Level Annotation and Test by Autonomous Exploration

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. The architecture of the explorer appears as Figure 1. Note that the explorer only runs off-line. Only the waypoint graph and annotations are used at run-time.

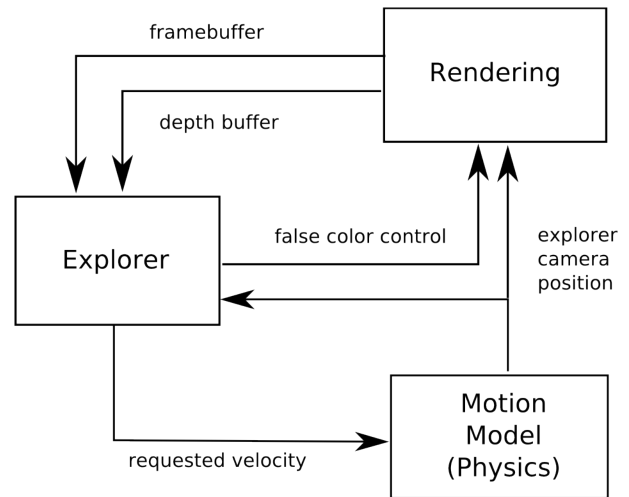

Figure 1: Explorer architecture.

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 produced by the explorer is given as Figure 2.

```
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).

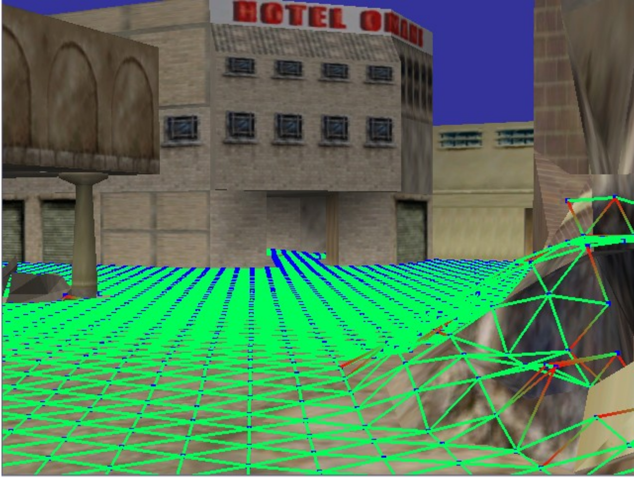

Figure 2: A small part of the waypoint graph generated for a full-sized level. The graph extends to all accessible parts of the level, including building interiors.

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 (see Figure 3), enabling them to be quickly detected by a level creator and corrected.

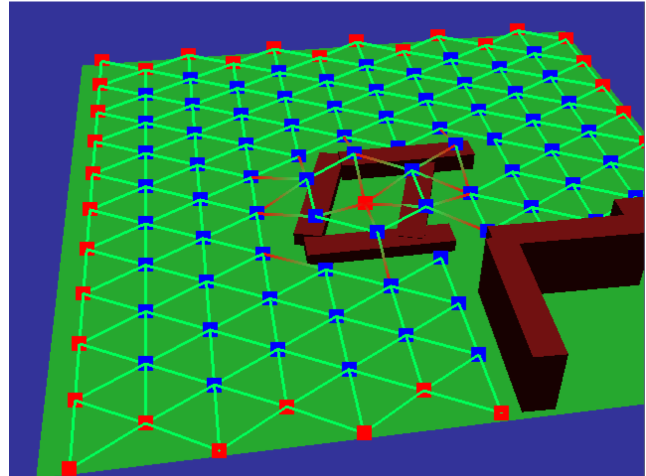

Figure 3: Visualization of the waypoint graph produced for a simple test level with inaccessible region at right, fall points around the edge, and a sticking point at center with one-way edges surrounding it.

Viewshed and Cover 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. These annotations will be used at run-time in conjunction with additional checks in order to help an NPC perform various tasks such as find cover or a good firing position. The basis for both types of annotation are rendered images, of the same type as would be provided to a human player.

The camera is positioned at each waypoint at eye height faced in a specific direction and an image is generated. The depth map corresponding to the image is then accessed. The depth map is an array of floats with the same dimensions as the rendered image. The value at each pixel corresponds to the distance from the camera to the polygon rendered at that pixel. The number is not an actual distance, but is rather a distance passed through a nonlinear function and scaled so that values of 0.0 and 1.0 represent the near and far clipping planes. Using a custom shader it would be possible to get actual distances, e.g. measured in

meters, but we have not found this necessary so far.

