Applying Inexpensive AI Techniques to Computer Games

Aaron Khoo and Robert Zubek, *Northwestern University*

Groo and tt14m are two systems that use simple and computationally inexpensive AI mechanisms to produce engaging character behavior for computer games, while remaining within performance constraints of modern game development.

Modern computer games are highly sophisticated in their simulation of an artificial game world. They present the player with beautifully rendered environments, often accompanied by a complex and consistent world physics model. Increasingly, these games also use intelligent characters that can help or hinder the player and interact with the world and each other. Developers are using artificial intelligence to develop more engaging games, and the complexity of this game AI has become an increasingly important selling point.

However, developing these intelligent systems is a complicated endeavor. Modern computer games impose tight computational constraints: graphics and world physics simulations consume most of a CPU's cycles, leaving only a small fraction available for the AI subsystem to control game agents. Some genres (such as first-person shooters) feature highly dynamic environments that update many times per second, requiring the AI system to be highly responsive to unexpected events. Thus the AI engine must continuously manage a potentially large number of agents while using only a small fraction of processor time.

Our approach concentrates on using existing inexpensive techniques to develop computer game characters. We used two such techniques to develop agents for deathmatch games in the first-person shooter genre. The first system, Groo, engages in intelligent tactical behavior using a fairly simple and static behavior network. The second system, trash-talking 14-year-old moron (tt14m), uses simple text processing to attempt engagement in the social aspects of the game Counter-Strike. The techniques used in both systems are computationally inexpensive and easy to implement, which addresses the constraints found in most game development.

Groo

The Groo project attempts to create an efficient agent, or *bot*, that plays a first-person-shooter deathmatch game in a tactically intelligent manner. Our objective was to model an average deathmatch player's basic tactical skills. Researchers on the QuakeBot project performed some related work in this area, with focus on improving the bot's cognitive abilities by incorporating predictive capabilities and learning.[1] Our project places heavier emphasis on efficiency of simple mechanisms.

As we mentioned earlier, the typical modern game leaves little time for AI processing, so game AI must be efficient. Our hypothesis is that a reactive system based on behavior-based techniques can model an average first-person-shooter player's skills while still being efficient enough for game developers.[2] We are not arguing against the eventual addition of more sophisticated cognition to the bots, but we make two observations.

First, existing AI techniques that implement complex cognition are computationally expensive relative to available CPU time. Traditional symbolic reasoning systems let you manipulate arbitrarily sophisticated representations but require highly serial computations operating on a large database of logical assertions. Updating and then reasoning over a symbolic knowledge base for a complex and highly dynamic environment in the small remaining time slice is a difficult challenge.

Second, a recent case study seems to suggest that for bots in a first-person shooter game, decision time and aim are key to the perception of skill.[3] The former quality is directly tied to our focus on efficient AI processing; the latter is not a function of higher-level cognition.

1094-7167/02/$17.00 © 2002 IEEE

Behavior-based techniques

The system we constructed uses behavior-based action selection techniques gleaned from the robotics field to perform actions in the game world. These techniques are specifically designed to control systems that reside in complex worlds where the agent is not the only change effector. Agents in these environments must constantly track incoming sensory data, and their control systems must be ready to alter plans and actions to suit the changing world.

In their purest form, behavior-based systems divide sensing, modeling, and control among 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 implement as feed-forward circuits or simple finite-state machines, which lets them completely recompute sensor, model, and control decisions from moment to moment. This, in turn, lets them respond immediately to changes in the environment. Although behavior-based systems do not have the representational power of traditional symbolic systems, they are computationally efficient. These techniques can generate sufficiently rational behavior in this domain.

Implementation

Half-Life is a popular first-person shooter game from Valve Studios. We chose it as our development platform for Groo because its game engine is open source, resulting in easy accessibility and an available online community of veteran programmers who could serve as mentors.

