

An Exploration into Computer Games and Computer Generated Forces

John E. Laird
University of Michigan
1101 Beal Ave.
Ann Arbor, Michigan 48109-2110
laird@umich.edu

Keywords:

Computer games, computer generated forces, artificial intelligence, anticipation

ABSTRACT: *The artificial intelligence (AI) components of computer games often appear to be very complex, possibly having abilities beyond the state of the art in computer generated forces. In this paper we study the similarities and differences between AIs for computer games and computer generated forces (CGFs). We contrast the goals of AIs and CGFs, their behavioral requirements, and the underlying resources available for developing and fielding them, with an eye to how they impact the complexity of their behaviors. Our conclusion is that CGFs are currently far ahead of game AIs, but that this may change soon. We argue that computer games have advantages for doing certain types of research on complex, human-level behavior. We support this argument with a demonstration of research we have done on AI and computer games. We have developed the Soar Quakebot, which is a Soar program that plays the death match version of Quake II. The design of the Soar Quakebot is based on TacAir-Soar, a real-time expert system that flies U.S. military air missions in simulation, and that is used for training in the U.S. Air Force. The Soar Quakebot incorporates complex tactics and the ability of the bot to anticipate the actions of its enemy.*

1. Introduction

Over the last five years, there have been amazing advances in the quality and complexity of computer games. The most noticeable advances have come in computer graphics, where the number of polygons that are rendered in a scene seems to increase exponentially each year. Images on today's \$400 game consoles rival or surpass those on \$50,000 computers from only a few years ago. Secondary to the graphics has been the complexity of the underlying environments, be they indoor rooms and corridors or outdoor landscapes. These and other features have led to a significant improvement of the realism of games. Embedded within these simulated worlds are AIs - computer controlled synthetic characters that populate the worlds, sometimes as members of the same team, but usually as enemy fodder for the player to kill. When taken together, these advances make the experience of playing games very compelling. Although it is tempting to dismiss them as frivolous, their behavior is often very captivating and shows enough intelligence to raise an uncomfortable thought in the minds of CGFs developers, "Have computer games made CGFs obsolete?"

The short answer is, "No." The best work in CGFs, as demonstrated in STOW-97, is still years ahead of the synthetic characters developed in computer games in terms of complexity and realism of behavior. The first part of this paper explores the reasons for this, which are rooted in the significant differences between the goals, available resources, and environments of the game industry vs. the DOD simulation world. These differences

lead to significant differences in the complexity and scalability of the behavioral representation approaches used in the two fields.

Even if computer games have a long way to go in terms of realistic modeling of complex human behavior, they provide environments that are worth considering for research on building human-level AI characters [5]. The immersive three-dimensional worlds of computer games provide complex, exciting, stable, and cheap environments for research and development into the advanced capabilities required by computer generated forces. This point is illustrated in the second part of this paper by an example of our own research where we have transitioned our technology for developing CGFs to developing synthetic characters in computer games. Computer games have simplified some of the issues that arise in research on CGFs while allowing us to create "authentic", realistic, and complex behavior. Our major advancement has been in extending our synthetic characters so that they anticipate the actions of their opponents.

2. Computer Generated Forces

Computer generated forces can be used in training, mission rehearsal, and weapons development and testing, as well as many other applications such as doctrine and tactics development. In these applications, the CGFs populate the synthetic battle space with entities that substitute for humans. They are used most often for training and mission rehearsal where the cost of

populating the battle space with humans and either real or simulated vehicles is prohibitively expensive for realistically sized training exercises. For example, when training or rehearsing a division of four Navy pilots, over forty additional entities might be needed to provide friendly support, enemy planes, and ground forces. Moreover, if the goal is to train a group of mid- to high-level commanders, hundreds, thousands, or tens of thousands of units might be needed.

When CGFs are used for training (and other purposes), the primary goal is to replicate the behavior of a human; that is the behavior should be *realistic* [8]. Without realistic, human-like behavior, the danger is that human trainees interacting with the CGFs will have negative training. Therefore, a CGF should obey appropriate doctrine and tactics, and have the same strengths and weaknesses as humans both in physical and mental abilities. In the limit, this includes human behavioral characteristics such as response to battlefield stress (e.g. fatigue), emotions (e.g. frustration, anger, fear), and other psycho-physiological human characteristics. For some applications of CGFs, the realism is less important and the CGFs can be less complex.

