The Suffering: A Game AI Case Study

Greg Alt

Surreal Software
701 N. 34th Street, Suite 301
Seattle, WA 98103
galt@eskimo.com

Abstract

This paper overviews some of the main components of the AI system for The Suffering, a single-player 1st/3rd-person action/horror game by Surreal Software for the PlayStation 2 (PS2) and XBox consoles (2004). A simpler version was used in the PC and PlayStation 2 versions of Lord of the Rings: The Fellowship of the Ring (2002). The behavior hierarchy, pathfinding, and steering components are described. The AI system was designed to satisfy goals based on lessons learned from previous projects and work within the constraints of developing a commercial title for videogame consoles. The main goals were to have: a modular behavior system able to support a large variety of behaviors, memory-efficient and robust saved games, many distinct NPC types with different styles of movement and combat, fast and robust pathfinding, robust movement and collision, and modular steering behaviors. The goals were largely met, though some issues became apparent in the course of development, primarily difficulties for designers with setting up movement graphs and NPC logic.

Introduction

This paper describes the AI system for The Suffering, a single-player 1st/3rd-person action/horror game by Surreal Software for the PS2 and XBox consoles (2004). A simpler version was used in the PC and PS2 versions of Lord of the Rings: The Fellowship of the Ring (2002).

Because there were many problems extending the AI code from Drakan: Order of the Flame (1999) for Drakan: The Ancients' Gates (2002), it was clear that a complete redesign and rewrite of the AI code was necessary for The Suffering. The main problems were a slow grid-based pathfinding system, an inflexible flat finite state machine for NPC behavior, an ad-hoc monolithic design with all behaviors sharing data, and a lack of a unified component responsible for moving the NPC.

The Suffering required a wide variety of NPC behavior, including melee combat, ranged combat, and fully capable companion NPCs. To make this possible and avoid many of the problems from Drakan, the behavior, movement, and steering systems of the new AI were designed with the goals of having:

- A modular behavior system capable of scaling to a large variety of behaviors.
- Memory-efficient and robust restoring from saved games (with saving allowed at any time).
- Support for human NPCs and 12 significantly different creature NPCs, each with different abilities and different styles of movement and combat.
- Fast, robust pathfinding anytime, for more purposes than just planning movement to a target position.
- Movement without getting stuck on objects or leaving valid terrain.
- A flexible movement system with modular steering behaviors.

Constraints

The constraints for the AI system came from two sources: the limited resources of the target platforms and the scheduling dictated by the commercial game industry.

The hardware constraints for the PS2 were about 1% of the 32 Megs of RAM for all NPCs at any given time and about 10% of the 33 ms available CPU time per frame. These were not hard limits, especially CPU time, as it was acceptable for CPU usage to have occasional spikes of 20% or more. Also, because CPU and memory usage fluctuated based on the complexity of the current scene, there were some tradeoffs between resources for more NPCs and resources for scene geometry in different areas of the game.

The scheduling constraints required an incremental development approach to avoid risk and to avoid temporarily losing ground for frequent milestones. For example, the "first playable" milestone for The Suffering required that the first enemy creature be largely finished within the first 6 months of development. Soon after that, The Fellowship of the Ring needed its AI system to be fully implemented and debugged. Later improvements were added incrementally for The Suffering, which had a much longer development cycle even though both projects started at the same time.

Behavior System

To achieve the goals of scalability in the behavior system and small saved-game sizes, a hierarchical behavior system was developed. The system borrowed some ideas from the hierarchy of actions described in (Atkin et al. 2000).

With this behavior system, all NPCs are controlled by a hierarchical finite state machine, which is represented as a tree of current behaviors. Each behavior is a separate C++ class that contains all necessary dynamic state for the behavior as member variables. Behaviors are allocated when added by their parent behavior, and they are deleted when removed by the parent.

Thus, data for current behaviors only is in memory or saved to a saved-game, creating memory-efficient and robust restoring from saved games. The behaviors are hardcoded in C++, but each behavior has constant parameters specified by designers for information such as specific animations, pause times, and speeds, as well as parameters that control the logic of the behavior.

Each behavior has a limited interface with member functions for startup (taking arguments from the parent behavior), cleanup when removed, repeated update, handling of messages from child behaviors, handling of messages from the rest of the world (like animation events for arming a weapon), and loading and saving of important data.