Working with the team at Northwestern, we started by developing a bot software developer's kit for Half-Life. Named FlexBot, the SDK is written in C++, lets designers create nonplayer characters through a fake client interface, and provides a set of sensors and actuators that a designer uses to program the bots. The sensors and actuators reside in a dynamic link library that talks to the Half-Life engine. The designer is responsible for writing FlexBot control programs—separate DLLs that talk to the FlexBot interface DLL (see Figure 1). FlexBot control programs do not have to adhere to any particular architecture and are intended for use as a general bot development platform for Half-Life (see http://flexbot.cs.northwestern.edu).

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

Therefore, we wrote the bot control programs in the Generic Robot Language,[4] a programming language originally designed for robot development. GRL is a simple, architecture-neutral language that extends traditional functional programming techniques to behavior-based systems. GRL provides a wide range of constructs for defining data flow within communicating parallel control systems and for manipulating it at compile time using functional programming techniques. We can concisely write most of the characteristics of popular behavior-based robot architectures 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. We can compile code written in GRL to many languages, including Scheme, Basic, and C++.

Architecture. The Groo system's design features actuators provided by the FlexBot SDK. We can independently control most of the 11 actuators. For example, the bot can maneuver using the **rotate** and **translate** actuators while simultaneously reloading or switching weapons.

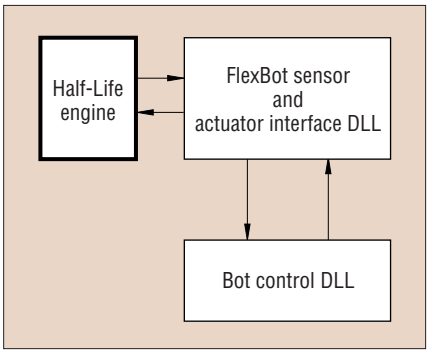


Figure 1. The FlexBot architecture design.

We ultimately divided the actuators into two groups: those that do and do not control navigation. The first group consists of **rotate**, **translate**, and **strafe** (also known as **move-sideways**); the behaviors that control them, *navigation behaviors*, maneuver Groo through the world. Groo senses its immediate surroundings through nine range sensors arranged in an evenly spaced semicircle in front of it. Other sensors include **nearest_visible_enemy**, **nearest_visible_teammate**, **distance_to_player**, **pitch_to_player**, **yaw_to_player**, and **nearest_item**.

A significant difference between Groo and other bot implementations is the lack of explicit path planning: Groo does not depend on traditional waypoints to perform navigation. Figure 2 shows the navigation behaviors