The U.S. DOD has significant resources available in order to develop realistic CGFs. For many years, DARPA funded research groups under the CFOR, STOW and the ASTT programs. The individual services have also invested 10's if not 100's of millions of dollars in CGFs. Development has covered semi-automated entity-level forces that combine simple autonomous behavior with human control, such as available under ModSAF or CCTT; autonomous intelligent entity-level forces that can work independently or as teams, obeying standard doctrine and tactics, such as our work on TacAir-Soar; command forces that plan activities for entity-level units, such as the systems developed under CFOR. Each of these efforts have involved at least 10's of man-years of development, usually spread over multiple years and they include significant knowledge-acquisition efforts combined with verification and validation of the behaviors.

The resources available in the fielding of CGFs span a broad spectrum depending on the required complexity and realism of behaviors of the CGFs. For the simplest CGFs, hundreds or thousands might run on a single processor. For those incorporating detailed realistic behavior at the entity or command level, there may be only one to ten per processor.

The environment that the CGFs exist in can be very large and complex. For example, the STOW environment covered a 500 x 775 km area that was populated by over 3,700 active entities and over 10,000 buildings. Although unusually large, other simulation environments have the

same unrestricted aspect in that the CGF developer does not know beforehand where and how his units will be used. During operation, the CGF's behavior must be unscripted, responding to the situations as they develop. However, they have to modulate and customize their behaviors based on the specific missions being performed. For STOW, our planes had to accept the standard Air Tasking Order that is used to specify air missions throughout the U.S. military. During the performance of those missions, the CGFs had to communicate with other CGFs and humans (this was a critical component of the CFOR project), sometimes responding to dynamic changes in their orders.

To summarize, the goal for the most complex CGFs is to display realistic behavior in the complex, uncontrolled simulation environments. The DOD has made available significant resources available to create these systems using various AI approaches such as hierarchical finite-state machines, rule-based systems, hierarchical goal-structured rule-based systems, constraint-satisfaction systems, and hierarchical planning systems. Although these approaches are all different, they share the idea of creating a separate level of description for encoding behavior about strategy, tactics, doctrine, and behavior. They rise above standard programming languages such as C and C++.

3. Game AIs

In computer games, AI can be used to control individual characters in the game, to provide strategic direction to groups of characters, to dynamically change parameters in a game to make it appropriately challenging, or to even produce the play by play commentary in a sports game [1,2,9]. For this paper I will concentrate on the types of AI that are used to control individual characters or provide strategic direction to groups of characters. These uses map most directly on to the uses of AI in military simulation. Just as in military simulation, the role of AIs in computer games is to populate an environment with entities that the human plays with and against.

In contrast to military simulation, the goal of computer games is to entertain the person playing the game. The AI must generate behavior that makes the game fun to play. This is even more vague than "human-like behavior", and can only be defined within the context of a specific game. Moreover, when there is a desire for human-like behavior, the desire is really only for the *illusion* of human-like behavior, and the effort is put into the illusion, not into the accuracy or competence of the underlying behaviors.

Often, the AI must just be competent, performing the standard actions of the game, such as blocking, running, passing, and tackling in a football game. But it is also important at the higher levels of a sports game that the

play calling that is not too predictable. In many action games, the role of the AI is not to be a human-level opponent, but to be a "punching bag" for the human player to beat up on. If the AI does more than run straight at you, it is considered very sophisticated. In strategy games, it is not critical that the units being controlled behave exactly like humans, only that they don't do obviously dumb things (like run into walls or shoot their own forces) and that they carry out the orders given them to by a human player or a computer controlled enemy. In games where there is a plot or storyline, such as adventure games and many action games, the game AI's behavior is usually tightly scripted so that they "pop-out" at specific times, or engage in fixed dialogue to forward the plot. In games where realism seems important, what is critical is that the behavior looks "good", not that it is verified to be good.

Another aspect of being entertaining is that the AI provides the human with a satisfying game experience, so that it is much more important that the AI opponent is neither too easy nor too hard than that it plays extremely realistic. Often the role of the AI is not to provide the kind of experience you would get in training, where you might get your butt kicked whenever you make a mistake. Instead, the AI just needs to put up a good fight, and then lose convincingly - so that the human player feels a sense of achievement.

The standard development cycle for a computer game is one year. Some ambitious games can take up to 3 years, but the majority of games are developed within a year or less. On a development team, there will usually be one person responsible for the AI. For action, adventure, and military simulation games, that person will create the basic behaviors and a scripting language. The scripting language is used by level designers to create specific AIs for the different parts of the game. The limits on available manpower strictly limit the complexity of behaviors for the AI. For sports games, usually the AI used in the previous years is tweaked, and rarely rewritten from scratch. And although more and more people in the game industry are being trained in AI methods, the development schedules prevent them from using the complexity of techniques employed in CGFs.

The fielding of computer game AI is pretty straightforward. The game AI is usually allocated only 5-10% of the CPU of the computer the game is played on. The remaining CPU is dedicated to graphics, networking, sound, game play, physics etc. Even for a sports game, such as soccer, where there are many players, the game AI is getting very little of the CPU.

