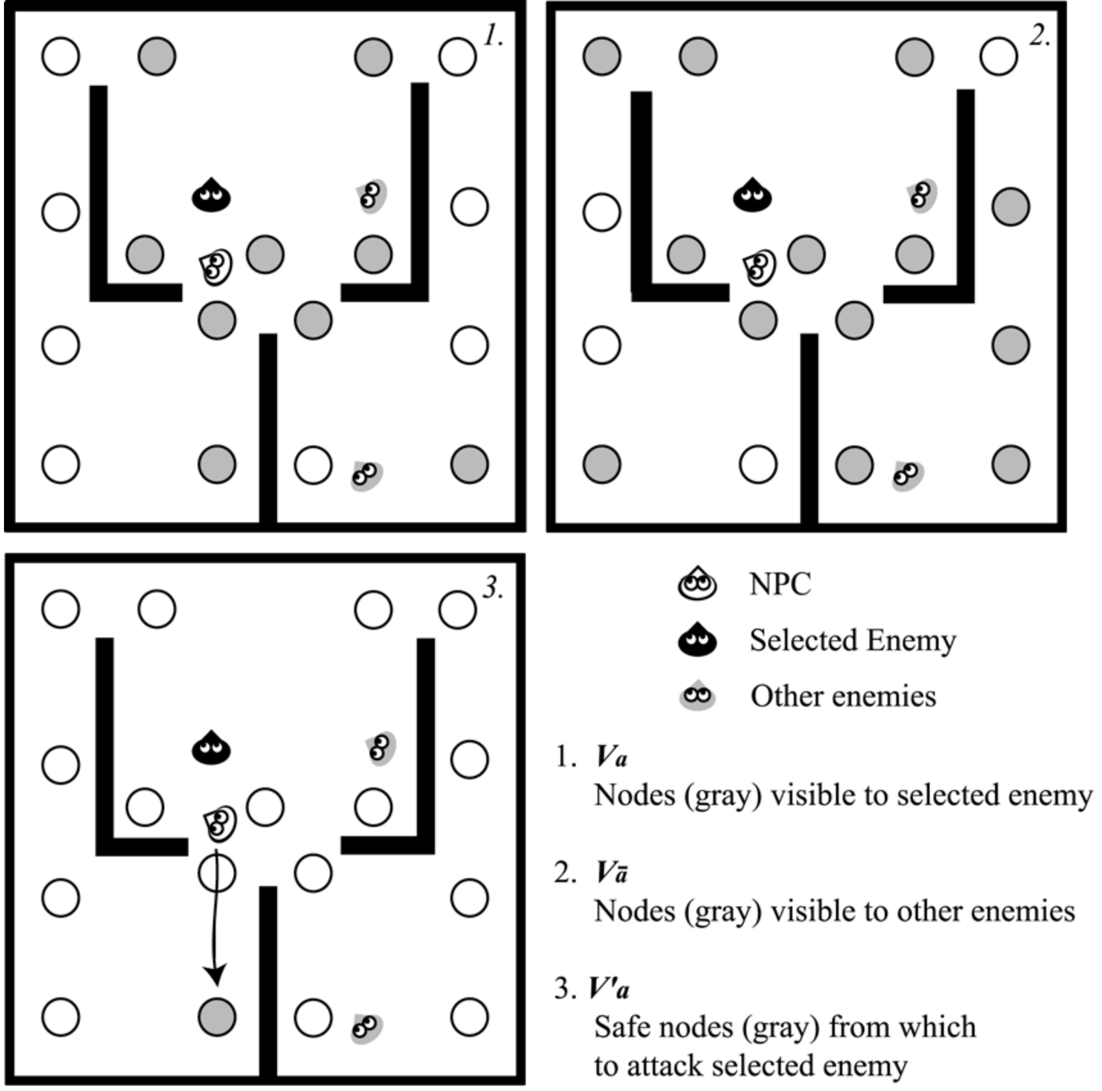

1.
2.
3.
NPC
Selected Enemy
Other enemies
1. Va
Nodes (gray) visible to selected enemy
2. Vā
Nodes (gray) visible to other enemies
3. V'a
Safe nodes (gray) from which
to attack selected enemy

Figure 2. Finding a safe place for an NPC to attack a selected enemy.

