Level Annotation and Test by Autonomous Exploration: Abbreviated Version

Christian J. Darken

MOVES Institute and Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
cjdarken at nps dot edu

Abstract

This paper proposes the use of an autonomous exploring agent to generate and annotate the waypoint graph as an off-line process during level development. The explorer incrementally generates the waypoint graph as it explores the level via the same motion model used for player movement, and then revisits the waypoints to annotate them using image-based techniques. Points where the explorer becomes stuck or falls off of the level are flagged for later investigation by a level designer.

Introduction

AI-controlled characters (NPC's) typically need to move about the geometry of a game's levels. Often they are hunting or hiding from the player. A game level starts out as a mere collection of polygons. NPC movement algorithms require a graph of waypoints as an input to some variant of A* search. Once the waypoint graph is created, the next step is often to attach extra information to the waypoints that will be used to drive run-time behavior, e.g. how good of a hiding place or firing position a waypoint is.

The quality of waypoint graph generation and annotation is often a limiting factor in the quality of run-time NPC behavior. Modern titles typically contain a lot of geometry, and the pursuit of more realistic behavior has driven game makers towards increasingly fine waypoint graphs. When waypoint generation and annotation is done all or partly by hand, it contributes to the ballooning cost of content generation, and adds a significant barrier to user-generated content. Proper waypoint generation and annotation requires experience with the run-time AI behavior that can challenge the AI creators, not to mention level designers and mod makers. We discuss existing

This paper proposes the use of an autonomous exploring agent to generate and annotate the waypoint graph as an off-line process during level development. The explorer incrementally generates the waypoint graph as it explores the level via the same motion model used for player movement, and then revisits the waypoints to annotate them using image-based techniques. Points where the explorer becomes stuck or falls off of the level are flagged for later investigation by a level designer.

A side-effect of generating waypoints by movement around the level is that problems with the level geometry can be automatically detected and flagged. Some recent AAA titles still have occasional problems where players can become stuck in, or fall through, the geometry.

We believe the primary contributions of this work are:

- Waypoint placement based on exploring the level using the player motion model. The AI only goes where the player can go, and level geometry problems can be flagged. Implementation is relatively simple as compared to navigation mesh techniques.
- Constant-time-per-waypoint assessment of cover and view using image-based techniques
- Assessment of waypoints as hiding places or sniper positions based on an empirical model of human target detection that is takes fog, lighting, and camouflage into account.

Related Work

Automated exploration of virtual environments is not a new idea, dating back at least to Mauldin's TinyMUD chatterbots (Mauldin 1994). These bots would constantly traverse the virtual world updating their knowledge, and could produce the shortest route between two points on request from human players. The environment was not, however, based on 3D geometry. It was created as a graph from the beginning, so waypoint placement was not an issue.

Today, the dominant method of automatic waypoint generation is the navigation mesh (Snook 2000) (Tozour 2002) (Farnstrom 2006). In a nutshell, navigation mesh

techniques start from the raw polygons of the level and produce a subset which constitute the navigation mesh. Polygons are routinely split and sometimes merged in the process of generating the mesh. Midpoints of edges of the navigation mesh polygons are the waypoints. The beauty of the navigation mesh is that the process of generating it can be automated, at least to a large degree. The down-side of the mesh is its difficulty of implementation. The mesh generation algorithm must take into account the collision geometry and motion model of the NPC's,. The relationship between these data and the subset of the raw geometry that belongs to the mesh is intricate, so much so that we see this as a limitation of navigation meshes that we would like to avoid.

Automated annotation of waypoints for visibility and cover was first described by Liden (2002). This work performed ray traces (line-of-sight checks) off-line and cached the results for use at run-time, e.g. for finding cover or firing positions. Additional off-line analysis of the ray trace data could be used to find and label sniping positions. Straatman et al. (2006) describes an extension of Liden's approach with a description of a compression strategy for the ray-trace data and several examples of how this data can be exploited on-line to produce various interesting behaviors. Our work uses image-based methods rather than ray-tracing to annotate waypoints. This approach has both computational and performance advantages. From a computation cost point of view, it replaces the O(N) per waypoint cost of all-pairs ray tracing with O(1) approach based on rendering a fixed number of views from each waypoint. Note that the cover and visibility information we cache is not intended to completely replace runtime chacks as in Liden (2002). From a performance point of view, rendered views provide an unprecedented opportunity to evaluate hiding positions based on how likely a player would be to detect an NPC located at a particular waypoint. Darken and Paull (2006) describe some of the problematic aspects of pre-computing cover information and propose run-time augmentation of the waypoint graph as a solution.

The automated processing of images from a virtual environment, i.e. “synthetic vision” (Renault et al. 1990), was applied to run-time navigation in a computer game by Blumberg (1997). To our knowledge, this is its first application to automated level annotation.

System Architecture

The autonomous level explorer has a similar architecture to an artificial creature such as the famous fish of Tu and Terzolpoulos (1994). It takes an image of the world as input and interacts with world via (simplified) physics. Of course, our motivation is not to create artificial life, but to perform a task, and therefore knowledge of the true positions of dummy targets placed in the world and even control of the target color is allowable. Note that the explorer only runs off-line. Only the waypoint graph and annotations are used at run-time.

