Using a Computer Game to Develop Advanced AI

Building agents that can survive the harsh environment of a popular computer game provides fresh insight into the study of artificial intelligence.

John E. Laird
University of Michigan

Although computer and video games have existed for fewer than 40 years, they are already serious business. Entertainment software, the entertainment industry's fastest growing segment, currently generates sales surpassing the film industry's gross revenues. Computer games have significantly affected personal computer sales, providing the initial application for CD-ROMs, driving advancements in graphics technology, and motivating the purchase of ever faster machines. Next-generation computer game consoles are extending this trend, with Sony and Toshiba spending $2 billion to develop the Playstation 2 and Microsoft planning to spend more than $500 million just to market its Xbox console.[1]

These investments have paid off. In the past five years, the quality and complexity of computer games have advanced significantly. Computer graphics have shown the most noticeable improvement, with the number of polygons rendered in a scene increasing almost exponentially each year, significantly enhancing the games' realism. For example, the original Playstation, released in 1995, renders 300,000 polygons per second, while Sega's Dreamcast, released in 1999, renders 3 million polygons per second. The Playstation 2 sets the current standard, rendering 66 million polygons per second, while projections indicate the Xbox will render more than 100 million polygons per second. Thus, the images on today's $300 game consoles rival or surpass those available on the previous decade's $50,000 computers.

The impact of these improvements is evident in the complexity and realism of the environments underlying today's games, from detailed indoor rooms and corridors to vast outdoor landscapes. These games populate the environments with both human and computer controlled characters, making them a rich laboratory for artificial intelligence research into developing intelligent and social autonomous agents.

Indeed, computer games offer a fitting subject for serious academic study, undergraduate education, and graduate student and faculty research. Creating and efficiently rendering these environments touches on every topic in a computer science curriculum. The "Teaching Game Design" sidebar describes the benefits and challenges of developing computer game design courses, an increasingly popular field of study.

COMPUTER GAME AI

In computer games, designers can use AI to control individual characters, provide strategic direction to character groups, dynamically change parameters to make the game appropriately challenging, or produce a sports game's play-by-play commentary.[2,3] Computer games offer an inexpensive, reliable, and surprisingly accessible research environment, often with built-in AI interfaces.

Moreover, computer games avoid many of the criticisms leveled against research based on simulations: Because researchers do not develop them, the games avoid embodying preconceived notions about which aspects of the world designers can simulate with ease and which aspects they can simulate only with difficulty. These games constitute real products that create real environments with which millions of humans can interact vigorously.

Development costs and the lack of processing power prevent games for today's computers and consoles from exhibiting a high level of artificial intelligence. Currently, developers devote more resources to advancing a game's graphics technology than to enhancing its AI. Within two to three years, however, the emphasis on graphics

0018-9162/01/$10.00 © 2001 IEEE

Teaching Game Design

At their core, most computer games simulate complex worlds. Games immerse players in an alternative reality and challenge them to assume a role—be it adventurer, world-class sports figure, or ruler of a civilization. Thus, a senior-level design course on computer game development provides the perfect integration of concepts from across computer science, all within the context of a highly motivating topic. For once, a professor needn't worry about students finding the subject matter unengaging—they dive into it, drawing from all their previous education to build their games.

Ironically, something as apparently frivolous as playing computer games makes students finally appreciate the importance of the seemingly obscure material they learned in earlier courses. Nothing motivates students to understand the computational complexity of collision-detection algorithms like having their game run slower and slower as they add more enemy spaceships from the Gamma Nebula.

Organizational lessons

Studying computer games also exposes students to the world outside computer science. Today, games development requires more than two programmers working together in a garage—it requires collaboration among people from diverse disciplines. A team normally includes producers, game designers, programmers, artists, sound engineers, and specialists such as voice actors and motion-capture experts. Each team member must understand how all the pieces fit together, forcing students to appreciate the importance of art, music, literature, and social skills.