Taking it further

So we have a quick way to find candidate locations from which to attack a particular enemy that is also safe from other enemies. What are some other strategies that an intelligent opponent might employ? It would be nice if our attack location had safety nearby in case our NPC needed to reload or the selected enemy suddenly launched a rocket in our NPC's direction. Fortunately this is an easy qualification to add. From Equation 2, the set off all safe nodes is given by $\overline{V}$. Furthermore, from the node-graph, we know the connectivity of each of the candidate attack nodes, namely $\boldsymbol{C_a}$. Therefore nodes in V'_a should be eliminated if:

6. $$C_a \bigcap \overline{V} = 0$$

The remaining set of locations is of ones that have LOS to the selected enemy, are protected from other enemies, and also have nearby locations to take cover from all enemies.

Flanking

Another interesting combat behavior is that of flanking. Flanking takes both the position and facing direction of an enemy into account, the goal being to surprise an enemy by attacking from behind. The procedure for finding potential flanking attack locations is virtually identical to that of finding normal attack locations. The only difference being that before eliminating nodes that are visible to other enemies (Equation 5), the set of potential attack locations, $\boldsymbol{V_a}$ should be culled to remove waypoints that the selected enemy is facing.

Once a flanking waypoint has been selected, if the pathfinding algorithm has been weighted to use safe waypoints, the NPC will look for a path that is out of sight of the selected enemy. The resulting behavior will be that of the NPC sneaking around behind to attack their enemy.

Static Waypoint Analysis

Unless cheating is employed, it's likely that an NPC doesn't have perfect knowledge about the locations of each of its enemies. Additionally, an NPC might want to place itself in a strategic location before enemies arrive. Consequently, in addition to finding tactical positions for a set of enemy locations during run-time, it is also useful to characterize each waypoint's strategic value in a static environment. Such characterization can be done in a pre-processing step before the game is played.

As we have seen, in quantifying a location's strategic value, visibility is perhaps the most important factor. Highly visible locations are dangerous as they can be attacked from many positions. One can readily identify such locations by looking at visibility between nodes in the node graph. The danger of each node can be characterized by giving it a weight based on the number of other nodes in the graph that have visibility to that node (Figure 3.1). If the weights are used to adjust the pathfinding cost function, the NPC will prefer to take paths that are less likely to be visible to an enemy.

Although nodes with low visibility are safe, they don't have a great deal of attack potential. Those with high visibility have the advantage that they can attack many positions. Ideally we want a location that is safe, but has the greatest attack potential. Such locations can readily be determined from the node graph by selecting nodes that have high visibility whose neighbors have low visibility (Figure 3.2).

