Developing an Artificial Intelligence Engine

Michael van Lent and John Laird
Artificial Intelligence Lab
University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
{vanlent,laird}@umich.edu

Introduction

As computer games become more complex and consumers demand more sophisticated computer controlled agents, developers are required to place a greater emphasis on the artificial intelligence aspects of their games. One source of sophisticated AI techniques is the artificial intelligence research community. This paper discusses recent efforts by our group at the University of Michigan Artificial Intelligence Lab to apply state of the art artificial intelligence techniques to computer games. Our experience developing intelligent air combat agents for DARPA training exercises, described in John Laird's lecture at the 1998 Computer Game Developer's Conference, suggested that many principles and techniques from the research community are applicable to games. A more recent project, called the Soar/Games project, has followed up on this by developing agents for computer games, including Quake II and Descent 3. The result of these two research efforts is a partially implemented design of an artificial intelligence engine for games based on well established AI systems and techniques.

The Soar/Games project has interfaced the Soar artificial intelligence architecture with three games developed as part of the project and two commercial games, Quake II and Descent 3. The Soar architecture is the result of 15 years of research in the fields of artificial intelligence and cognitive psychology at various research universities. The interface between Soar and the games, using either sockets or the Tcl language, includes as many as 80 sensors and 20 actions. Complex agents, that include the ability to plan and learn new knowledge, have been developed for the non-commercial games. Simple agents have been created for Quake II and Descent 3 and complex agents, that share a large amount of knowledge, are under development.

The Soar/Games project has a number of benefits for both the research community and the game developer community. For the research community, the Soar/Games project has already provided environments for testing the results of research in areas such as machine learning, intelligent architectures and interface design. The difficult aspects of the Soar/Games project have also suggested a number of new research problems relating to knowledge representation, agent navigation and human-computer interaction. From a game development perspective, the main goal of the Soar/Games project is to make games more enjoyable by making the agents in games more intelligent and realistic. If done correctly, playing with or against these AI agents will more closely capture the challenge of playing online. An AI engine will also make the development of intelligent agents for games easier by providing a common inference machine and general knowledge base that can be easily applied to new games.

The division of labor in the DARPA project first suggested the concept of an artificial intelligence engine that consists of three components. During that project, one programmer worked on the inference machine, one on the interface to the simulator, and three programmers created the knowledge base. The Soar architecture, used in the DARPA project, will serve as the AI engine's inference machine. The interface between Soar and the game or simulator must be designed separately for each application but a number of general principles have been developed to guide the interface design. The knowledge base is generally the most time-consuming component of the AI engine to develop. However, a carefully designed knowledge base can easily be applied to multiple games in the same genre somewhat offsetting the cost of development with the benefit of reusability.

The main advantage of the AI engine approach is exactly this reusability of the engine and especially the game independent behavior knowledge. Rather than develop the AI for a new game from scratch, a programmer can implement an interface to the AI engine and take advantage of the inference machine and pre-existing knowledge bases. As part of developing the AI engine we plan on creating a general knowledge base containing information applicable to any first person perspective action game. Similar knowledge bases for other game genres could also be developed and reused across multiple games. Additionally, the operator-based nature of the knowledge base, as required by Soar, is modular, allowing programmers to mix and match tactics, behaviors and goals as appropriate for their game.

This paper will describe our artificial intelligence engine design and give examples of how the techniques and systems incorporated make agents in games more intelligent. The next section will present five requirements an AI engine should fulfill and describe some common approaches to game AI against which our engine will be compared. The next three sections will describe the components of the AI engine and discuss how each is influenced by the requirements. Finally, the conclusion will detail which aspects of the engine have been implemented and which aspects still need work. Additionally, a number of more advanced AI techniques will be discussed with an eye towards future inclusion in the engine.

Artificial Intelligence Engine Requirements

An effective artificial intelligence engine should support agents that are:

1. Reactive
2. Context Specific
3. Flexible
4. Realistic
5. Easy to Develop

Reactive agents respond quickly to changes in the environment and those reactions are specific to the current situation. Context specific agents ensure that their actions are consistent with past sensor information and the agent's past actions. Flexible agents have a choice of high level tactics with which to achieve current goals and a choice of lower level behaviors with which to implement current tactics. Realistic agents behave like humans. More specifically, they have the same strengths has human players as well as the same weaknesses. Finally, an artificial intelligence engine can make agent development easier by using a knowledge representation that is easy to program and by reusing knowledge as much as possible. Each of the components of the artificial intelligence engine must be carefully designed to implement the five requirements discussed above.