How do computer games give any semblance of realistic behavior? First, the environments in which they behave are very restricted and are preprocessed to simplify their

runtime processing. All of those tough problems that CGFs must solve by themselves (realistic sensor processing, terrain and spatial reasoning, coordination, and complex strategic planning) are finessed in computer games. For example, in strategic games, a human has already analyzed the terrain and created an annotated map for the game AI with all of the important tactical and strategic locations. Moreover, the human expert has probably created a set of scripts for the AI to follow and one is picked randomly when the game starts. In driving games and action games, specific paths through the environment have been laid down for the AI to follow. Furthermore, the computer AI feels no compunction to play fair. To simplify its reasoning, the game designer will often give the game AI access to the complete game state - it knows where your forces are even though it cannot realistically "sense" them. Remember, what is important is that the game is fun to play, so a valid approach is for the game AI to make up for its weaknesses by cheating, which works well as long as it doesn't get caught. The result of all of these simplifications and cheats is that the AI in computer games does not scale up to real world problems and must be hand customized to new scenarios and new missions.

Because of the lack of resources for development, game developers often code the AIs directly in C. Conditional actions that would normally be encoded as rule-based system end up being lots of embedded if-statements in C. Moreover, only rarely is there a clean separation between the AI logic and the rest of the game logic. The one exception is that in many adventure and action games, scripting languages are developed so that level designers, not programmers, can specify the behavior of game AIs to further the plot or story. These languages provide a way for non-programmers to specify bits of conditional and sequential behavior as well as some coordination. However, the scripts are always very specific to one small part of the game. Some progress is being made, with finite-state machines being used more and more for control of simple units and characters. In addition, complex path finding algorithms are employed for unit-level AIs being built on top of A*.

Although game AI has a long way to go to compare to the state of the art in CGFs, I expect that it will make up ground fast. Up to now, graphics has been the driving force behind advances in computer games; however, within 2-3 years advances in graphics may run their course as the improvements in the underlying technology leads to only marginal improvements in the game experience. The advent of the Playstation 2 and the Microsoft X-box will support such a high-level of graphics that more of the CPU can be dedicated to AI. Thus, we can expect to see more development and runtime resources available for game AI, and making it possible to increase complexity and realism in game AI. As games

become less linear and more complex, it will become cheaper to develop more realistic and general AIs than hardcode in specific behaviors. Moreover, game designer will look for new areas to distinguish their games, with AI being an obvious choice. Already games are marketed based on their AI, as weak as it is.

4. Computer Games in CGF Research

Although the goals of computer games are not always aligned with the DOD simulation community, the game industry has created some amazing environments in which CGF related research could be performed. Action and adventure games are set in complex three-dimensional worlds, often with military themes. Although computer game designers have chosen to finesse many of the hard research issues, these environments can be used to study those issues. There are many features of these worlds that are exciting from a research perspective:

1. The games provide immersive environments of greater detail than available with the majority of military simulations. The detail is for unit level interactions and not for large-scale simulations, but at the unit-level, these games have amazingly complex and real-world environments.
2. Some games provide well-defined interfaces for connecting external software that can control an entity in the game. These dynamically loaded libraries (DLLs) make it possible to use the commercial game without have access to the source code. Games such as Quake, Half-Life, and Descent all publish DLLs.
3. The games are cheap - \$49.95.
4. The games come with editors that allow a user to create their own environments in which to play.
5. The game code is extremely robust and bug free. It runs right out of the box, with out complex installation, licensing agreements, updates, etc.
6. University students are extremely excited about working on computer games.

All of these reasons make computer games an attractive arena for academic research on many of the issues related to computer-generated forces. The most significant overheads are creating the interface between the DLL and the AI system that the researcher is using. Other than that, the overhead in using a computer game is much less than using a military simulation system such as ModSAF/JSAP.

For now, the best support in computer games is from action games where research on single characters or groups of characters is easiest to pursue. However, I expect this to expand to strategy games where command level research can be studied.

5. The Soar Quakebot

Over the last two years, we have been experimenting with research in computer game AI within action games. The next three sections summarize our experiences. An expanded version of these sections can be found in [4].

The Soar Quakebot plays the death match version of Quake II. Quake II is a popular commercial computer game. In a death match, players exist in a "level", which contains hallways and rooms. The players can move through the level, picking up "powerups", such as weapons, ammo, armor, and health, and firing weapons. The object of the game is to be the first to kill the other players a specified number of times. Each time a player is shot or is near an explosion, its health decreases. When a player's health reaches zero, the player dies. A dead player is then "reborn" at one of a set of spawning sites within the level. Powerups are distributed throughout the level in static locations. When a powerup is picked up, a replacement will automatically regenerate in 30 seconds. Weapons vary according to their range, accuracy, spread of damage, time to reload, type of ammo used, and amount of damage they do. For example, the shotgun does damage in a wide area if used close to an enemy, but does no damage if used from a distance. In contrast, the railgun kills in a single shot at any distance, but requires very precise aim because it has no spread of the projectiles.