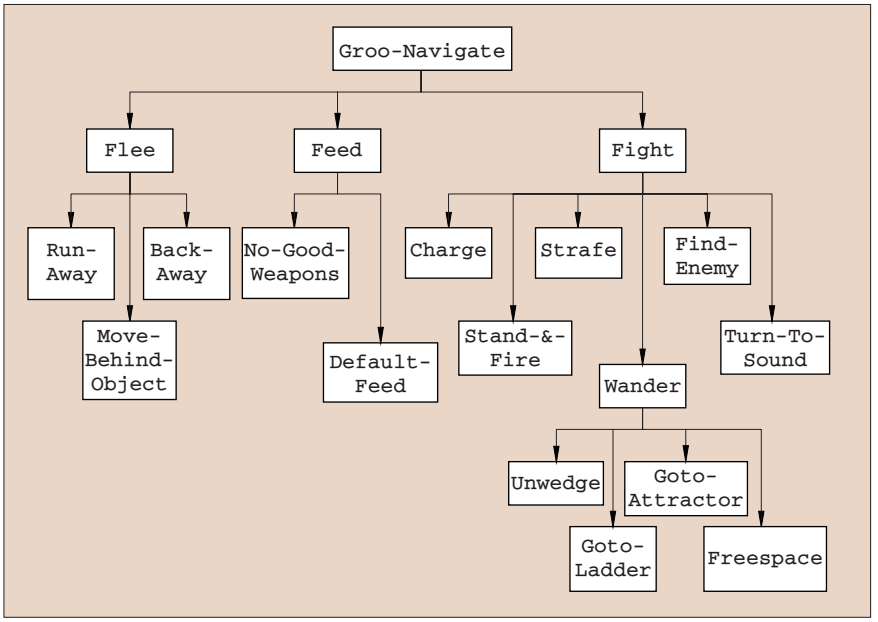


Figure 2. Groo navigation behaviors. On every program cycle, the navigation behavior with the highest activation level wins and gets to dictate the Groo bot's current *rotate*, *translate*, and *strafe* values.

```
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?))
```

Figure 3. Definition for the *shoot* actuator behavior.

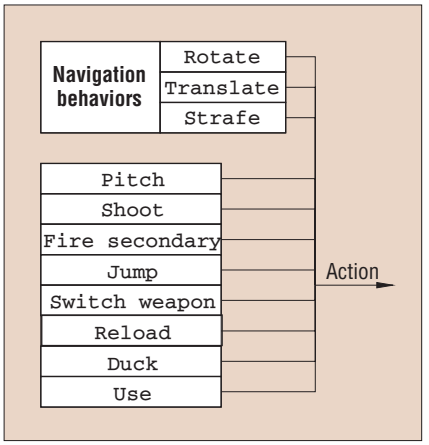


Figure 4. Combining outputs from individual behavior sets to form a single action vector that is sent to the Half-Life engine.

that control Groo's movement through the world. These behaviors are based on a Tinbergen hierarchy and are purely reactive.[5] The Groo bot runs the navigation behavior with the highest activation level on every program cycle. Most behaviors are self-explanatory, but some interesting ones include

- **Goto-attractor**. Although Groo does not use traditional waypoints, attractors draw the bot closer to certain locations such as doorways. These attractors act like a potential field; when Groo is close, it "senses" the attractor and is pulled toward it. After arriving at the current attractor, Groo "forgets" about it for a little while to prevent being stuck at one location. No explicit connection exists between the attractors, so we cannot use them for path planning.
- **Move-behind-object**. If Groo is being shot and wants to take cover, **move-behind-object** attempts to strafe or move Groo sideways behind a nearby object.
- **Wander** (also known as the pick-a-fight behavior). When Groo has nothing better to do, it wanders around the map randomly looking for someone to fight.

The remaining actuators each have their own set of controlling behaviors—for example, the **pitch** actuator has a set of controlling behaviors independent of the **jump** actuator's behaviors. Figure 3 defines the **shoot** actuator's behavior.

On every program cycle, the outputs from each set of behaviors are combined into a single action vector before being sent to the FlexBot DLL (see Figure 4). Interestingly, **jump** is not considered part of the navigation behaviors. This is because **jump** is only used in particular situations, often during shootouts (to confuse opponents) or when the bot is stuck (to leap out of tight spots). We did not attempt to implement behaviors where Groo jumps onto tall objects such as boxes, primarily because of the difficulty in sensing such objects. **Jump** is also not coordinated with any of the navigation actuators.

Rather than attempting to control all 11 actuators simultaneously, we divided the control into separate smaller modules that are more concise and readable. This modularity also lets us easily shut off superfluous actuators if necessary during debugging, which facilitates the development process. Figure 5 shows the Groo behaviors in action.

Results. Groo's compiled machine code is extremely efficient and stable. The bots can run concurrently with the Half-Life game server on the same physical machine. We have successfully run 32 bots (the maximum number the game engine supports) 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 to which external players can connect. A listen server, however, is both server and client, meaning that a player physically plays on it while it serves external connections. In the latter case, the added graphics-processing load created too much lag as the number of bots in the game increased. Our test machine was a 450-MHz Pentium II with 384 Mbytes of RAM and a 16-Mbyte Riva TNT graphics card.