As an additional incentive, students with game programming experience can easily find interesting and challenging summer and full-time jobs in the videogame industry. Further, many game companies searching for fresh talent run summer internship programs.

A growing number of campuses offer computer game design courses, including North Texas University, the University of Michigan, Georgia Tech, Northwestern, and the University of California, Irvine. DigiPen and Full Sail offer degree programs in computer game design.

Our program

My course provides a rigorous introduction to the technology, science, and art behind computer game development. The course assumes that students already have a solid background in computer science, including programming and algorithms. In addition to four programming projects designed to expose students to the breadth of topics in computer game development, the course also provides the opportunity to build complete games. Students must integrate both the technical and aesthetic aspects of computer games. They must design, implement, then iteratively refine the game to enhance its entertainment value. The course's four projects, and the goals for each, follow:

- *Arcade game.* We provide the students with a shell package for Pong, a game that runs under Windows on a PC that uses Direct-X. The students must devise, implement, and test their own game. Students can reimplement classic arcade or console games, but we encourage them to develop novel games.
- *Interactive fiction.* The students use the Inform interactive fiction authoring system to develop their own text-based interactive fiction. Although text-based interactive fiction went out of fashion more than ten years ago, this project forces students to focus on the role of plot and story in a game without having to worry about graphics.
- *Artificial intelligence.* Students use the Soar architecture to build a bot that controls a tank in a simulated maze world. We then hold an elimination tournament in which the student bots fight for supremacy. The project exposes students to AI programming and to one approach to encoding complex tactics.
- *Final project.* During the last five weeks of the course, the students work in teams to develop the game of their choice. For their final project, students must complete all the standard stages of game development: creating a concept document, pitching the concept to the class, developing a design document, and drafting the technical specification. The course culminates in an open house during which more than 150 people come to play and rate the best games developed in class.

These projects build on the coursework and require the students to research both the design and mechanics of developing their game.

will likely have run its course as incremental improvements in the underlying technology lead to only marginal improvements in the game experience.

Game designers continually look for new ways to distinguish their games, and companies already market games based on their AI. Recently, AI has been featured in two very successful games, The Sims and Black and White. In the near future, more development and runtime resources will be available to increase game AI complexity and realism, at which point we can expect to see a significant growth in the role of AI in games.

DEVELOPING QUAKE II AI

My research group uses computer games as its environment for research on building human-level AI. We do basic research on AI within computer games and also demonstrate AI's potential for future computer games. Our most mature research involves building AI bots that play the infamously violent action computer game Quake II. We selected the game as an AI development platform because its developer, id software, publishes an interface that makes it possible to control Quake II's bots via external software. Following id's lead, other games, such as Unreal Tournament and Half-Life, provide interfaces so that anyone can write code to control characters in the game through a dynamically linked library (DLL). Other research groups are starting to pursue similar research projects using Unreal Tournament, although with an emphasis on team play.[4]

The bots included in the game provide challenging opponents, but only because of their superhuman reaction times and aiming skills, not because of their

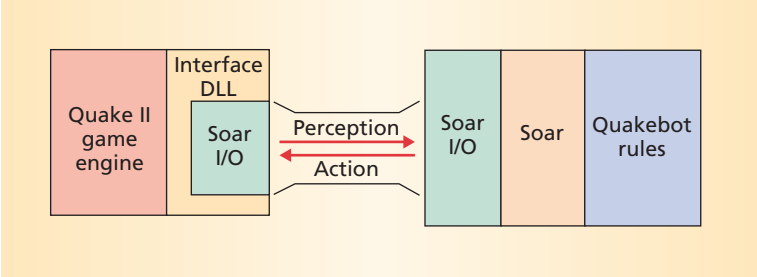

Figure 1. A graphic depiction of the interface between Quake II and the Soar Quakebot, with Quake II on the left, the interface dynamic linked library (DLL) in the middle, and Soar and its associated Quakebot rules on the right.

complex tactics or game knowledge. In contrast, we seek to create bots that use many of the tactics humans use so that they beat you by outthinking you, not by outshooting you.