The Soar Quakebot controls a single player in the game. As shown in Figure 1 on the next page, the Soar Quakebot reasoning code currently runs on a separate computer and interacts with the game using the Quake II interface DLL. C code, which implements the Soar Quakebot's sensors and motor actions, is embedded in the DLL along with our inter-computer communication code, called Socket I/O. Socket I/O provides a platform-independent mechanism for transmitting all perception and motor information between the Quakebot and the game.

The Quakebot uses Soar [6] as its underlying AI engine. All the knowledge for playing the game, including constructing and using an internal map, is encoded in Soar rules. The underlying Quake II game engine updates the world and calls the DLL ten times a second. On each of these cycles, all changes to the bot's sensors are updated and any requested motor actions are initiated.

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 consuming 5-10% of the processing of a 400MHz Pentium II running Windows NT.

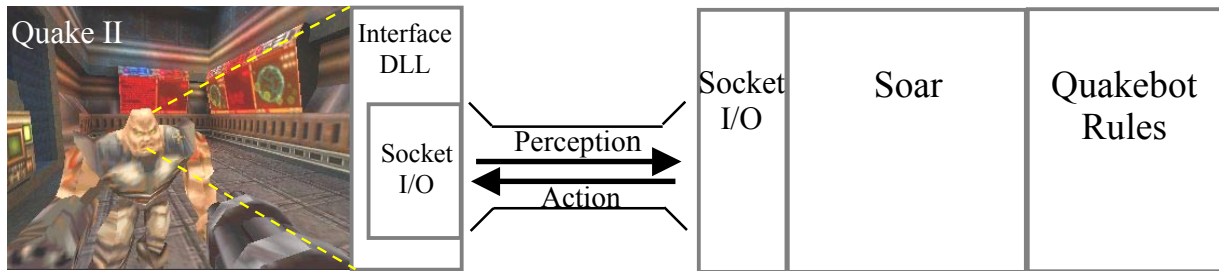


Figure 1: Interface between Quake II and the Soar Quakebot

The Soar Quakebot is designed based on the principles developed early on for controlling robots using Soar and then extended in our research on simulating military pilots in large scale distributed simulations [3]. Soar is an engine for making and executing decisions - selecting the next thing the system should do and then doing it. In Soar, the basic objects of decision are called *operators*. An operator can consist of primitive actions to be performed in the world (such as move, turn, or shoot), internal actions (remember the last position of the enemy), or more abstract goals to be achieved (such as attack, get-item, goto-next-room) that in turn must be dynamically decomposed into simpler operators that ultimately bottom out in operators with primitive actions. These primitive actions are implemented by if-then rules.

The basic operation of Soar is to continually propose, select, and apply operators to the current state via rules that match against the current state. When an abstract operator is selected that cannot be applied immediately, such as get-item, then a substate is generated. For this substate, additional operators are then proposed, selected and applied until the original operator is completed, or the world changes in such a way as to lead to the selection of another operator.

Figure 2 shows a subset of the operator hierarchy in the Quakebot. This is a small part of the overall hierarchy, but includes some of the top-level-operators, such as wander, explore, attack, and those that are used in the substate that can arise to apply the collect-powerups operator.

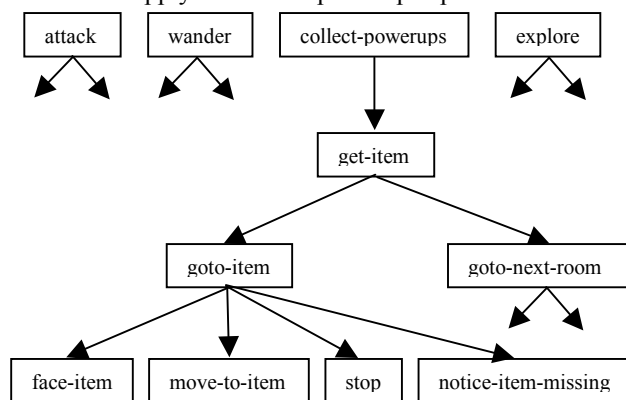


Figure 2: Partial operator hierarchy

Rules encode all of Soar's long-term procedural knowledge, and they are matched against the states stored in Soar's global declarative *working memory*. Working memory holds all of the bot's information about the current situation, including perception, elaborations of perception, data structures representing the map of the game, etc. All rules that successfully match working memory *fire* in parallel to change working memory by either adding or deleting declarative structures.