The game engine updates its world model, including any bots, once every 100 ms. We observed our bots individually consuming approximately 0.3 ms per processing cycle, thus our system only used 0.3 percent of the CPU's cycles per second per bot. Ultimately, the bottleneck was in the game engine, not the AI; the CPU could potentially run over 100 bots per game. The Groo code base was approximately 800 lines of GRL code (including comments) and compiled to 1,200 lines of C++ code. Each instance of the Groo bot only uses 52 bytes of static data memory during runtime, yielding a small footprint.

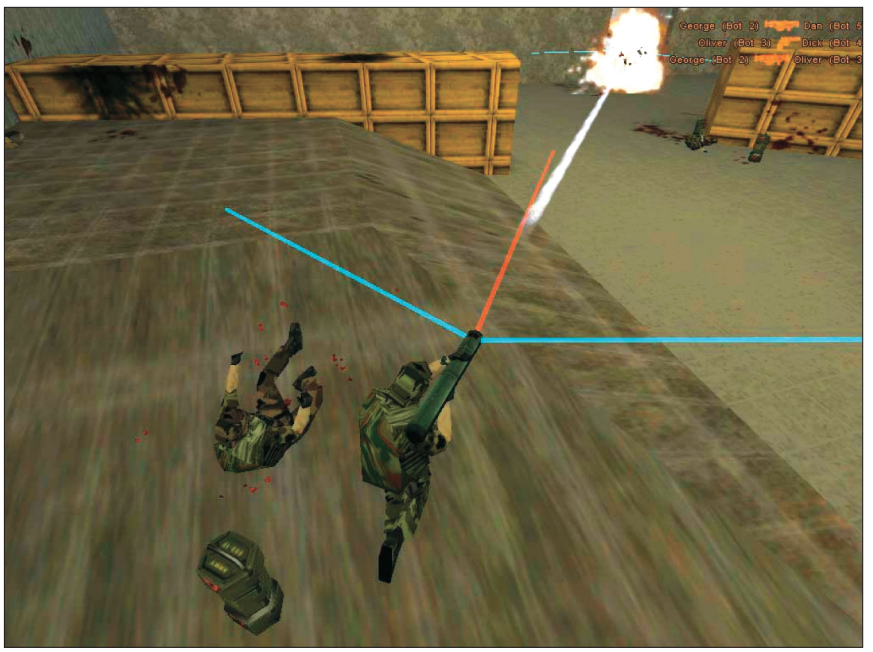


Figure 5. Groo in action. The two bots in the foreground run the Groo behaviors: the one on the left has just been eliminated while the one on the right is firing a rocket launcher. The lines protruding from the latter bot indicate its view field.

The mean time to system failure is also long. We successfully ran the bots in dedicated server mode continuously for over two weeks. During one test run, the variable storing the number of kills for a bot actually overflowed.

Anecdotal evidence seems to indicate that Groo has succeeded in its goal of realistic behavior to a certain extent. Some experienced Half-Life players (including us) have tested the Groo bots, and we also demonstrated the system at IJCAI-2001, where several participants played it. Reactions were positive: most players agreed that the bots exhibited behaviors that they would associate with a human player. The Groo system will be on display as one of the Intelligent System Demonstrations at AAAI-2002, where we hope to gather more user data.

tt14m

Although Groo presents the use of efficient mechanisms for tactical engagement, we now turn to a different aspect of deathmatch gameplay—the competition between individual players.

An important attraction of many competitive computer games lies not in the gameplay's technical challenges but in the social involvement it offers. Given the choice between playing a deathmatch game against bots or humans, gamers characteristically prefer to play human opponents. This is unsurprising to anyone familiar with multiplayer games, but it is interesting to consider why this is so.