The viewshed value is computed by summing over the depth buffer. Larger values mean that more is visible. The cover value is computed by summing over the lower half of the buffer only. Lower values correspond to more cover. See Figure 4 for an example. We use these values for relative comparison only, e.g. to determine which of two waypoints has a better view to the north.

After both values are computed, the camera is rotated 60 degrees and the process is repeated until a full 360 degree sweep is completed.

Visibility Annotation

We annotate each waypoint with a single number indicating how difficult it is to see a character at that location. We accomplish this by making use of the GBBA (Graphics Buffer Based ACQUIRE) target acquisition algorithm, described below. In prior work (Liden 2002) with a similar goal of finding sniping positions, ray trace-based (line-of-sight) models of visibility were used. Ray traces assess a character as visible if an uninterrupted line can be drawn between the eye points of the two characters involved, regardless of fog, lighting conditions or camouflage. GBBA takes these factors into consideration.

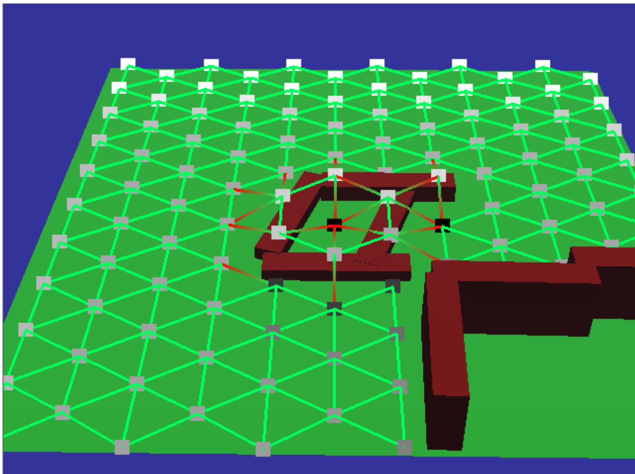

Figure 4: Cover from threats in the north (top of picture), where black represents maximum cover and white no cover.

ACQUIRE is a standard target acquisition model (Reece, 1996). It is capable of producing the probability of detecting a target in a given amount of observation time. ACQUIRE computes the detection probability as a function of the brightness (irradiance in watts per square meter) of the target, the brightness of the background of the target, and the subjective size of the target, in terms of its “number of resolvable cycles”. While not officially recommended as a model of optical target detection, it has been used for that purpose many times in military simulations. Constructive simulations, lacking detailed 3D models of combatants and terrain, obviously must use nominal brightness values, for example using a constant brightness for the background depending on geographic location and time of day, and looking up target brightnesses by target type from a table.

GBBA is an application of ACQUIRE to simulations that are being rendered on conventional graphics cards. It is applicable to any simulation that has full control of the visual rendering of the models and access to the framebuffer.

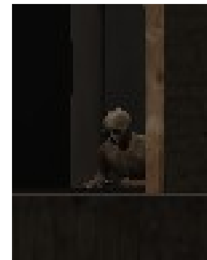

Figure 5: Detail from wide shot of target in a building and partially visible through a window.

GBBA's computation is based on a rendering of the target from the agent's point of view that is generated in the same manner as the rendering of the environment that a human user of the simulation sees. See Figure 5 for an example. The angles and colors of lights and the textures applied to the surface are all taken into account, together with any occluding objects, smoke, and fog. For each agent/target pair, a rendering of the target from the agent's point of view is produced. Since only a tight view of the target is required, we call this image a “mini-render”. This image does not ever need to be shown on a screen; it's sole purpose is to provide the data that will feed GBBA's algorithm.

Figure 6: Normal color mini-render (left), false color mini-render (right).

The mini-render contains complete information on the appearance of the target and background, but ACQUIRE requires us to separate the two. Segmentation of images

into objects as practiced in computer vision is a computationally expensive and error prone operation. Fortunately, in this context we can get pixel-perfect segmentation at the cost of a second mini-render. For the second render, we use our control of the rendering to color the figure bright red. This false-color mini-render is generated for the same scene conditions as the first, and so is in perfect registration with it. We use the false-color mini-render to tighten the view of the target to the minimum rectangular window that includes all target pixels. To include a modest amount of background, we then expand this minimum rectangle by 5% in all directions. The normal color mini-render is cropped identically to maintain registration between the normal color and false color images. The images in Figure 6 have both been cropped according to this procedure.

Not all pixels in the expanded mini-render are considered valid parts of the background, however. One insight behind GBBA is that not all contrast between target and background is equally valuable. Where the "background" is actually closer to the observer than the target, high contrast means that we see the contour of the background well, but not necessarily the target. Experimental testing showed that this aspect of GBBA was helpful in matching the performance of human subjects (Darken 2007).