Play mechanics

When multiple human and bot players engage in a Quake II match at the same time, they exist in a level that contains hallways and rooms. To explain the game using its own terminology, the players move through the level, acquiring *power-ups* such as weapons, ammo, armor, and first-aid boxes. The players also fire their weapons in an attempt to injure their enemies. When a player's health reaches zero, the player dies, then instantly reappears at one of the level's preset *spawning sites*. The first player to rack up a prespecified number of kills wins the game.

The game distributes power-ups in static locations throughout the level. When a player takes a power-up, a replacement automatically regenerates in 30 seconds. Weapons vary according to their range, accuracy, damage spread, reload time, ammo type, and amount of damage they cause. For example, the shotgun spreads its damage across a wide area at short range, but does no damage if the player fires it at distant targets. In contrast, the railgun kills in a single shot at any distance, but requires very precise aim because it has zero spread.

Although it might appear from this description that Quake II is a mindless point-and-shoot game, the factors I've listed—combined with the layout of the level's topology—make the game tactically complex, which becomes obvious when watching an expert play.

The Quakebot

Our Quakebot uses Soar[5]—an engine for making and executing decisions—as its underlying AI engine for controlling a single player. We chose Soar as our AI engine because our real research goal is to understand and develop general integrated intelligent agents. Soar embodies our current best hypotheses for the cognitive architecture underlying general intelligent agents, with research and refinement of it stretching back twenty years.

Figure 1 shows the overall structure of the system. Quake II is the commercial version of the game, which we have extended with the DLL. The DLL, written in C, implements the Quakebot's sensors and motor actions, along with the Soar I/O, our intercomputer communication code. Soar I/O provides a platform-independent mechanism for transmitting all perception and motor information between the Quakebot and the game. We encoded all knowledge for playing the game into the Soar rules.

The underlying Quake II game engine updates the world and calls the interface ten times a second. On each of these cycles, it updates all changes to the bots and initiates any requested motor actions. Soar runs asynchronously to Quake II and executes its basic decision cycle anywhere from 30 to 50 times a second, allowing it to take multiple reasoning steps for each change in its sensors. Soar consumes 5 to 10 percent of the processing power of a 400-MHz Pentium II that runs Windows NT.

The Soar Quakebot's design uses principles originally developed for controlling robots, then extended in our research to simulate military pilots in large-scale distributed flight simulations.[6,7] In Soar, an operator—the basic object of decision—consists of

- primitive actions that bots perform in the world such as move, turn, or shoot;
- internal actions such as remembering the enemy's last position; or
- more abstract goals such as attack, get-item, or goto-next-room that, in turn, must dynamically decompose into simpler operators that ultimately bottom out in primitive actions.

Soar continually proposes, selects, and applies operators to the current state via if-then rules that match against the data structures that make up the current state. When Soar selects an abstract operator that cannot be applied immediately, such as collect-power-ups, it generates a substate. Within a substate, Soar then proposes, selects, and applies additional operators until the original operator completes, or until the game environment changes such that it triggers the proposal and selection of another operator. These additional operators may require further decomposition, leading to the recursive dynamic generation of an operator hierarchy.

Figure 2 shows a subset of the Quakebot's operator hierarchy. Although a small part of the overall hierarchy, this list includes some of the top-level operators, such as wander, explore, attack, and other operators that Soar uses to apply the collect-power-ups operator.

During a game, rules will propose top-level operators—such as wander, explore, and attack—by testing information gleaned from the Quakebots sensors. For example, if the Quakebot does not sense an enemy or have a recent memory of one, and an inventory check reveals that it's missing some important power-ups, a rule will propose the collect-power-ups operator.

Once Soar selects collect-power-ups, a substate will generate in which rules propose the get-item operator for each missing power-up. Additional rules will then fire to create preferences so that the bot pursues the

most important power-up first. After Soar selects get-item, another substate generates, and rules test whether the item being pursued is in the current room or another one. If the power-up is in the current room, the rules further decompose the problem until primitive operators, such as face-the-item and move-to-the-item, are proposed, selected, and applied.

