Efficient, Realistic NPC Control Systems using Behavior-Based Techniques

Aaron Khoo, Greg Dunham, Nick Trienens, Sanjay Sood

Computer Science Dept., Northwestern University
1890 Maple Avenue
Evanston, IL 60201
khoo@cs.northwestern.edu, {gdunham, n-trienens, s-sood3}@northwestern.edu

Abstract

Computer games are an application in which the perception of intentionality is often more important than intentionality itself. Players often attribute more intelligence to non-player characters than is actually warranted. Therefore, rather than investing more AI complexity in the system, we propose an approach which refines and extends the finite-state machine techniques already prevalent in commercially available games. Taking advantage of the similarities between robots and NPCs in dynamic environments, we describe two NPC control systems that uses behavior-based techniques.

Introduction

There has been a recent surge in interest in the use of computer games as platforms for AI research (Laird and van Lent 00). However, most research into the AI of non-player characters, or NPCs, in computer games has focused on increasing the underlying complexity of the agents. The motivation has been to develop increasingly 'human-like' bots which replicate the thought processes of the human player in the same given situation. For example, the QuakeBot agent (Laird and van Lent 99) developed at the University of Michigan has recently incorporated usage of predictive capabilities and learning (Laird 00). QuakeBot is able to anticipate a player's future actions through observation of the player's current behavior and react accordingly. Its developers have invested a lot of time and energy incorporating an "expert" human player's tactical thought processes into the QuakeBot.

Issues with Cognitive Simulation

However, there are two potential problems with this human-level approach. First, this method can be expensive. Many modern computer games are performance driven. There is typically only a small percentage of the CPU available for AI relative to the amount devoted to graphics. Traditional AI reasoning systems allow arbitrarily sophisticated representations, but usually at the cost of computational complexity. While that complexity does not always involve exponential-time or undecidable computations, it does generally involve highly serial computations operating on a large database of logical assertions. Most current NPC implementations in AI games research, such as QuakeBot or GameBots (Adobbati et al 01), assume a separate CPU for each NPC.

Second, it is unclear that this increased complexity of human-like cognition has added much to the final playability of the product. That is, the player might not notice any difference in the outward behavior of the NPC. In some cases, the underlying intricacy might actually produce strange or annoying NPC behavior from the player's perspective since the player is not always able to discern the NPC's intricate deliberations. On the other hand, people are often willing to ascribe far more intelligence to very simple but obvious actions than is often warranted. One well-known anecdotal example is taken from the Half-Life game. During the course of the Half-Life single player scenario, the player encounters small groups of hostile marines. In the course of attacking the player, the marines will yell things such as "cover me!". Although the NPC marines are acting as independent entities and the yells are randomly generated, players often relate observed instances of 'skillful AI teamwork' in groups of attacking marines. Researchers at the Oz project have also noticed a similar phenomenon occurring as they attempt to build believable agents that exhibit emotions through simple goal-driven behavioral constructs (Bates 94).

Furthermore, a recent case study (Laird and Duchi 00) seems to suggest that, at least for NPCs in a first-person shooter, decision time and aiming skill are key to the perception of humanness. The first parameter implies that the control program should be made as efficient as possible to avoid any perceivable slowdown in the NPC's reaction time, while the latter is a parameter that does not require human-like cognition to control. In fact, NPCs often have perfect information about the world, so varying their aiming skill simply involves adding some level of noise to the appropriate sensors.

Ultimately, it is possible to end up constructing an overly complicated system that is underappreciated by the player. We would argue that in most cases of NPC development, the generality that traditional symbolic reasoning systems bring is overkill. Instead, we should aim for efficient systems with a simple underlying architecture that allows us to produce most behaviors that a human observer can relate to, while avoiding overtly stupid actions that betray their underlying simplicity. In the end, if an NPC looks, talks and walks like a duck, then it will probably be perceived as a duck, regardless of its actual inner workings.

Behavior-based Techniques

NPC control systems in commercially available games are generally based on finite-state machine techniques. While efficient, this approach is usually criticized as being unwieldy and results in NPCs which are unrealistic. We admit that, in general, current NPCs leave much to be desired, with limited response capabilities and simplistic behaviors. However, we argue that this is not a consequence of the finite-state machine approach per se, but rather is a result of the ad hoc AI development model utilized by most game companies.

We believe that realistic yet efficient NPCs can be developed by using a methodology based on well-understood behavior-based techniques. NPCs that reside in a dynamic environment, such as a first-person shooter (FPS), real-time strategy game (RTS) or a massively multiplayer game, live in a continually changing world where the NPC is not the only change effector. The NPC's modeling systems must constantly track the incoming sensory data, and its control systems must be ready to alter plans and actions to suit the changing world.