GBBA uses the depth buffer to exclude any part of the background that is in front of the target from consideration. It does this by first making a pass over all the target pixels (identified using the false-color mini-render) and finding the one that is closest to the camera. Then a pass over the non-target pixels in the false-color mini-render colors them depending on whether how far they are relative to the closest target pixel. In the example presented as Figure 7, the background pixels that are closer than the target, the edge of the window to the right and below the target, are colored blue.

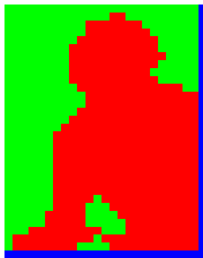

Figure 7: False color mini-render including depth buffer information. The blue pixels on the right and bottom of the image are closer to the camera than the target.

ACQUIRE produces a probability of detection that is based on the brightness and size of the target, as well as the brightness of the target's background. Adapting ACQUIRE to a video game or game-like simulation requires specifying exactly what values will be provided to ACQUIRE in all circumstances.

ACQUIRE's requirement for target size information is a bit unusual. The subjective size of the target must be specified to ACQUIRE in terms of its "number of resolvable cycles". Imagine painting the target with alternating black and white bars and asking the observer to report the number of bars. Obviously, as the bars become very fine, the observer will eventually see them as a solid gray and be unable to count them. The number of resolvable cycles of a target is based on the maximum number of bars that the observer can count before they become too fine. This number is then divided by two to represent the number of complete cycles, one cycle including both a white bar and its neighboring black bar.

Consider a target that is represented on the computer screen as a single row of pixels. If we assume that a human observer is placed sufficiently close to the screen, stripes that are one-pixel wide should be resolvable. We estimate the number of resolvable cycles as twice the number of pixels in the target, giving full benefit of the doubt to the observer's eye.

Most targets are not a single pixel wide, of course. For a square target, the number of pixels on an edge is the appropriate number, i.e. the square root of the number of pixels. Following Reece (1996), we use the square root of the number of pixels in all cases, regardless of target shape.

ACQUIRE requires the brightness (irradiance) of both target and background in Watts/m^2. This is problematic for two reasons. First, while we assume we have access to the framebuffer, and thus to the color values of every pixel of an image that is actually or potentially displayed to the screen, those color values are not in units of Watts/m^2. Secondly, while we assume access to all of the pixel color information contained in the framebuffer, ACQUIRE wants only one number to characterize the brightness of the target and background.

Using camera light meters, we have empirically determined that the irradiance of a pixel in Watts/m^2 is well approximated to within a multiplicative constant by the sum of the pixel color values squared. Since ACQUIRE depends only on ratios of irradiance values, we can therefore plug in squared sum-of-color values wherever an irradiance value is called for.

For GBBA, we simply take the average sum of squares of the pixel color values over the all target and background pixels in the mini-render as the input to ACQUIRE. Full details of the algorithm are provided elsewhere (Darken 2007) (Jones 2006).

For each waypoint under consideration as a hiding place or sniper location and for each posture to be assessed, a random subset of waypoints are selected to be camera positions, and the visibility of the figure is assessed from each. The average visibility is then used to annotate the waypoint.

Conclusions and Future Work

We have described an autonomous exploring agent that positions waypoints, discovers which are accessible from which, and annotates them. The annotations include numeric values describing how much can be seen from the waypoint, the amount of cover provided to a character at the waypoint, and how visible a character at that waypoint is. We are continuing to develop our explorer for production use in various training and analysis simulations under development in our lab using the Delta3D game/simulation engine. Our current explorer implementation does not come close to fully exploiting the system architecture. Jumping could be added to the set of actions attempted by the explorer. Since the explorer runs off-line, we believe much more can be done with computer vision-like techniques to enhance it. For example, the 60 degree intervals which the explorer attempts are fairly coarse. It should be possible to use rendered images to hypothesize additional interesting directions to try in order to find, e.g., narrow catwalks at odd angles.

Acknowledgments

The author wishes to thank Bradley Anderegg and the Delta3D development team for technical assistance. This work was partially funded by grants from the Naval Modeling and Simulation Management Office via the Office of Naval Research and the US Army TRADOC Analysis Center Monterey.

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. 2007. “Computer Graphics-Based Target Detection for Synthetic Soldiers”, unpublished manuscript, submitted to Behavior Representation in Modeling and Simulation (BRIMS) 2007.

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.