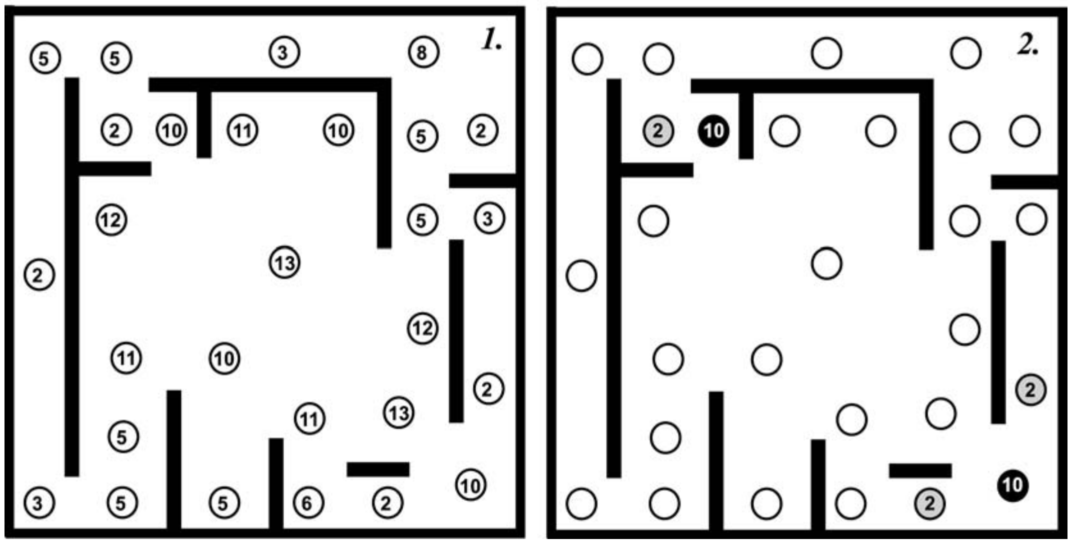


Figure 3. Static waypoint analysis. 1. Nodes marked for their visibility count. Nodes in protected areas such as hallways automatically get a low weighting, while those in exposed areas get a high rating. 2. Sniper locations (marked in black) have high exposure and are attached to regions with low exposure (marked in gray).

Pinch Points and Squad Tactics

Observation of human players reveals that experienced FPS players anticipate the actions of their opponents [Laird00]. For example, if an enemy enters a room with a single exit, rather than follow the enemy into the room, an experienced player will wait just around the corner, setting up an ambush at the exit point.

One can readily pre-calculate such tactical pinch-points by analyzing the node graph (Figure 4.1):