On top of immediate, purely technical gameplay, multiplayer games offer the social engagement of participating in a competition. Deathmatch games are group activities that involve teams of players competing against each other in contests of skill with a clear determination of the winner. Emotional involvement in the game is surprisingly high and quite visible—players care about victory and are vocal in expressing their emotions through the game's chat medium. When they lose a game, they express frustration and pain; when they win, they brag and boast loudly. This emotional involvement is not merely a side effect of the players' being human—rather, it fuels much of their enjoyment of the game.[6]

Bots cannot help but be thrown into this social gameplay, but they are utterly incapable of properly participating in it. Their inability to treat the game as a competition makes them technically challenging but not nearly as enjoyable as human opponents. Unlike a human player, a computer character feels no joy in winning and no humiliation in losing. In the end, the player knows a bot simply does not care, which makes beating it a thankless task. We anticipate that creating bots that mimic human players' involvement in the game would make them more appealing as opponents.

As a starting point, we developed a cheap and efficient system that attempts to model the two most common, and arguably simplest, characteristics of players' social interactions: displays of emotional involvement and verbal posturing. Our objective for this project was to build a bot modeled on the stereotypical juvenile delinquents who frequent these online games, hence the project's name. The characteristics of communication in online multiplayer games are particularly advantageous in helping us model this demographic.

Exploiting the domain

Creating a simple chatting bot does not require solving the full natural language problem. The gaming domain limits the problem's scope in several helpful ways.

Online conversations in these games are generally neither deep nor well structured. Chatting in online games resembles Internet Relay Chat in that the conversations are often

- Disconnected—topics change frequently, so it's easy to lose track of who is talking to whom and about what.
- Layered—it's not unusual for one person to participate in more than one thread of conversation simultaneously; conversely, it is entirely possible to miss entire sections of one thread while replying to another.
- Filled with bad spelling, worse grammar, and vulgar language.
- Stocked with stereotypical personality types, such as the boaster, sore loser, or "are you a chick?" lecher.

Our hypothesis is that the nature of online conversations is unstructured enough to let us exploit classic text-processing techniques to fake participation in them. In particular, we can easily build a fast system that

- Recognizes good and bad events in the game (winning, losing, killing, getting killed, and so on) and reacts with appropriate responses from a large repertoire.
- Roughly recognizes when other players are trash talking and responds appropriately, depending on whether the player is friend, foe, victor, and so on.
- Ignores anything too complex to process, such as unrelated bits of conversation about real-life events.

Similar approaches have proven quite useful in the creation of chatting bots for *multiuser dungeons*, such as Julia[7] and Cobot.[8] MUD domains tend to allow for the application of simpler techniques because the game is usually highly playful and stylized, which lets the agent get away with surprisingly unrealistic behavior.[9] In contrast, first-person shooters have a much faster pace, and the communication is often more chaotic than in text-based games, further helping the bot disguise itself. Human players will not likely be overly suspicious if the bot misses some remarks directed toward it or makes nonsensical comments from time to time. Furthermore, the presence of obvious stereotypes provides a clear guide for development.

Implementation

As our development and test platform, we selected the first-person shooter Counter-Strike, a popular game that has an average of 20,000 online players and over 4,000 active game servers at any given time. The game consists of short scenarios of four minutes or less. When a player is eliminated, he or she remains in the game as a spectator until the next scenario begins. Eliminated players often observe the ongoing action and chat with one another extensively, thus the game mechanics are tailor-made for our experiment. Many juveniles play the game, several of whom fit the stereotypes mentioned in the previous section; the game itself builds on periods of inactivity where the players are implicitly encouraged to communicate with one another through a chat interface.

Although Counter-Strike was originally developed as a purely multiplayer online game, members of the Counter-Strike community have developed many bots that are readily available for download, often with source code. We chose one of these bots, Teambot (see www.planethalflife.com/teambot), as the foundation for our experiment.

The original Teambot had some chatting capability, but only as a small preset number of canned responses emitted sporadically during the game. We decided to remove all the original chatting code in Teambot and implement our own version.