Figure 3 shows the main tactics the Quakebot uses. It implements these actions via the top-level operators and their decompositions. Although Quakebots react to different situations and opponents, they originally did not anticipate other players' behavior.

Anticipation's role

The following quote from Dennis "Thresh" Fong,[8] Quake's Michael Jordan equivalent, gives some insight into the importance of anticipation:

> Say my opponent walks into a room. I'm visualizing him walking in, picking up the weapon. On his way out, I'm waiting at the doorway and I fire a rocket two seconds before he even rounds the corner. A lot of people rely strictly on aim, but everybody has their bad aim days. So even if I'm having a bad day, I can still pull out a win. That's why I've never lost a tournament.

We can program these tactics manually for specific locations within a specific level. For example, we could add tests that direct a bot to set an ambush if it occupies a specific location on a specific level and hears a specific noise, such as the sound of an enemy picking up a weapon. Computer games currently use this approach, which requires a tremendous effort to create a large number of tactics that work for each specific level.

An alternative to encoding behaviors for each of these specific situations is to add a general capability for anticipating an opponent's actions. Anticipation is a form of planning that AI researchers have studied for 40 years. The power of chess and checkers programs comes directly from their ability to anticipate opponent's responses to their own moves. Anticipation for bots in first-person shooters requires a few twists that differentiate AI for the FPS genre from standard AI techniques such as alpha-beta search:

- Unlike a chess or checkers player, an FPS player lacks access to the complete game state.
- The choices for a player's actions in an FPS game unfold continuously as time passes. At any time, the player can move, turn, shoot, jump, or remain in one place.

The breadth and depth of possible actions in an FPS game quickly make search intractable and require more knowledge about which actions might prove useful.

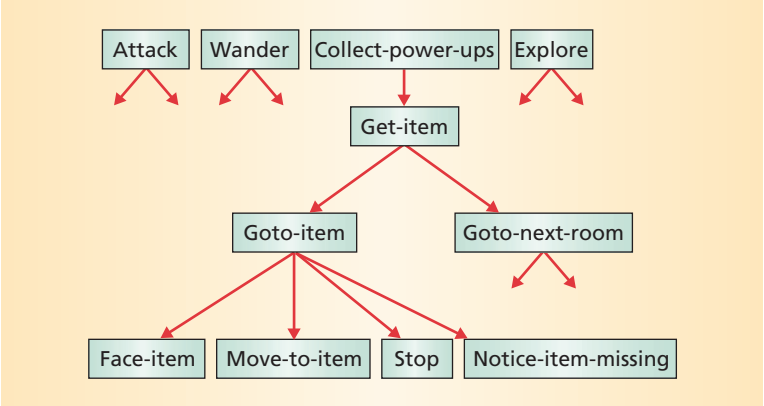

Figure 2. A subset of the Quakebot's operator hierarchy, which determines how the bot reacts to its environment and changing game conditions.

Internal representation. To support anticipation, the first thing the Quakebot must have is the ability to create its own internal representation of what it believes to be the enemy's internal state—what the enemy is currently sensing. It builds this representation based on its observations of the enemy. The Quakebot then predicts the enemy's behavior by using its own knowledge of tactics to select what it would do if it were that enemy. Using simple rules to internally simulate external actions in the environment, the bot forward projects until it either gets a

Collect-power-ups
Pick up items based on their spawn locations
Pick up weapons based on their quality
Abandon collecting items that are missing
Remember when missing items will respawn
Use shortest paths to get objects
Get first-aid kits and armor if needed
Pick up other quality weapons and ammo if accessible

Attack
Use circle-strafe (walk sidewise while shooting)
Move to best distance for current weapon

Retreat
Run away if low on health or outmatched by the enemy's weapon

Chase
Pursue the enemy based on sound of running
Proceed where enemy was last seen

Ambush
Wait in a corner of a room that can't be seen by enemy coming into the room

Hunt
Go to nearest spawn room after killing enemy
Go to rooms enemy is often seen in

Figure 3. A breakdown of Quakebot's tactics, listing the accompanying actions for each operator.