Behavior-based systems (Arkin 98) solve these problems very effectively. In their purest form, behavior-based systems divide sensing, modeling, and control between many parallel task-achieving modules called behaviors. Each behavior contains its own task-specific sensing, modeling, and control processes. Behaviors tend to be simple enough to be implemented as feed-forward circuits or simple finite-state machines, allowing them to completely recompute sensor, model, and control decisions from moment to moment. This, in turn, allows them to respond immediately to changes in the environment. Unlike purely stimulus-response systems, it is possible for behavior-based systems to maintain state information.

Using a behavior-based approach gives us a methodology for building our NPCs systematically while maintaining great efficiency. We can add one observably intelligent behavior at a time, eventually building an NPC which appears very sophisticated but is still utilizing a finite-state machine as a control system. In the following section, we will examine Ledgewalker, a set of behaviors that we developed to control bots within the Half-Life game. Then, we conclude with a section on some possible future directions.

Implementation

Half-Life and FlexBot

Half-Life is a popular first-person shooter game developed by Valve Studios. We chose this game as our development platform because the game engine has been open source for some time, resulting in easy accessibility and an available online community of veteran programmers who serve as helpful mentors.

We developed an NPC or 'bot' SDK for the Half-Life game written in C++. Named FlexBot, the SDK allows designers to create NPCs through a 'fake client' interface and provides a set of sensors and actuators which a designer uses to program the bots. The sensors and actuators reside in a DLL which talks to the Half-Life engine. The designer is responsible for writing FlexBot control programs, which are separate DLLs that talk to the FlexBot interface DLL, as shown in figure 1 below.

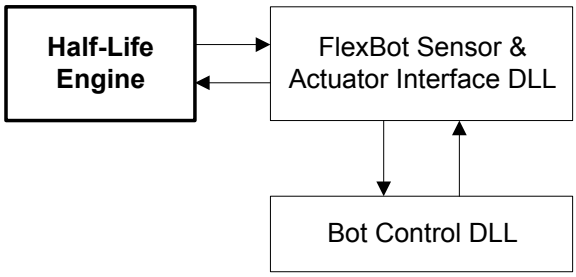

Figure 1 : FlexBot Control Flow

Although we have focused on behavior-based systems, FlexBot control programs do not have to adhere to any particular architecture. It is intended for use as a general bot development platform for Half-Life. For more information on FlexBot, including a download of the latest version, see http://flexbot.cs.northwestern.edu.

Generic Robot Language

The behavior-based control programs we created for Half-Life bots utilize the sensor and actuator interface provided by the FlexBot SDK. However, we realized that writing C++ code to implement ever more complicated finite state machines would be unfeasible. We wanted to take advantage of higher-level, Lisp-style functional programming techniques familiar in AI applications.

Therefore, the bot control programs were written in Generic Robot Language or GRL (Horswill 99), a programming language originally designed for robot

development. GRL is a simple language that extends traditional functional programming techniques to behavior-based systems. It is an architecture-neutral language based on Scheme for describing reactive control systems. GRL provides a wide range of constructs for defining data flow within communicating parallel control systems and manipulating them at compile time using functional programming techniques. Most of the characteristics of popular behavior-based robot architectures can be concisely written as reusable software abstractions in GRL. This makes it easier to write clear, modular code, to "mix-and-match" arbitration mechanisms, and to experiment with variations on existing techniques. Code written in GRL can be compiled to a variety of languages, including Scheme, Basic and C++.