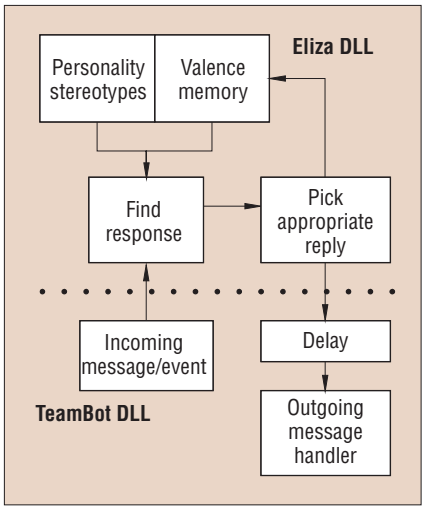


Figure 6. TeamBot chat interface.

Architecture. Figure 6 shows an outline of our architecture design. The system that we constructed is essentially an augmented version of Eliza, the classic text-processing system. It is a simple state-machine-based text-pattern-matcher, written in C for extra speed.

Incoming chat messages or significant game events are sent to Eliza, which is equipped with a set of stereotypical personalities, such as boaster, whiner, and warez dude (see Figure 7 for two examples). Each of these profiles includes triggers for matching events and conversational bits that would be of interest to that particular personality. For example, warez dude would be interested in talking about pirated software, boaster would have an extended repertoire of put-downs and boasts, and so on. The bot uses a particular personality for the game's duration. Within each personality is a different priority stack of speech behaviors. The incoming messages or events are text-based and matched against regular expressions within each behavior. The first matching behavior generates a response. At the bottom of each priority stack is a **confusion script** behavior, which generates generic stock responses to inputs that seem important but cannot be discerned by any other behavior.

Not every input message and event is passed to the Eliza module: the bot determines probabilistically whether it will respond to a given message or event. When it actively participates in the game, the bot has a low probability of responding to any messages. After it is eliminated and is merely observing, the bot becomes far more responsive.

Furthermore, the bot maintains a simple valence memory of how it used to respond to each player currently in the game. Valence is not a simulation of emotion—rather, it presents a semblance of context in the bot's conversations. A high positive value indicates that the bot used to react positively to the player, and a negative value suggests a history of negative reactions. Events and communication in the game generate a range of possible responses, and valence memory influences the bot's choice of an appropriate response. Each response also includes a valence modifier. Once the bot picks a response, the modifier further adjusts the valence value for the appropriate player.

The output response chosen is delayed for a small amount of time before being passed to the outgoing message handler. This prevents the bot from producing output faster than humanly possible.

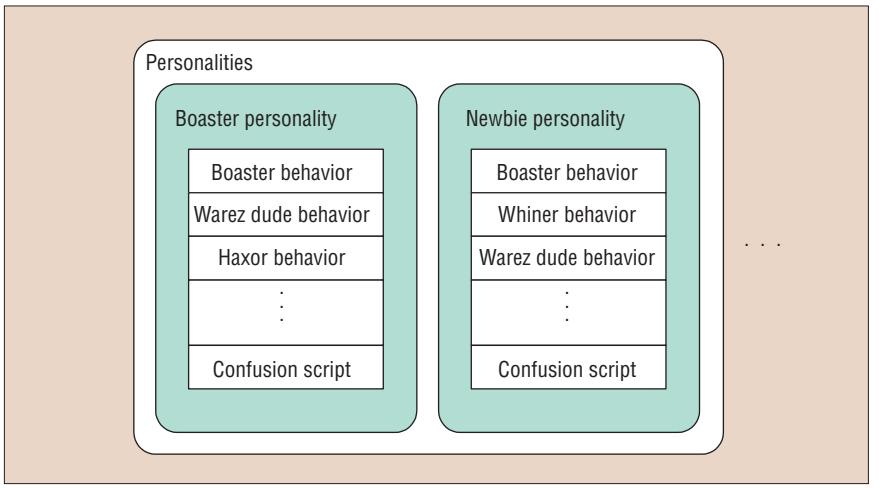


Figure 7. Behaviors with stereotypical personalities.