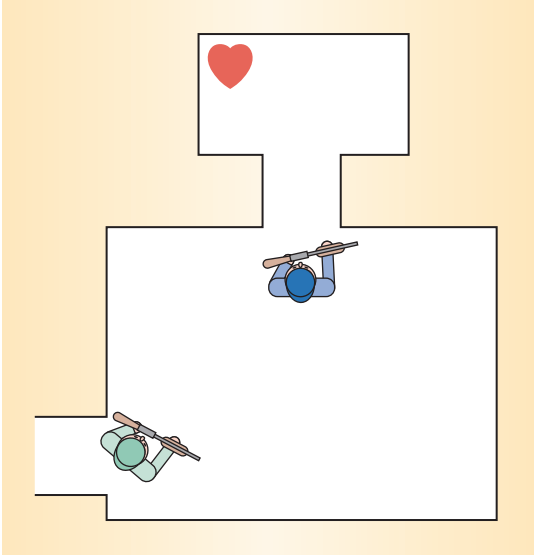

Figure 4. The Quakebot, lower left, decides to predict the actions of its enemy, which stands at the threshold to the small room at the top. The bot predicts that the enemy will continue into the room to obtain the heart, a valuable game item. If the enemy turns on the bot, however, the prediction process aborts so that the bot can respond to the attack immediately.

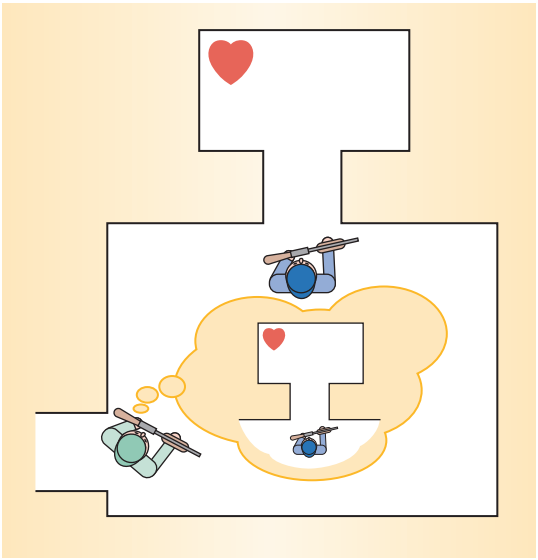

Figure 5. The Quakebot, lower left, uses its knowledge of what it would do in the enemy's state to create an internal representation of the enemy's situation.

useful prediction or decides too much uncertainty exists about what the enemy will do next. The bot uses such predictions to, for example, set an ambush or deny the enemy an important weapon or health item.

Figure 4 shows an example in which the Quakebot, pictured in the lower left, sees its enemy, pictured in the upper center, heading toward a desirable object—the heart. As the enemy quickly moves out of range, the Quakebot decides to predict the enemy's future course of action. If the enemy turns toward the Quakebot instead of continuing toward the heart, the prediction process aborts so that the Quakebot can immediately attack and not be caught planning.

Once it makes the decision to predict the enemy's behavior, the Quakebot creates an internal representation of the enemy's state based on its perception of the enemy. Figure 5 shows the Quakebot creating an internal representation of the enemy's situation that it will use for its internal planning.

Quakebot prediction. In the next step, the Quakebot uses its representation of the enemy's state and its knowledge of what it would do in the enemy's state to predict the enemy's actions. Thus, we assume that the enemy's goals and tactics are essentially identical to the Quakebot's. In this example, the bot knows, from prior explorations, that the power-up resides in the small room at the top, and it now attributes that knowledge to the enemy. The bot projects that its adversary will collect the power-up, which leads to a cascade of problem solving as it decides to execute get-item and goto-next-room operations.