Below is a list of the main tactics the Quakebot uses. These are implemented across the top-level operators.

- Collect-powerups
 - 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 health and armor if low on them
 - Pickup up other good weapons/ammo if close by
- Attack
 - Use circle-strafe (walk sideways while shooting)
 - Move to best distance for current weapon
- Retreat
 - Run away if low on health or outmatched by the enemy's weapon
- Chase
 - Go after enemy based on sound of running
 - Go 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

6. Anticipation

Although the Quakebot as described above can react to different situations and opponents, as of yet, it and other game AIs do not anticipate or adapt to the behavior of other players. The following quote from Dennis (Thresh) Fong, the Michael Jordon of Quake, gives some insight

into the importance of anticipation (Newsweek, November 1999):

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.

These tactics can be added manually for specific locations in a specific level of a game. For example, we could add tests that if the bot is ever in a specific location on a specific level and hears a specific sound (the sound of the enemy picking up a weapon), then it should set an ambush by a specific door. Unfortunately, this is the approach currently used in computer games and it requires a tremendous effort to create a large number of tactics that work only for the specific level.

Instead of trying to encode behaviors for each of these specific situations, a better idea is to attempt to add a general capability for anticipating an opponent's actions. From an AI perspective, anticipation is a form of planning; a topic researchers in AI have studied for 40 years. The power of chess and checkers programs comes directly from their ability to anticipate their opponent's responses to their own moves. Anticipation for bots in first-person shooters (FPS) has a few twists that differentiate it from the standard AI techniques such as alpha-beta search.

1. A player in a FPS does not have access to the complete game state as does a player in chess or checkers.
2. The choices for action of a player in a FPS unfold continuously as time passes. At any time, the player can move, turn, shoot, jump, or just stay in one place. There is a breadth and depth of possible actions that quickly make search intractable and requires more knowledge about which actions might be useful.

However, as we developed the Quakebot, we found that in order to improve the behavior of the bot, we were forced to add more and more specialized tactics. Our approach to anticipation is to have the Quakebot create an internal representation that mimics what it thinks the enemy's internal state is, based on its own observation of the enemy. It then predicts the enemy's behavior by using its own knowledge of tactics to select what it would do if it were the enemy. Using simple rules to internally simulate external actions in the environment, the bot either forward projects until it gets a useful prediction, or there is too much uncertainty as to what the enemy would do next. The prediction is used to set an ambush or deny the enemy an important weapon or health item.

Anticipation requires adding knowledge about when it should be used, how it is done, and how the results are used to change behavior. These map on to the following structures in the Quakebot:

1. Proposal and selection knowledge for a predict-enemy operator.
2. Application knowledge for applying the predict-enemy operator.
3. Proposal knowledge for selecting operators that will use the predictions.

6.1 Predict-Enemy Proposal and Selection

When should the Soar Quakebot attempt to predict the enemy's behavior? It should not be doing it continually, because of the computational overhead and the interference with other activities. It shouldn't do it when it has absolutely no idea what the state of the other bot is and it also shouldn't do it when any prediction will be ignored because the bot already knows what to do. The Soar Quakebot attempts to anticipate an enemy when it senses the enemy (so it knows some things about the enemy's state), and the enemy is not facing the bot and is outside the range of its currently selected weapon (otherwise the bot should be attacking). The Quakebot will also attempt to anticipate the enemy if the enemy has just disappeared from view, such as when it has left a room through a doorway.

Figure 3 shows an example where the Quakebot (lower left) sees its enemy (upper center) heading north, on its way to get a desirable object (the heart). This corresponds to the situation described above and causes the Quakebot to propose and select the predict-enemy operator.

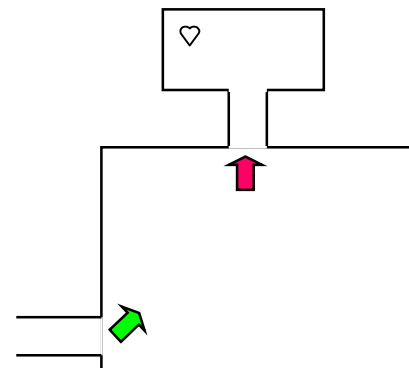


Figure 3: The predict-enemy operator is selected. One important aspect of Soar is that if the enemy turns toward the Quakebot instead of continuing north, the predict-enemy operator will be retracted (proposal rules implement justification-based reasoning maintenance) so that the Quakebot can select the attack operator and not be caught napping.