Behaviors at the top of the tree are abstract and generally use child behaviors to achieve subtasks. The behaviors tend to be small and responsible for just one task. Creating small behaviors allowed greater reuse of behaviors and made debugging easier.

One issue with this style of behavior tree is that all state in a behavior is lost when the behavior is removed. In general this is not a problem, as most data does not need to persist across different instantiations of a behavior. In the few cases where it is necessary, a separate permanent component can be added to the NPC and the behaviors be given access to it. A more general solution, not used in The Suffering, would be to have each behavior have the option of a persistent data class specific to the behavior.

Figure 1 shows an NPC's behavior tree when encountering an enemy while leading the player. The NPC currently has a path to the attack position and is moving there using steering behaviors. Figure 2 shows the NPC's behavior tree after all its enemies are destroyed but before it finds a path to the next waypoint when leading the player.

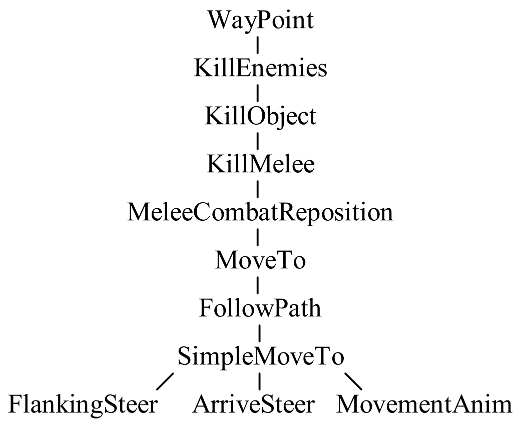

Figure 1. Behavior tree when moving to attack.

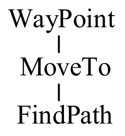

Figure 2. Behavior tree when finding path to next waypoint.

This system allows complex behavior and variety among different NPC types (including human NPCs and 12 significantly different creature NPCs) by allowing alternative behaviors at different levels in the hierarchy. For example, there are several different NPC-specific KillMelee behaviors. The KillMelee behavior is used as a child of the KillObject behavior. It is generally responsible for moving to good positions to attack and for using MeleeAttack behavior to attack a target, though it also handles things like taunting and pausing between attacks.

Having NPC-specific versions means that Slayers can attack from the ceiling, Burrowers can taunt the player character by approaching and looping around him, and Infernas can encircle him with their fire-trails. Specific NPCs are hardcoded to use their specific alternate behavior when, for example, the shared KillObject behavior tries to add the generic KillMelee as a child behavior.

Designers can change the root behavior of an NPC at any time with BehaviorChanger objects that can be triggered by a variety of events. Additionally, they can specify behavior overrides so that a different set of constant parameters is used for a behavior under different circumstances. For example, when a Slayer has his head shot off, he goes berserk and his combat parameters become more aggressive.

In all, there are 109 behaviors, with root behaviors such as KillEnemies, Waypoint, Death, and Conversation. At the bottom there are leaf behaviors such as PlayAnim, SimpleFall, ArriveSteer, PlayLineOfDialogue, FindPath, and FaceTowards.

While this system satisfied the initial goals of flexibility and robustness, several issues came up in the course of development.

The rigid, hardcoded structure of the behavior tree meant that designers didn't have the same benefits of modularity that programmers had. Allowing two NPCs of the same type to have fundamentally different behavior required a programmer to add a parameter to the parent behavior that amounted to "Use Method A or Method B." For example, the Flee behavior could just try to get the NPC as far away from an enemy as possible, or it could try to get the NPC to cower in a safe spot. Having multiple strategies in one behavior made the code more complicated and confused designers.

Another issue was the mechanism for interrupting behaviors. When an NPC is shot and reacts, the current behavior tree is pushed onto a 1-level stack. When the HitReaction behavior is done, the stack is popped and the previous behaviors continue. Often (but not always) the HitReaction behavior leaves the NPC in a state in which the original behavior doesn't make any sense. This meant that code had to be added to many behaviors to do the right thing when restored.