Ledgewalker

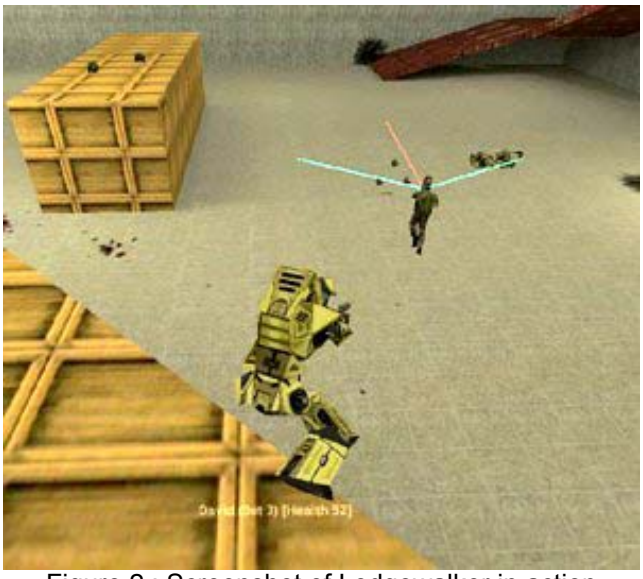

Figure 2 : Screenshot of Ledgewalker in action

Ledgewalker is the codename for our original Half-Life bot. The Ledgewalker control code consists of a series of behaviors executed in parallel. Arbitration is achieved through a priority stack with a slight twist. The behaviors are checked starting with the behavior with the highest priority, and the first active behavior is taken to be the output of the bot for the current processing cycle. Figure 3 shows the current set of Ledgewalker behaviors. Most of the behaviors shown in figure 3 are self-explanatory, so we will not go into details for each behavior.

There are two interesting points to note about figure 3. First, the *shoot* behavior supercedes all others, i.e. if the bot sees an enemy, it will drop everything it is doing to attack. Second, notice that that the rest of the behaviors are divided into two separate stacks, enclosed by a dotted rectangle. This is because the two stacks can be run in parallel, e.g. the bot may reload its weapon while it is wandering or turning to check on a noise it heard. Therefore, Ledgewalker chooses the most active behavior from the left and right stacks per processing cycle, and then combines them into a single action vector through a union operator. This final action vector is then sent to the FlexBot DLL, assuming the shoot behavior is turned off.

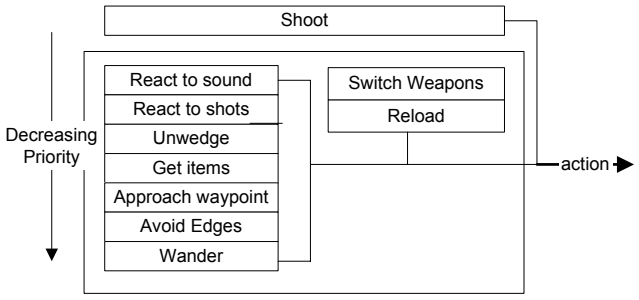

Figure 3 : Ledgewalker Behavior Stack

Groo

The latest incarnation of our Half-Life bot is known as Groo, named for the Sergio Aragonés comic character of the same name. Groo is a brutish, fierce but foolish barbarian who often gets into trouble because of his propensity for violent action above civil dialogue. This somewhat aptly describes the behavior of our bots.

The design for Groo came about as we realized the limits of Ledgewalker's original design. The FlexBot interface affords eleven actuators :

- rotate
- translate
- strafe
- pitch
- shoot
- fire secondary
- jump
- switch weapon
- reload
- duck
- use

Furthermore, the interface expects a value for each actuator on each program cycle. Ledgewalker's behaviors initially attempted to control all the actuators simultaneously. However, we realized that some actuators should be controlled independently of each other. For example, as figure 3 shows, the bot can reload or switch weapons while performing some movement. After more consideration, we realized that, in fact, most of the actuators should be controlled independently. We ultimately divided the actuators into two groups : those that controlled movement, and those that didn't. The first group consists of rotate, translate and strafe. Our reasoning is that the bots should be able to perform any of the other actions while maneuvering.

This approach to Groo's design has two advantages. First, it is a more modular design. Rather than attempting to control all eleven actuators simultaneously, we have divided the control into separate smaller modules that are

more concise and readable. The modularity also allows us to easily "shut off" any superfluous actuators if necessary during debugging, making the development process easier. The second advantage of Groo's design is the emergent properties that arise from the interaction of the actuators. Since most of the actuators are now independently controlled, the bot can react to situations with a greater repertoire of actions. This newfound diversity of reactions makes for a bot that plays in a more "natural" manner; at least the bot's behavior is now less overtly predictable.