The common approaches currently used in computer games generally excel at some of the requirements listed above while falling short in others. For example, stimulus-response agents just react to the current situation at each time step with no memory of past actions or situations. This type of agent is generally very responsive because, without contextual information, the proper reaction to the current situation can be calculated very quickly. Stimulus-response agents can also implement multiple behaviors but aren't easily able to represent higher level tactics. Script-based agents, on the other hand, naturally make use of contextual information but can be less reactive. These agents have a number of scripts, or sequences of actions, one of which is selected and executed over a number of time steps. Once a script is selected all the actions performed are consistent with the context and goals of the script. However, if the situation changes, script-based systems can be slow to change scripts or stuck executing a irrelevant script which makes them less reactive.

Perhaps the most common approach to building intelligent agents in games is to use C code to implement the AI with a large number of nested if and case statements. As the agents get more complex, the C code that implements them becomes very difficult to debug, maintain and improve. A more constrained language, which better organizes the conditional statements, could be developed but we believe this language would turn out to be very similar to the Soar architecture.

The Inference Machine is Key

The inference machine is the central component of the AI engine design because it sets forth constraints that the other components must meet. The job of the inference machine is to apply knowledge from the knowledge base to the current situation to decide on internal and external actions. The agent's current situation is represented by data structures representing the results of simulated sensors implemented in the interface and contextual information stored in the inference machine's internal memory. The inference machine must select and execute the knowledge relevant to the current situation. This knowledge specifies external actions, the agent's moves in the game, and internal actions, changes to the inference machine's internal memory, for the machine to perform. The inference machine constantly cycles through a perceive, think, act loop, which is called the decision cycle.

1. Perceive: Accept sensor information from the game
2. Think: Select and execute relevant knowledge
3. Act: Execute actions in the game

The inference machine influences the structure of the knowledge base by specifying the types of knowledge that can be used and how that knowledge is represented. For example, a reactive inference machine, with no internal memory, would limit the knowledge base to stimulus-response knowledge represented as rules of the form "if X is sensed then do Y." The knowledge base couldn't contain high level goals because, without any internal memory, the inference machine couldn't remember the current goal across the multiple decision cycles needed to achieve it. Thus, a feature of the inference machine, the lack of internal memory, effects the knowledge base by limiting the types of knowledge included. A second example is how the speed of the inference machine constrains the speed of the interface. Because the interface must provide updated sensor data at the beginning of each decision cycle, the amount of time it take the inference machine to think and act is the amount of time the interface has to extract the sensor data. If the interface is too slow, the inference machine will be selecting incorrect actions due to out of date sensor information.

The most characteristic details of an inference machine are how it implements the think step of the decision cycle and any internal actions of the act step. For example, during the think step a stimulus-response inference machine compares each stimulus-response rule to the current sensor information. One rule is selected from the rules that match according to the specific inference machine's selection mechanism. A common mechanism is to order the rules by priority and execute the highest priority rule that matches. Since a stimulus-response machine doesn't have any internal memory there aren't any internal actions to be supported.   A slightly more complex inference machine might include a simple form of internal memory by allowing the knowledge to select from a number of modes (attack, retreat, explore…) which influence behavior. Separate rules would be used for each mode and rules could change the machine's internal mode of behavior. Agents that use stimulus-response inference machines, usually with some form of behavior modes, are common in the early action games. These agents usually sit in "sleep" mode until they sense the player and then change to an "attack" mode. Stimulus-response inference machines support agents that are very reactive but tend not to be very context specific, flexible or realistic.

A second class of inference machines common in games use scripted sequences of actions to generate the agent's behavior. At specific points in the game or when the agent senses certain conditions, the inference machine begins to execute one of the scripts stored in its knowledge base. Once a script is selected the inference machine performs the actions in sequence over a number of decision cycles. The inference machine's internal memory stores the agent's place in the script and possibly some details of previous sensor information or actions used to slightly customize the remainder of the script. More complex scripts include branch points in the sequence of actions where sensor inputs, such as the human player's responses to earlier actions, can influence the remainder of the actions in the script. Agents that use script-based inference machines are common in adventure and interactive fiction games where agents interact with players through scripted conversations. Usually, once the script has been completed, the agent switches to a reactive inference machine and a behavior mode based on the player's reactions during the script. Script-based inference machines tend to be less reactive than stimulus-response machines but their behavior is more context specific and somewhat more realistic.