Movement Graph and Pathfinding

The NPC movement graph is similar to the one described in (Hancock 2002) but uses square nodes instead of circles to better cover areas with right angles. Nodes are placed and connected manually by designers. Each node is a square with width, 2D rotation, and height. An edge between two nodes defines a convex volume--the 2D convex hull of the two node squares, aligned to a plane going through the centers of the nodes and extruded vertically about a meter in both directions. Each NPC keeps track of his current edge region and uses A* (Nilsson 1998 and Higgins 2002) to find paths on the fly from the current edge region to a goal edge region. Figure 3 shows a sample movement graph.

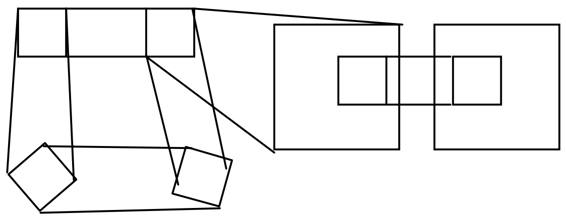

Figure 3. Sample movement graph.

Additionally, this graph is used to mark areas where an NPC is allowed to move. If an NPC attempts to move horizontally outside a valid edge region, he is constrained by a vertical plane of the convex volume, preventing him from leaving valid terrain. This also allows designers to build the movement graph around static objects, preventing NPCs from having to test them for collisions and eliminating the problem of NPCs getting stuck on complicated static objects.

Because the edge regions are often very large, paths are cleaned up using string-pulling, as described in (Tozour 2003), within the movement graph. This straightens paths and shortens them so they are more realistic than they would be with connecting node centers. Steering behaviors and limits for turn speed and acceleration also smooth out movement, by softening turns.

Ceiling movement works just like floor movement but with inverted gravity and up-vector. Transitions between ceiling and floor use precomputed transition points that are associated with edges that go between ceiling nodes and floor nodes and are guaranteed to be lined up with both the ceiling and floor movement graphs. When an NPC has a path that transitions, he moves towards the transition point and begins the transition once he is able to.

While the movement graph is generally static, there is support for toggling specific edges to handle bridges being destroyed or passageways being cleared. There is also support for special types of edges: ceiling/floor transition edges, doors, and ladders. Additionally, there is support for tagging edges to exclude specified NPC types.

Once a path is found, the FollowPath behavior is used move to the target. The type of each edge in the path determines the child behavior used to traverse that edge. SimpleMoveTo is used to move across ordinary floor or ceiling edges, FloorCeilingTransition is used to jump up to the ceiling, LadderClimb is used for ladders, and so on.

To avoid spikes when multiple NPCs try to find paths at the same time, only one path can be found during each frame. Other requests must wait until the next frame. This keeps CPU usage to acceptable levels and prevents the added memory expense of several pathfinding attempts that occur concurrently over the course of a few frames.

The worst-case memory and CPU expense occurs when an NPC tries to find a path to an unreachable goal and the trivial reject test doesn’t work. Generally, if no path is possible, this is detected immediately when the region ID of the start and goal are compared. The movement graph is preprocessed to automatically mark separate disconnected regions using a simple flood-fill algorithm. In some cases, this trivial reject test doesn’t work because an edge is not valid (for example, sometimes two edges are in the same region but the only connection passes through an edge that excludes a specific NPC type or is toggled off by an edge toggler). The CPU and memory expense of this worst case is acceptable and occurs rarely.

The quarry level is a good example of the complexity of movement graphs in The Suffering. This is the largest level, roughly 380 meters by 130 meters. It contains 626 movement graph nodes and 2100 edges, counting each bi-directional edge as 2 edges. Generally, each node is connected to 3 others with bi-directional edges, making an average of about 6 edges connected to each node, though this ranges from a minimum of 2 edges (for dead-end nodes) and a maximum of 16 edges.

As an example of the constraints of scheduling, pathfinding was initially implemented using a simple iterative deepening search (Nilsson 1998) with all edges having equal cost for the "first playable" milestone. As implemented, this algorithm was less efficient than A* and wasn't guaranteed to generate optimal paths, but it provided necessary functionality for early milestones and required just a few lines of code. Later, when higher-priority functionality had been implemented, A* was added as a drop-in replacement for iterative deepening.

This system satisfied the original goal of speed and allowed NPCs to find paths frequently. Since pathfinding was so fast, it was used not only to determine how to navigate to a goal position but also to try to navigate a specified distance away from a position; to determine actual path length instead of straight-line distance for some behavior logic; and to determine whether the player was ahead or behind a friendly NPC on their waypoint path.