Once the Quakebot gets to this point, it internally simulates the enemy's movement from room to room. It also can simulate the changes that will happen as other operators apply, such as its health increasing if it picks up a first-aid kit. The selection and application of operators continues until the Quakebot determines that the enemy will take the power-up, as Figure 6a shows. At that point, the Quakebot predicts the enemy will change top-level operators and choose the wander operator. Figure 6b shows that, because the room has only one exit, wander will direct the enemy to exit, move back down the hallway, and finally reenter the room from which it started—and which the Quakebot now occupies.

During the internal simulation, the bot continually compares the distance in terms of the number of rooms the enemy has traveled to the distance the Quakebot requires to get to the same location. In the preceding example, the Quakebot predicts that the enemy will require four moves to get back to the room that the Quakebot currently occupies. Why doesn't the Quakebot stop predicting at the point that the enemy would be in the hallway—only three moves for the enemy versus one for the bot? Because the Quakebot knows an ambush cannot be set in a hallway, where there is no place to hide. Thus, it waits until the enemy's predicted location is a room, where it knows it can set an ambush, as shown in Figure 7.

When the Quakebot predicts that the enemy will be in another room that the bot can reach sooner, the Quakebot attempts to set an ambush by moving to an open location next to the door that its code predicts the enemy will move through. In general, the bot tries to shoot the enemy in the back or side as it enters the room. If the bot carries a rocket launcher, it will take a preemptive shot when it hears the enemy getting close—just as star player Dennis Fong would. We associate time limits with both of these ambush strategies, however, so that the bot will not wait indefinitely at the ambush site.

Our work with the Quakebot demonstrates that we can successfully pursue serious research on autonomous AI agents within the context of computer games. Our research directly applies to computer-generated forces,[6,7] where we need to

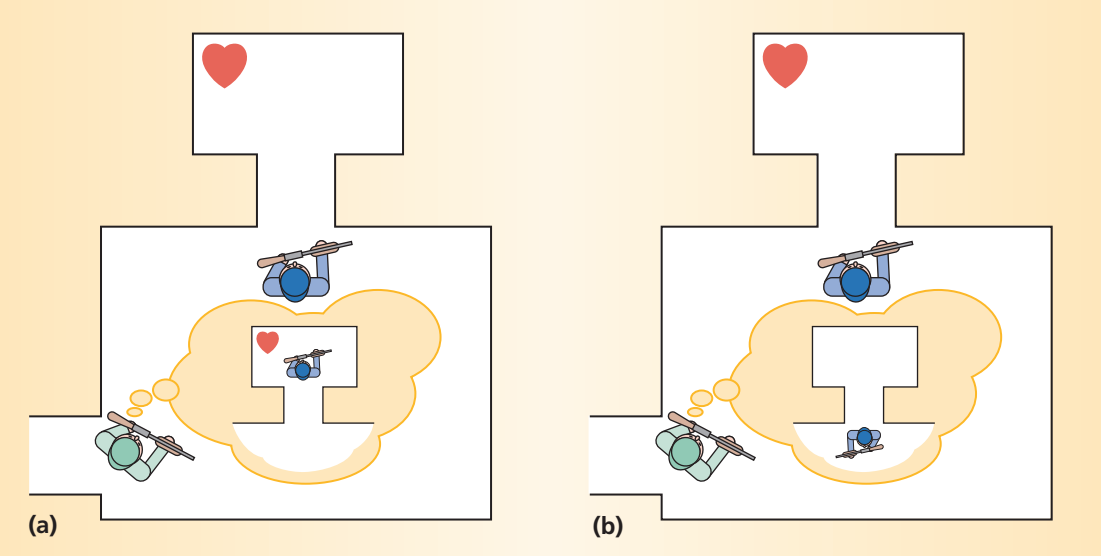

Figure 6. The Quakebot projects that its enemy will (a) move into the upper room in pursuit of the heart power-up, then (b) return to the room the bot occupies. The bot uses this knowledge to plan and execute its countermoves.

model realistic, entity-level behavior. We find computer game environments useful in doing small studies on the effect of different cognitive parameters on our Quakebot's skill levels. Using Quake II, we have successfully studied the impact on overall game performance from changes in reaction time, tactics level, and perceptual and motor skills.