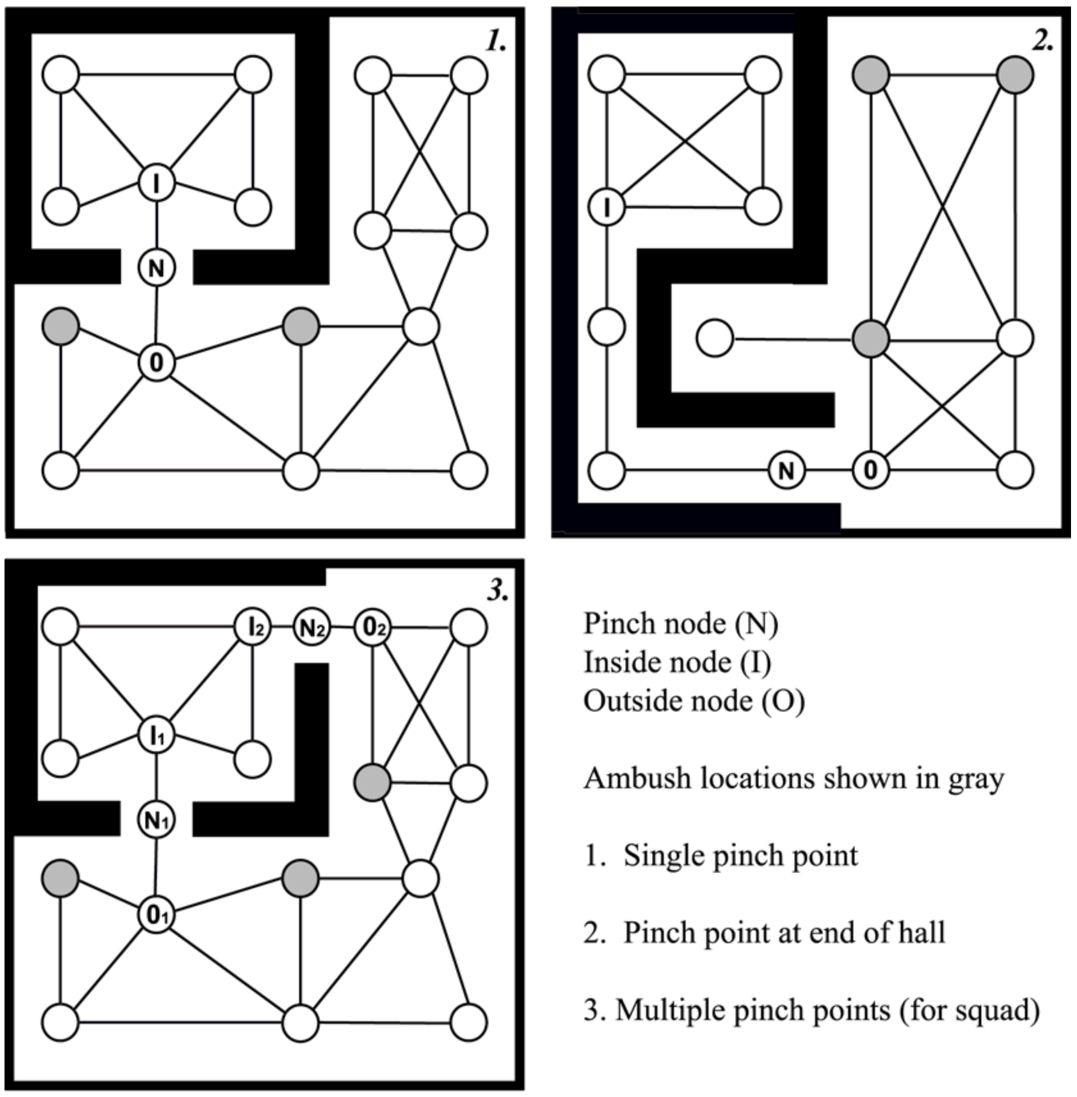

1.
I
N
O
2.
I
N
O
3.
I2
N2
O2
I1
N1
O1
Pinch node (N)
Inside node (I)
Outside node (O)
Ambush locations shown in gray
1. Single pinch point
2. Pinch point at end of hall
3. Multiple pinch points (for squad)

Figure 4. Finding places to ambush an enemy.

For each node, **N** in the node graph with only two neighbors:

- Temporarily eliminate node, **N**, from the graph, call its neighbors as **A** & **B**.
- If both **A** & **B** are connected to large regions, **N** is not a pinch point, try another **N**.
- Attempt to find a path between **A**& **B**.
- If path exists, **N** is not a pinch point, try another **N**.
- Call the node connected to the larger region, **O** (for outside).
- Call the node connected to the smaller region, **I** (for inside).

To find potential ambush locations for **N**, we need a waypoint that has line-of-sight to **O** and not **N**. This is simply:

7. $$A = V_O \bigcap \overline{V}_N$$

When an enemy enters the region gated by node **I**, and NPC can set up an ambush at any of the nodes in the set **A**.

Things are slightly more complicated when detecting pinch points at the end of hallways. To include such cases, the following step must be added (Figure 4.2):

- If **O's** neighbor has only one other neighbor in addition to **N**
    - Move **N** to **O**.
    - Move **O** to the other neighbor of the old **O**.
    - Repeat until **O** has only one neighbor.

Squad Tactics

Regions with more than one exit can still qualify as having valid pinch points for NPCs that organize into squads as each exit from a region can be guarded by a different NPC in the squad. For a squad with two members (Figure 4.3):

```
For each node, N_1 in the node graph with only two neighbors:
• Temporarily eliminate node, N_1, from the graph, call its neighbors as A & B.
• If A & B are connected to large regions, N_1 is not a pinch point, try another N_1.
• Attempt to find a path between A& B.
• While generating the path if a node with only two neighbors is found,
    o Temporarily eliminate it and call it N_2.
    o Attempt to find a path between A& B.
• If path exists, not a pinch point, try another N_1.
• Call the nodes connected to the larger regions, O_1 and O_2 (for outside).
```

The ambush points for the two member of the squad are:

8. $$A_1 = V_{O_1} \bigcap \overline{V}_{N_1} \quad \text{and} \quad A_2 = V_{O_2} \bigcap \overline{V}_{N_2}$$

This can easily be generalized for squads of any size.