Results. We started a game server online for 16 players and staffed eight of those slots with our tt14m bots. From a tactical standpoint, the bots were somewhat competent on the maps, which had prepared waypoints to aid with navigation although they still performed some strange maneuvers occasionally. We logged all conversations that occurred in the game, including any between bots and human players (see Figure 8).

Our main objective was to investigate how far we could push a simple mechanism, such as an augmented Eliza, to fool a human into not treating it as a bot. We found that, in general, we could fool some of the humans for a good amount of time, but there were some people who caught on surprisingly quickly.

The primary fault we found with the current system is that Eliza-based conversations are too schizophrenic to be believable over a long period of time. The bots would jump from topic to topic without any history other than the general valence toward the speaker. Over an extended period of time, this property becomes quite noticeable. The system needs a better way of representing the current topic and should maintain a minimal topic history. A mechanism such as a decaying episodic memory model associated with each player in the game would help. The system would also benefit from some mechanism for finding and mimicking patterns in verbal behavior.

Another important problem was stylistic. Rather than referring to a person by his or her full name, players generally use some abbreviation—for example, Robzilla the Horrible would be referred to simply as Robzilla or even Rob. The bots had no understanding of this, so they routinely missed messages directed specifically at them. Conversely, when referring to a player, the bots would use the player's full name. This appears very strange, particularly when online players have a habit of using odd symbols as part of their names or have long monikers—for example, @Zbk@$hooter! or Bill Nye the Violence Guy. This reduced the bots' realism and helped players break through the illusion.

Finally, there was a serious but purely technical glitch. All players within a multiplayer game server show a specific roundtrip ping time to the server. Because they sit directly on the server, bots show up as having a ping time of only 5 ms. This is far too low for a real human player and quickly reveals that it's a bot and not a human.

An obvious next step for our work is to combine tt14m and Groo into a single system. We did not do this at present because we started the Groo project later than the tt14m project, and we felt that we should have our own infrastructure rather than rework someone else's.

On the social engagement side, we have begun investigating more complex social interaction using computationally simple mechanisms. We hope to engage in more elaborate interactions than those that Eliza affords.

On the tactical side, we have started to investigate scenarios that involve more teamwork. Deathmatch mode in first-person shooters offers a fairly limited range for team cooperation. A dynamic flocking behavior appears to be an overwhelmingly successful tactic for this mode of gameplay: it creates numerical superiority for the bot team and is simple and efficient to implement.[10] Flocking is quite easy to implement for Groo; in fact, an undergraduate student did so in his final project for a class at Northwestern. Advancing from this mode of play, we have started work on a cooperative scenario based on a capture-the-flag game built on top of FlexBot. Our goal is to exploit a coordination architecture that we developed for physical robots called HIVEMind (Highly Interconnected Very Efficient Mind) to create teams that still meet our efficiency constraints.[11] ■

Acknowledgments

We thank Greg Dunham and Nick Trienens; Groo would not exist without their invaluable work on the FlexBot infrastructure. We are also grateful to Ian Horswill for his advice and guidance.