Waypoint Finding

Given the geometry of a level and a single waypoint (or set of waypoints) provided by the user, we describe a method for covering the entire accessible portion of the level with waypoints. Our technique assumes that an NPC motion model has been defined (possibly the same one as is used for player movement). We assume additionally that a finite set of actions (terminating programs to drive the motion model) is available. Each action represents a way to explore that might result in accessing a new part of the level. As a concrete example and the primary one we use in practice, there might be six actions, each of which explores a different point of the compass separated by 60 degrees. The action set is a novel requirement of our technique, and we discuss it further below. An example of a waypoint graph is provided in Darken (2007b).

```
MAIN ALGORITHM

Add user waypoint(s) to list

For each waypoint on list
 For each action:
  Execute action until termination
  Are we somewhere new?
  If so, add new waypoint to list
  Add new edge to waypoint graph
```

The “somewhere new” test requires some discussion. We use a 3D Euclidean distance test against the waypoint set for this purpose. We found the naïve O(N) approach of testing against each existing waypoint to be unacceptably slow. Efficient algorithms for “range queries” of this type and their supporting data structures have been well studied in the field of computational geometry, and there are many good approaches. We implemented a kd-tree (Bentley and Friedman 1979) for this purpose, reducing the computation required on average (for points “in general position”) to O(log N), though the worst case (all waypoints in a circle around the query point) still requires O(N).

Implementing a set of actions need not be a difficult task. The actions produce inputs to the motion model, and respond to whatever outputs are available from the game engine, so a general prescription for them cannot be given. A guarantee of termination is required, and is easily provided by implementing a time out for each action.

Our motion model requires a requested velocity as input. The resulting motion must be checked by determining the change in position of the motion model and its status (walking, sliding, or falling). All six of our actions are based on a single primitive action: moving in a straight line

to a specified x-y position. Our motion model allows sliding along obstacles that do not squarely block the requested step. After each requested step, the actual progress achieved is checked. If the motion model is not closer to the goal by at least half the expected distance given the requested velocity and inter-frame time, the action aborts. If the goal is reached, but the model is out of control (sliding or falling), the action aborts. Otherwise, the requested x-y position is achieved.

Level Test

A little more processing on top of the waypoint finding algorithm discussed above provides a method of finding two common types of problems with newly-created levels: sticking points and holes. A sticking point is a place that is accessible to the player, but which are impossible to leave. The result of visiting a sticking point is usually that the player is forced to restart the game, a very annoying experience. A hole in a level is a place where the player can fall through the geometry of the level. Once again, the only remedy is usually a restart. Both types of problem are encountered, even in some recent AAA titles.

Our algorithm for level testing is very simple. In the course of exploring a level, if all actions available to the explorer at a given waypoint fail to move it to a new location, that waypoint is considered a sticking point. If any action causes the agent to fall further than a specified maximum distance, that waypoint is considered to be near a hole. Waypoints that are sticking points or near holes are colored red in the GUI, enabling them to be quickly detected by a level creator and corrected. An example visualization is given in Darken (2007b).

Waypoint Annotation

We annotate each waypoint with numbers to indicate how much can be seen and how much protection from fire they provide in each of the six directions. Additionally, we analyze how visible the NPC model used at runtime would be at this location using image-based techniques. These techniques are described in Darken (2007b).

References

Bentley, J. and Friedman, J. 1979 “Data Structures for Range Searching”, *ACM Computing Surveys*, Vol. 11, Issue 4, pp. 397-409.

Blumberg, B. 1997. “Go with the Flow: Synthetic Vision for Autonomous Animated Creatures”, Proceedings of AAAI 1997.

Darken, C. and Paull, G. 2006. “Finding Cover in Dynamic Environments”, *AI Game Programming Wisdom 3*, Charles River Media, pp. 405-416.

Darken, C. 2007a. “Computer Graphics-Based Target Detection for Synthetic Soldiers”, Proceedings of Behavior Representation in Modeling and Simulation (BRIMS) 2007.

Darken, C. 2007b. “Level Annotation and Test by Autonomous Exploration”, The full version of this paper is available at http://www.nps.navy.mil/cs/cjdarken/

Farnstrom, F. 2006. “Improving on Near-Optimality: More Techniques for Building Navigation Meshes”, *AI Game Programming Wisdom 3*, Charles River Media, pp.113-128.

Jones, B. (2006) A Computer Graphics Based Target Detection Model. Master's Thesis, Naval Postgraduate School.

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

Reece, D. and Wirthlin, R. (1996). Detection Models for Computer Generated Individual Combatants. In *Proceedings of the 6th Conference on Computer Generated Forces and Behavioral Representation*.

Renault, O., and N. Magnenat-Thalmann, D. Thalmann. 1990. “A vision-based approach to behavioral animation”, The Journal of Visualization and Computer Animation 1(1).

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

Straatman, R., Beij, A., and Van der Sterren, W. 2006 “Dynamic Tactical Position Evaluation”, *AI Game Programming Wisdom 3*, Charles River Media, pp. 389-404.

Tozour, P. 2002. “Building a Near-Optimal Navigation Mesh”, *AI Game Programming Wisdom*, Charles River Media, pp. 171-185.

Tu, Xiaoyuan and D. Terzopoulos. 1994. “Artificial Fishes: Physics, Locomotion, Perception, Behavior”, Proceedings of SIGGRAPH 94.