Constraining NPCs to the movement graph also satisfied the goals of not getting stuck on static objects, not moving into invalid terrain, and not having to perform expensive collision tests against static objects outside the movement graph.

The main problem with this system is that it was very difficult and time-consuming for designers to manually set up and debug. A designer would spend roughly 8 hours setting up and debugging a movement graph for an indoor level and up to 20 hours for a large outdoor level. This is largely due to the primitive interface and a lack of good 3D visualization tools. Ideally, this process would be more automated, but it is not clear how feasible this would be, given that designers would still need to be able to adjust and debug the automated graph.

Steering and Collision

Steering was implemented as a collection of behaviors that are simultaneously children of the SimpleMoveTo behavior for an NPC. These are essentially the same as the steering behaviors included in (Reynolds 1999): Arrival, Evade, Wander, ObstacleAvoidance, UnalignedCollisionAvoidance, Containment, and Separation. Another steering behavior, Flanking, was added. This caused an NPC to try to approach an enemy from the side or behind. Each steering behavior outputs an acceleration vector to the low-level movement system. The movement system adds these together and applies the maximum acceleration limit before generating and limiting the new velocity. The NPC is then moved and animated based on this velocity. This system is described in more detail in (Alt and King 2003).

The NPC's collision system is mostly the same as the player's. A stack of spheres is used to detect collisions, and the lowest sphere is used to detect the ground. The main difference is that NPC collision detection is optimized to consider only collisions with objects that intersect a movement graph edge region--each edge region has a list of objects that is autogenerated for easy lookup.

The steering and collision systems satisfied the primary goals of having realistic movement without getting stuck on static objects or walking out of the level, but there were still some issues with getting stuck on dynamic objects, which required designers to take steps to prevent. Also, the large variety of NPCs were difficult to set up, because the various designer-specified parameters needed separate tweaking for each NPC type. A common problem was handling tradeoffs in which tweaking a parameter to improve the look of an NPC's movement caused problems like overshooting narrow doorways, while tweaks to improve functionality made for otherwise less realistic movement.

The ObstacleAvoidance and CharacterAvoidance steering behaviors were also problematic because it was difficult to tune their acceleration weights, and the initial implementation for searching for nearby objects was computationally expensive. This led to designers turning off these steering behaviors for many of the NPCs.

Conclusion

To a large extent, the AI system for The Suffering satisfied the original goals. The modularity of the code resulted in much fewer bugs than previous projects and with a significant increase in functionality and diversity of NPCs.

The primary problems encountered were the difficulty for designers to set up NPC logic and movement graphs. For the movement graph, this was due to the fact that the graph had to be created and tweaked manually with a less-than-ideal interface and visualization tools. For NPC logic, the problem was that the modularity of the code didn't directly translate into a modular interface for designers.

Future enhancements may help prevent these problems. The difficulty in creating movement graphs can be significantly lessened with a level editor interface that better shows the movement graph the NPCs will use. Also, to aid in debugging movement graphs, more in-game debug tools can be added. Finally, to prevent the behavior tree from being hardwired, alternate versions of some behaviors can be made available to designers regardless of NPC type.

References

Atkin, M., King, G., Westbrook, D., and Cohen, P. 2000. Some Issues in AI Engine Design, *Artificial Intelligence and Interactive Entertainment: Papers from the 2000 AAAI Spring Symposium:* AAAI Press.

Hancock, J. 2002. Navigating Doors, Elevators, Ledges, and Other Obstacles, *AI Game Programming Wisdom:* Charles River Media.

Higgins, D. 2002. Generic A* Pathfinding, *AI Game Programming Wisdom*: Charles River Media.

Nilsson, N. 1998. *Artificial Intelligence: A New Synthesis*: Morgan Kaufmann.

Tozour, P. 2003. Search Space Representations, *AI Game Programming Wisdom 2.*: Charles River Media

Alt, G., and King, K. 2003. Intelligent Movement Animation for NPCs, *AI Game Programming Wisdom 2*: Charles River Media

Reynolds, C. W., 1999. Steering Behaviors for Autonomous Characters, *GDC 1999 Conference Proceedings*, 763-782: Miller Freeman Game Group *http://www.red3d.com/cwr/steer/*