6.2 Predict-Enemy Application

Once the decision has been made to predict the enemy's behavior (via the selection of the predict-enemy operator), the next stage is to do it. Our approach is straightforward. The Quakebot creates an internal representation of the enemy's state based on its perception of the enemy and then uses its own knowledge of what it would do in the enemy's state to predict the enemy's actions. Thus, we will assume that the enemy's goals and tactics are essentially the same as the Quakebot's. This is the same approach that is taken in AI programs that play most games, such as chess or checkers. However, in this case the actions that are taken are not moving a piece on a board but are the movement of a Quakebot through its world using perception and motor commands.

The first step is to create the internal representation of the enemy's situation so that the Quakebot's tactics can apply to them. This is easy to do in Soar because Soar already organizes all of its information about the current situation in its state structure in working memory. All that needs to be done is that when the predict-enemy operator is selected and a substate is created, that state needs to be transformed into a state that looks like the top-level state of the enemy. The internal representation of the enemy's state is only approximate because the Quakebot can sense only some of it and must hypothesize what the enemy would be sensing. Surprisingly, just knowing the enemy's position, health, armor level, and current weapon is sufficient to make a plausible prediction of high-level behavior of players such as the Soar Quakebot.

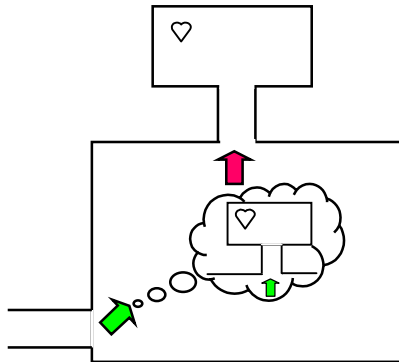


Figure 4: The Quakebot creates an internal representation of enemy's situation.

The second step involves letting the Quakebot's tactics work on its representation of the enemy's state. In the internal simulation of the example in the figures, rules would propose the collect-powerups operator in order to get the heart powerup. The Quakebot knows that the powerup is in the room to the north from prior explorations and attributes that knowledge to the enemy. Once collect-powerups is selected, a substate will be created, and then get-item, which in turn will have a substate, followed by goto-next-room. If this was not an

internal simulation, goto-next-room would lead to a substate in which goto-door is selected. However, for tactical purposes, the Quakebot does not need to simulate to that level of detail. To avoid further operator decompositions, a rule is added that tests that a prediction is being done and that the goto-next-room operator is selected. Its actions are to directly change the internal representation so that the Quakebot (thinking it is the enemy) thinks it has moved into the hall. Similar rules are added to short-circuit other operator decompositions. Additional rules are needed to update related data structures that would be changed via new perceptions (frame axioms), such as that health would go up if a health item was picked up. One additional rule is added to keep track of how far the enemy would travel during these actions. This information is used later to decide when to terminate the prediction. Figure 5 shows the updated internal representation of the Quakebot.

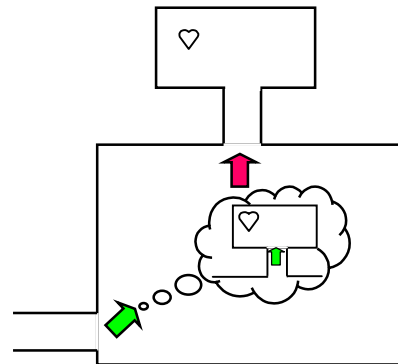


Figure 5: The Quakebot projects that enemy will move into hallway in pursuit of powerup.

The selection and application of operators continues until the Quakebot thinks that the enemy would have picked up the powerup. At that point, the enemy is predicted to change top-level operators and choose wander. Because there is only one exit, wander would have the enemy leave the room, going back into the hallway and finally back into the room where the enemy started (and where the Quakebot is).

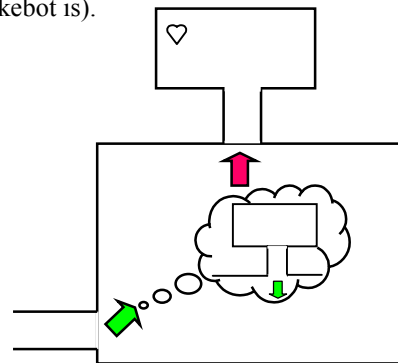


Figure 6: The Quakebot projects that enemy will return to the current room.

6.3 Predicting

Throughout this process, the Quakebot is predicting the behavior of the enemy. That prediction is only useful if the Quakebot can get into a tactical position that takes advantage of the prediction. Up until the enemy returns to the room, the prediction does not help the Quakebot. However, if the Quakebot hides by the hallway, it can get off a shot into the back or side of the enemy as it comes into the room. Thus, following the prediction, the Quakebot can set an ambush.