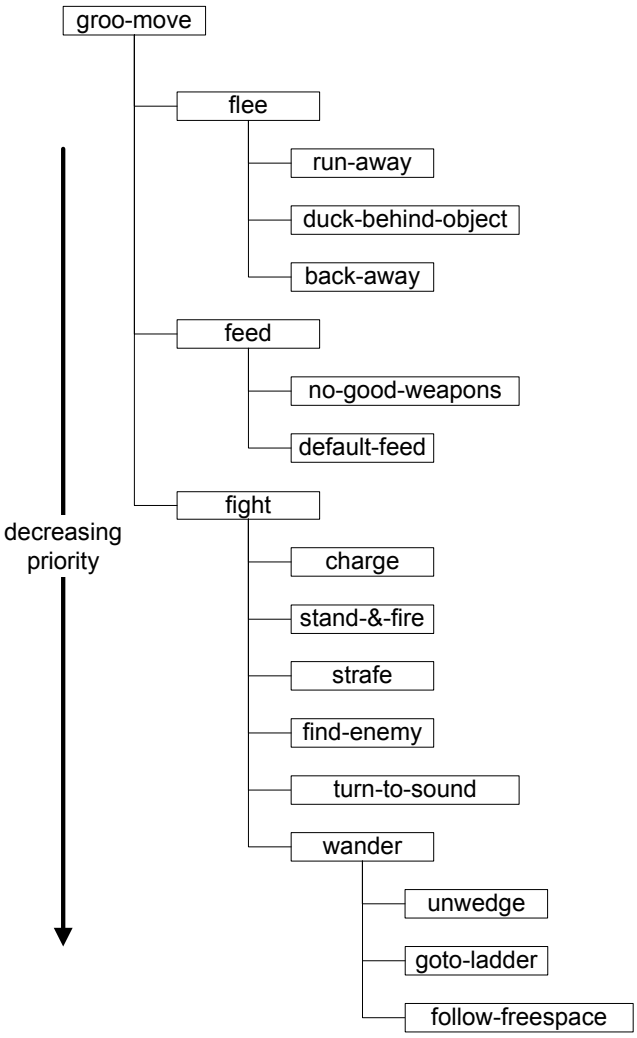

Figure 4 : Prioritization of Groo Movement Behaviors

Movement Behaviors As discussed earlier, Groo's movement consists of three actuators, i.e. rotate, translate and strafe. The behaviors that control these actuators are known as the *movement behaviors*. The arbitration mechanism used to determine which movement behavior is allowed to execute during the current program cycle is still a priority stack. However, the individual behaviors are now further divided into groups, forming the Tinbergen hierarchy structure (Tinbergen 51) shown in figure 4. The higher the behavior is on the stack, the higher its priority. Hence, Groo's sense of self-preservation overrules his killer instinct and the bot will try to flee if necessary. The feed behavior describes Groo's attempt to pick up items such as ammo and weapons. If Groo has no good weapons in its inventory, *no-good-weapons* will force it to grab a weapon if it is in sight. *Default-feed* will not activate as long as Groo sees an enemy and has a decent weapon in its inventory.

Remaining Actuators The remaining eight actuators each have their own behaviors. The behaviors for these actuators are mostly a collection of rules of thumb gleaned from domain experts. For example, the behavior for *shoot* actuator is defined as :

```
shoot =
  (and
    facing-enemy?
    not-fire-secondary?
    (or  weapon-clip-not-empty?
         (= current-weapon crowbar))
    (or  (and enemy-long-range?
              current-weapon-long-range?)
         enemy-short-range?
         being-shot?))
```

On every program cycle, the movement behaviors produce outputs for the rotate, translate and strafe actuators. The outputs from the behaviors governing each of the remaining eight actuators is combined with the movement actuators using a union operator, and the resulting action vector is sent to the FlexBot DLL on every program cycle, as shown in figure 5.

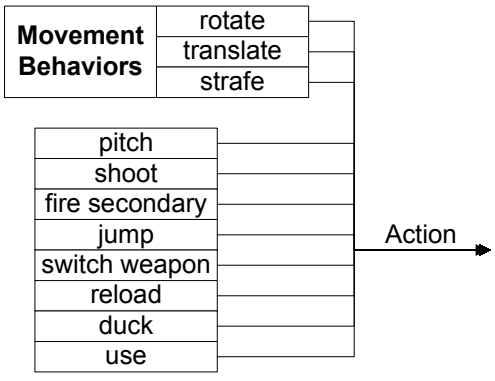

Figure 5 : Output Actuator Vector

Results

The compiled machine code for both Ledgewalker and Groo are extremely efficient and stable. The bots run concurrently with the Half-Life game server on the same

physical machine. We have successfully run 32 bots, the maximum number supported by the game engine, on one machine under dedicated server mode, and 16 in listen server mode. A dedicated server has no graphical processing duties. Its only job is to provide a multiplayer game that external players can connect to. A listen server, on the other hand, is both a server and client, meaning that a player is physically playing on it while it is also serving external connections. In the latter case, the added load of the graphics processing created too much lag as the number of bots in the game increased.