Limitations and Advanced Issues

There are a couple of caveats to the methods discussed here. Although use of a bit string class to store visibility and calculate tactical positions keeps the memory and computational costs down, for exceptionally large node-graphs the size of bit strings could get prohibitively large. Each node stores one bit for every other node in the network, for a total of [# *of nodes*]^2 bits stored in memory. Each <and> and <or> operation requires [# *of nodes* / *sizeof (int)*] bitwise operations. The size of bit strings can be reduced by eliminating visibility and connectivity data for nodes in widely separated regions of world space that have no chance of having visibility or direct connectivity. Rather than representing the world as one large node-graph, a hierarchy of networks can be employed. Using node-graph hierarchies rather than a single large node-graph also has other unrelated advantages, including faster pathfinding [Rabin00].

A second limitation is that the effectiveness of pre-calculated tactical information relies to some degree on a level designer placing nodes in proper positions. If, for example, a level designer neglects to place nodes at a second exit to a region, the pre-processing step may incorrectly assume that a pinch point exists when in actuality there is a second exit that is usable by the player. With experience level designers can learn proper node positioning. Additionally research on the automatic generation of node placement by the computer may eliminate the need to rely on level designers for intelligent node placement.

There are several other issues to consider that are beyond the scope of this paper. Here it was assumed all connections were bi-directional. In many situations connections (such as jumps) are uni-directional. The determination of pinch points will be slightly different for uni-directional connections and movement in both directions must be checked.

In many games it is possible for a player or an NPC to gain cover by ducking behind obstacles. In such cases, a single location may serve as both a cover location and a position from which to attack. Such locations can be exploited by annotating the relevant waypoints. This can be done either manually by a level designer, or done as part of the pre-processing computations by the computer.

Conclusions

Computer controlled characters commonly use waypoints for navigation of the world. This article demonstrated how existing waypoints can be used to automatically generate combat tactics for computer-controlled characters in a first person shooter or action adventure game. The techniques may be applicable to other genres but have yet to be tested in these arenas.

A level designer who is familiar with the behavior of NPCs and their ability to navigate, usually places waypoints in key locations in an environment. Tactical information about locations in the environment can be efficiently calculated from this implicit data and exploited by NPCs. Storing data in a bit string class allows for an economical method for calculating tactical information whose computational cost remains reasonable for large numbers of nodes and enemies.

Pre-calculated visibility information can provide a rough approximation of the danger of particular areas in a given map and locations from which an NPC can mount an intelligent attack. With waypoint visibility information, it is relatively straightforward to establish a line-of-sight to an enemy, automatically generate flanking locations, sniping locations, and to detect “pinch locations” where an enemy can be ambushed.

References

[Laird00] Laird, John, "It Knows What You're Going to Do: Adding Anticipation to a Quakebot," *Artificial Intelligence and Interactive Entertainment: Papers from the 2000 AAAI Spring Symposium*, Technical Report SS-00-02, 41-50, 2000.
[Lidén00] Lidén, Lars, "The Integration of Autonomous and Scripted Behavior through Task Management," *Artificial Intelligence and Interactive Entertainment: Papers from the 2000 AAAI Spring Symposium*, Technical Report SS-00-02, 51-55, 2000.
[Lidén01] Lidén, Lars, "Using Nodes to Develop Strategies For Combat with Multiple Enemies," *Artificial Intelligence and Interactive Entertainment: Papers from the 2001 AAAI Spring Symposium*, Technical Report SS-01-02, 2000.
[Rabin00] Rabin, S., "A* Speed Optimizations," *Game Programming Gems*, Charles River Media, 2000.
[Stout00] Stout, W. B., "The Basics of A* for Path Planning," *Game Programming Gems*, Charles River Media, 2000.
[Stout96] Stout, W. B., "Smart Moves: Intelligent Path-Finding," *Game Developer Magazine*, October 1996.
[vanderSterren01]: van der Sterren, W., "Terrain Reasoning for 3D Action Games", *Game Programming Gems 2*, 2001.