What are the general conditions for using the prediction: that is, what advantage might you get from knowing what the enemy is going to do? For Quake II, we've concentrated on the case where the bot can predict that it can get to a room before the enemy, and either set an ambush or deny the enemy some important powerup. This is done by continually comparing the distance that the enemy would take to get to its predicted location to the distance it would take for the Quakebot to get to the same location. For the current system, the number of rooms entered is used as a rough distance measure. In the example above, the Quakebot predicts that it will take the enemy four moves to get back to the current room, and it knows it is already in that room. Why doesn't the Quakebot stop predicting when the enemy would be coming down the hallway, which is three moves for it vs. one for the bot? The reason is that the Quakebot knows that it cannot set an ambush in a hallway, and thus waits until the predicted location is a room.

A prediction can also terminate when the Quakebot (thinking as the enemy) comes across a situation in which there are multiple possible actions for which it does not have a strong preference. This would have arisen in the previous example if there had been three doors in the north most room - with only two doors, the prediction would have gone forward because of the preference to avoid going back where you came from. When this type of uncertainty arises, the Quakebot abandons the prediction and attempts to get to the room it predicts the enemy is going to.

6.4 Using the Prediction

In the Soar Quakebot, three operators make use of the predictions created by predict-enemy: hunt, ambush, and deny-powerups. When a prediction is created that the enemy will be in another room that the Quakebot can get to sooner, hunt is proposed and it sends the bot to the correct room. Once in the same room that the enemy is predicted to be in, ambush takes over and moves the bot to an open location next to the door that the enemy is predicted to come through. In general, the bot will try to shoot the enemy in the back or side as it enters the room (shown below in the figure). But if the bot has the rocket

launcher, it will take a pre-emptive shot when it hears the enemy getting close (a la Dennis Fong, who was quoted earlier). Both of these ambush strategies have time limits associated with them so that the bot waits only a bit more time than it thinks the enemy will take to get to the room in which the bot has set the ambush.

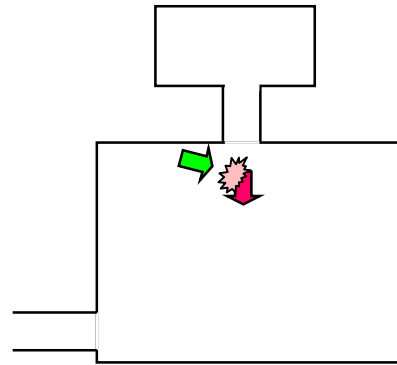


Figure 7: The Quakebot executes an ambush based on the results of its prediction.

6.5 Learning predictions

Inherent to Soar is a learning mechanism, called chunking, that automatically creates rules that summarize the processing within impasses. Chunking creates rules that test the aspects of the situation that were relevant during the generation of a result. The action of the chunk creates the result. Chunking can speed up problem solving by compiling complex reasoning into a single rule that bypasses the problem solving in the future. Chunking is not used with the standard Quakebot because there is little internal reasoning to compile out; however, with anticipation, there can be a long chain of internal reasoning that takes significant time (a few seconds) for the Quakebot to generate. In that case, chunking is perfect for learning rules that eliminate the need for the Quakebot to regenerate the same prediction. The learned rules are specific to the exact rooms, but that is appropriate because the predictions are only valid under special circumstances.

Below is an English version of a rule learned by the Quakebot.

```
If predict-enemy is the current operator
and
    there is an enemy with health 100,
    using the blaster, in room #11 and
    I am distance 2 from room #3
then
    predict that the enemy will go to
    room #3
    through door #7.
```

Compiled into the prediction is that the bot can get to room #3 before the enemy.

Once this rule is learned, the bot no longer needs to go through any internal modeling and will immediately

predict the enemy's behavior when it sees the enemy under the tested situations. The impact is that as the bot plays the game, it will build up a set of prediction rules, and it will make fast predictions in more situations. In fact, it might turn out that when it originally does prediction, the time to do the prediction sometimes gets in the way of setting an ambush or denying a powerup, but with experience that time cost will be eliminated. One possibility to create more challenging opponents is to pre-train Quakebots so that they already have an extensive set of prediction rules.

7. Limitations and Extensions

This section presents various limitations and extensions to the anticipation capabilities of the Soar Quakebot.

7.1 Recursive Anticipation

The Quakebot anticipates what the enemy does next. An obvious extension is for the Quakebot to anticipate the enemy anticipating its own actions. This recursion can go on to arbitrary depths, but the usefulness of it is probably limited to only a few levels. Recursive anticipation could lead the Quakebot to actions that are deceptive and confusing to the enemy. Although this might be useful in principle and for non-real-time computer games, such as real-time strategy games where there is more global sensing and a less frantic pace, it might be of only limited use for the Quakebot. The reason is that the bot must sense the enemy in order to have some idea of what the enemy's state is, and the enemy must sense the bot in order to have some idea of what the bot's state is. In Quake, there are only rare cases where the bot and the enemy can sense each other and one will not start attacking the other. However, we plan to do some limited investigation of recursive anticipation to find out how useful it is.