From its scoring method, which rewards the highest number of kills, it's obvious that Quake II epitomizes violent computer games. Although we have found computer games to be a rich environment for research on human-level behavior representation, we do not believe that the future of AI in games lies in creating more and more realistic arenas for violence.

Better AI in games has the potential for creating new game types in which social interactions, not violence, dominate. The Sims[9] provides an excellent example of how social interactions can be the basis for an engaging game. Thus, we are pursuing further research within the context of creating computer games that emphasizes the drama that arises from social interactions between humans and computer characters.

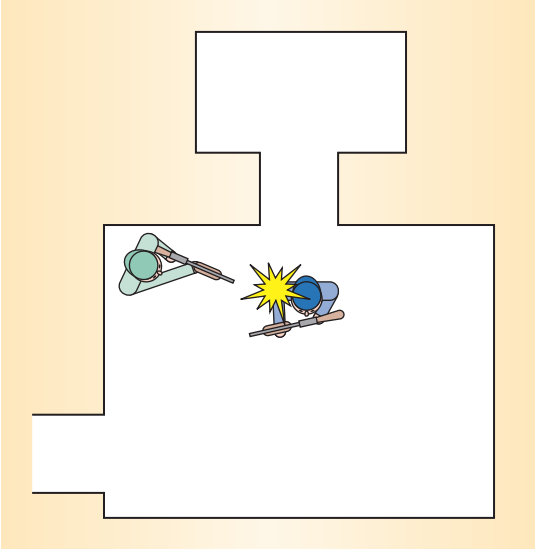

Figure 7. Since only a room provides the cover necessary to set an ambush, the Soar bot does not attack the enemy when it comes back down the hallway. Instead, the bot waits until the enemy reenters the room and surprises it from behind.

References

1. International Digital Software Association, http://www.idsa.com (current 24 May 2001).
2. J.E. Laird and M. van Lent, "Interactive Computer Games: Human-Level AI's Killer Application," *Proc. Nat'l Conf. A.I.*, AAAI Press, Menlo Park, Calif., 2000, pp. 1171-1178.
3. M. van Lent and J. E. Laird, eds., "Artificial Intelligence and Interactive Entertainment," Tech. Report SS-01-02, AAAI Press, Mar. 2001.
4. R. Adobbati et al., "GameBots: A 3D Virtual World Test-Bed for Multi-Agent Research," *Proc. 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, ACM Press, New York, 2001, pp. 47-52.
5. J.E. Laird, A. Newell, and P.S. Rosenbloom, "Soar: An Architecture for General Intelligence," *Artificial Intelligence*, Mar. 1987, pp. 1-64.
6. R.M. Jones et al., "Automated Intelligent Pilots for Combat Flight Simulation," *AI Magazine*, Jan. 1999, pp. 27-42.
7. J.E. Laird, "An Exploration into Computer Games and Computer Generated Forces," *Proc. 9th Conf. Computer Generated Forces and Behavioral Representation*, 2000, http://www.sisostds.org/cgf-br/9th/view-papers.htm (current 6 June 2001).
8. N. Croal, "Making a Killing at Quake," *Newsweek*, 22 Nov. 1999, pg. 104.
9. M. Macedonia, "Using Technology and Innovation to Simulate Daily Life," *Computer*, Apr. 2000, pp. 110-112.

***John E. Laird** is a professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, where he is the associate chair of Computer Science and Engineering. He received a PhD from Carnegie Mellon University. His research interests focus on building autonomous intelligent agents and the architecture underlying intelligence. Contact him at laird@umich.edu.*