The game engine updates its world model, including any bots, once every 0.1 seconds. We have observed our bots individually consuming 3ms per processing cycle. Therefore, it is only using 0.3% of the CPU per bot. Ultimately, the game engine is the bottleneck, not the AI. Otherwise, the CPU could run over a hundred bots per game. Each instance of the Ledgewalker bot only uses 436 bytes of memory during runtime, yielding a very small footprint. The Ledgewalker code base was about 900 lines of GRL code, which compiled to 1963 lines of C++ code. The numbers were fairly similar for Groo, which had 800 lines of GRL code and 1600 lines of C++ code. Finally, the mean time to failure of the system is very long. We have successfully run the bots in dedicated server mode continuously for over two weeks. In fact, during one test run, the variable storing the number of kills for a bot actually overflowed.

While we have not run any empirical studies similar to (Laird and Duchi 00), anecdotal evidence seems to indicate that Ledgewalker has succeeded in its goal of realistic behavior to a certain extent. Some experienced Half-Life players, including ourselves, have playtested the Ledgewalker and Groo bots. The system was also demonstrated at IJCAI-2001, where it was played by several participants. In general, the reaction was positive. Most players agreed that the bots did exhibit behaviors that one would associate with a human player.

Conclusions and Future Work

The Ledgewalker and Groo bots are an attempt to create an efficient control system that can exhibit human-like behavior. Instead of using a traditional AI reasoning system, we endeavored to utilize behavior-based techniques from robotics. These techniques allow an agent to track changes in the dynamic environment quickly, and react within O(1) time to any relevant events. Behavior-based systems are extremely efficient, as evidenced by our ability to run up to 32 bots simultaneously and the small memory footprint per bot. Anecdotal evidence appears to suggest that Ledgewalker behaves like a human player, but this needs to be verified through an empirical experiment.

However, behavior-based systems introduce their own set of issues. Their greatest weakness is the the use of simple propositional representations, which makes most reasoning and planning tasks both difficult and clumsy. We propose to incorporate variable binding and quantified inference to Ledgewalker using role-passing (Horswill 98, Horswill and Zubek 99), a deictic representation technique. This extension to traditional behavior-based systems maintains the efficiency of those systems while allowing us to express control reasoning in a more reusable, taskable fashion. We also intend to extend Ledgewalker into a multi-agent research testbed utilizing HIVEMind, a multi-robot architecture presently implemented on a robotic team performing a search task.

Reference

R. Adobbati.; A.N. Marshall; A. Scholer; S. Tejada; G. A. Kaminka.; S. Schaffer; C. Sollitto(2001) Gamebots: A 3D Virtual World Test-Bed for Multi-Agent Research, In Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada.

R.C. Arkin(1998) Behavior-based Robotics. MIT Press. Cambridge, MA.

J. Bates(1994) The Role of Emotion in Believable Agents. Communications of the ACM, vol. 37, no. 7, pp. 122-125

I. Horswill(1998). Grounding Mundane Inference in Perception. In Autonomous Robots, 5, pp. 63-77.

I. Horswill(1999). Functional programming of behavior-based systems. In Proc. IEEE International Symposium on Computational Intelligence in Robotics and Automation

I. Horswill and R. Zubek (1999) Robot Architectures for Believable Game Agents. In Proceedings of the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games.

O. Khatib(1985) Real-time Obstacle Avoidance for Manipulators and Mobile Robots, Proceedings of the IEEE International Conference on Robotics and Automation, St. Louis, MO, pp. 500-05

B. Krogh(1984) A Generalized Potential Field Approach to Obstacle Avoidance Control, SME-RI Technical Paper MS84-484, Society of Manufacturing Engineers, Dearborn, Michigan.

J.E. Laird(2000) It Knows What You're Going To Do : Adding anticipation to a QuakeBot AAAI 2000 Spring Symposium Series : Artificial Intelligence and Interactive Entertainment, March 2000 : AAAI Technical Report SS00 –02

J.E. Laird and J.C. Duchi(2000) Creating Human-Like Synthetic Characters with Multiple Skill Levels: A Case

Study Using the Soar Quakebot AAAI 2000 Fall Symposium Series : Simulating Human Agents, November 2000 : AAAI Technical Report FS-00-03

J.E. Laird and M. van Lent(1999) Developing an Artificial Intelligence Engine. In Proceedings of the Game Developers Conference, San Jose, CA 577-588.

J.E. Laird and M. van Lent(2000) Human-level AI's Killer Application : Interactive Computer Games. AAAI Fall Symposium Technical Report, North Falmouth, Massachusetts, 2000, 80-97.

N. Tinbergen(1951) The Study of Instinct. Clarendon Press, Oxford, England.