7.2 Enemy-Specific Anticipation

The current anticipation scheme assumes that the enemy uses exactly the same tactics as the Quakebot. However, there may be cases where you know beforehand that an opponent has different tactics, such as preferring different weapons. By incorporating more accurate models of an enemies weapon preferences, the Quakebot can decide to ambush an enemy in completely different (and more appropriate) rooms. We have made this extension by adding rules that encode weapon preferences that are specific to a player. These rules test the name of the opponent so that they apply only for the reasoning about the appropriate enemy.

7.3 Adaptive Anticipation

Unfortunately, an enemy's tactics and preference are rarely known beforehand. It is only through battle that one learns about the enemy. We've further extended the

Quakebot so that it notices the weapon preferences of its opponent during a match and dynamically modifies its model of that opponent.

A more general, but more difficult approach is to have the bot modify its knowledge each time the enemy does something unpredictable. The bot would continually try to build up its knowledge so that it can successfully predict the enemy. One final complexity is that the enemy will not be static, but will be adapting to the bot's tactics, and even to the bot's use of anticipation and its adaptation to the enemy. For example, after the first time an enemy is ambushed after getting the powerup from a dead-end room, it will probably anticipate the ambush and modify its own behavior. Our research into these issues will build on previous research we've done on learning from experience with dynamic environments [7].

8. Summary and Perspective

Our work with the Quakebot demonstrates how research on autonomous AI agents can be successfully pursued within the context of computer games. The research we've done has direct application to CGFs where it is desirable to have realistic, entity-level behavior. The same methods for anticipation could be directly applied to other CGFs systems. Moreover, it is easy to imagine doing further research within the context of the Quakebot or related characters on other aspects of human modeling such as the impact of emotion and battlefield stressors, or on coordination and cooperation in small group. We have also found these environments useful in doing small studies on the effect of different cognitive parameters on the skill-levels of our Quakebot. Although only preliminary, we have been able to study the impact on changes in reaction time, tactics level, and perceptual/motor skills on overall performance level in the game. Overall, we have found computer games to be a rich environment for research on human-level behavior representation.

9. Acknowledgments

The author is indebted to the many students who have worked on the Soar/Games project, most notably Michael van Lent, Steve Houchard, Joe Hartford, and Kurt Steinkraus.

This research was funded in part by grant N61339-99-C-0104 from ONR and NAWCTSD.

10. References

- [1] AAAI: Papers from the AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games, Technical Report SS-99-02, AAAI Press, 1999.

- [2] AAAI: Papers from the AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment, Technical Report SS-00-02, AAAI Press, 2000.
- [3] R. M. Jones, J. E. Laird, P. E. Nielsen, K. J. Coulter, P. G. Kenny, and F. V. Koss: "Automated Intelligent Pilots for Combat Flight Simulation" AI Magazine, 20(1), 27-42, 1999.
- [4] J. E. Laird: "It Knows What You're Going To Do: Adding Anticipation to a Quakebot" Papers from the AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment, Technical Report SS-00-02, AAAI Press, 2000.
- [5] J. E. Laird and M. van Lent: "Human-Level AI's Killer Application: Computer Game AI" To appear in Proceedings of AAAI 2000, Austin, TX, August 2000.
- [6] J. E. Laird, A. Newell, and P. S. Rosenbloom: "Soar: An architecture for general intelligence: Artificial Intelligence, 33(3), 1-64, 1987.
- [7] J. E. Laird, D. J. Pearson, and S. B. Huffman: "Knowledge-directed Adaptation in Multi-level Agents" Journal of Intelligent Information Systems, 9, 261-275, 1997.
- [8] National Research Council, "Modeling Human and Organizational Behavior: Applications to Military Simulations" National Academy Press, Washington D.C. 1998.
- [9] S. Woodcock: "Game AI: The State of the Industry" Game Developer, 6(8), 1999.

Author Biography

JOHN E. LAIRD is a Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his B.S. from the University of Michigan in 1975 and his Ph.D. from Carnegie Mellon University in 1983. He is one of the original developers of the Soar architecture and leads its continued development and evolution. From 1992-1997, he led the development of TacAir-Soar, a real-time expert system that flew all the U.S. fixed-wing air missions in STOW-97. He was an organizer of two symposia on AI and computer games and has been a presenter at the last three Computer Game Developers' Conferences.