References

1. J.E. Laird, *It Knows What You're Going to Do: Adding Anticipation to a QuakeBot*, AAAI tech. report SS-00-02, AAAI Press, Menlo Park, Calif., 2000.
2. R. Arkin, *Behavior-Based Robotics*, MIT Press, Cambridge, Mass., 1998.
3. J.E. Laird and J.C. Duchi, *Creating Human-Like Synthetic Characters with Multiple Skill Levels: A Case Study Using the Soar Quakebot*, AAAI tech. report, SS-00-03, AAAI Press, Menlo Park, Calif., 2000.
4. A. Khoo and I. Horswill, "An Efficient Coordination Architecture for Autonomous Robot Teams," to be published in *Proc. IEEE Int'l Conf. Robotics and Automation*, IEEE CS Press, Los Alamitos, Calif., 2002.
5. N. Tinbergen, *The Study of Instinct*, Clarendon Press, Oxford, UK, 1951.
6. R. Zubek and A. Khoo, *Making the Human Care: On Building Engaging Bots*, AAAI tech. report, SS-02-01, AAAI Press, Menlo Park, Calif., 2002.
7. M. Mauldin, "Chatterbots, TinyMUDs, and the Turing Test: Entering the Loebner Prize Competition," *Proc. 12th Nat'l Conf. Artificial Intelligence* (AAAI-94), AAAI Press, Menlo Park, Calif., 1994, pp. 16–21.
8. C. Isbell et al., "Cobot in LambdaMOO: A Social Statistics Agent," *Proc. 17th Nat'l Conf. Artificial Intelligence* (AAAI-2000), AAAI Press, Menlo Park, Calif., 2000, pp. 36–41.
9. L.N. Foner, "Entertaining Agents: A Sociological Case Study," *Proc. 1st Int'l Conf. Autonomous Agents*, ACM Press, New York, 1997, pp. 122–129.
10. M.J. Mataric, *Interaction and Intelligent Behavior*, tech. report AITR-1495, Massachusetts Inst. of Technology, Cambridge, Mass., 1994.
11. A. Khoo et al., *Efficient, Realistic NPC Control Systems Using Behavior-Based Techniques*, AAAI tech. report, SS-02-01, AAAI Press, Menlo Park, Calif., 2002

***DEAD*!PRV!AKOrbValk :** whats the name of this map
***DEAD*@PRV@DeathFubar :** guard that bomb
*DEAD*Dr-Azrael : hah
***DEAD*Silent Bob :** we black
DEAD+EwokAce-KEC+ : doh
*DEAD*Dr-Azrael : de_dust (<— *responds with the name of the map*)
***DEAD*EvilSuperMalachi :** i'm sick of being capped
*DEAD*Silent Bob : *shakes head 'no'*
***DEAD*!hun!EvilSuperFoo :** how do you use the scope?
*DEAD*Dr-Azrael : right click

(a)

***DEAD*@kNP@SecretToolValk** : who are the bots here?
Accord : the people with 5 ping are bots
FunkyFunk : i hear the 5 ping thing is a server problem
BossWhax : heard anything about the new cs expansion?
Accord : damnit, i just answered a bot
PsychoPainHead : i've known damnit, i just answered forever, he aint no bot

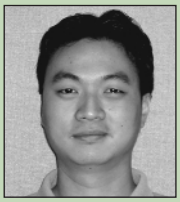
(b)

Dr-Azrael : who me?
Dr-Azrael : damn it i was reloading
CountFoo[NUCS] : real men dont camp
@MonsterFooQuee-NUCS@ : whip that f—- a—
Silent Bob : *nods*
Dr-Azrael : i wasn't camping

(c)

Figure 8. Some of the more interesting exchanges from the tt14m game logs: (a) sample conversation involving bots; (b) sometimes human players notice that they are talking with bots, (c) but sometimes they don't. Boldface denotes bot names.

The Authors



Aaron Khoo is a PhD candidate at Northwestern University, where he is a member of the Autonomous Mobile Robotics Group. His primary research interests lie in the realm of multiagent systems and human–computer interaction. He received a BA in computer science and mathematics from Knox College. He is a member of AAAI. Contact him at Northwestern Univ., 1890 Maple Ave., Evanston, IL 60201; khoo@cs.northwestern.edu; www.cs.northwestern.edu/~khoo.

Robert Zubek is a PhD candidate at Northwestern University, where he is a member of the Autonomous Mobile Robotics Group and the Interactive Entertainment Group. His research interests include artificial intelligence for computer entertainment and modeling of social interaction. He received his BS in computer science from Northwestern. Contact him at Northwestern Univ., 1890 Maple Ave., Evanston, IL 60201; rob@cs.northwestern.edu; www.cs.northwestern.edu/~rob.