As an inference machine, the Soar architecture combines the reactivity of stimulus-response machines with the context specific behavior of script-based machines. Additionally, agents based on Soar are flexible in that they can respond to a given situation in multiple different ways. In Soar, knowledge is represented as a hierarchy of operators. Each level in the hierarchy represents a successively more specific representation of the agent's behavior. The top operators in the hierarchy represent the agent's goals or modes of behavior. The operators at the second level of the hierarchy represent the high level tactics the agent uses to achieve the top level goals. The lower level operators are the steps and sub-steps, called behaviors, used by the agent to implement the tactics. In any given decision cycle Soar can select one operator to be active at each level of the hierarchy.

As shown in figure 1, an agent that plays Quake II might have a top level "Attack" goal with various tactics for attacking at the second level of the hierarchy. The behaviors and sub-behaviors that implement each attack tactic would fill out the lower levels. Because Soar considers changing each operator every decision cycle it is very reactive. If the situation suddenly changes, the inappropriate operators will immediately be replaced with operators more suitable to the new situation. On a 300 MHz Pentium II machine Soar can handle 6-10 agents

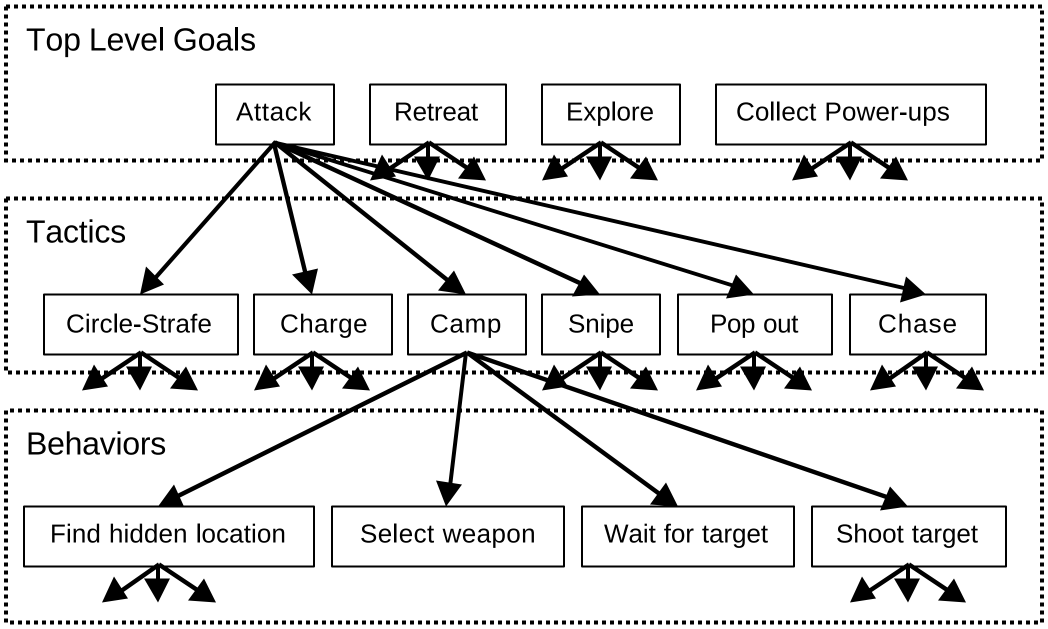

Figure 1: A portion of a sample operator hierarchy for an action game such as Quake II or Descent 3. This hierarchy has four top-level goals. The Attack goal can be implemented by any of six tactics. The Camp tactic has four behavior sub-operators, some of which have sub-operators of their own.

allowing each to perform 5 decision cycles per second. Unlike Soar, script-based inference machines usually don't consider changing scripts until the current script is finished. This can sometimes be seen in role playing games when the player attacks a computer controlled agent in the middle of a conversation and the agent doesn't fight back until it has finished its lines. Soar, on the other hand, could change from a "converse" high level operator to a "defend" operator in a single decision cycle.

Because operators can remain selected for many decision cycles, Soar can easily support context specific sequences of actions in like script-based machines. In Soar, a script would take the form of a single high level operator and a sequence of sub-operators. Soar would select the high level operator to execute the script and that operator would remain selected (or persist) through out the execution of the script. Each sub-operator would be selected in turn and perform a step in the script. Since each operator has its own selection conditions branching scripts, as described above, are also easy to implement. If at any point the situation changed making the script inappropriate, the high level operator would be replaced and the script wouldn't continue.

Unlike both stimulus-response machines and script-based machines, the Soar architecture includes a full internal memory that can store a variety of types of information. In addition to the persistence of selected operators, the persistence of information in the internal memory supports context specific behavior. For example, if an enemy moves out of sight, a pure stimulus-response machine will

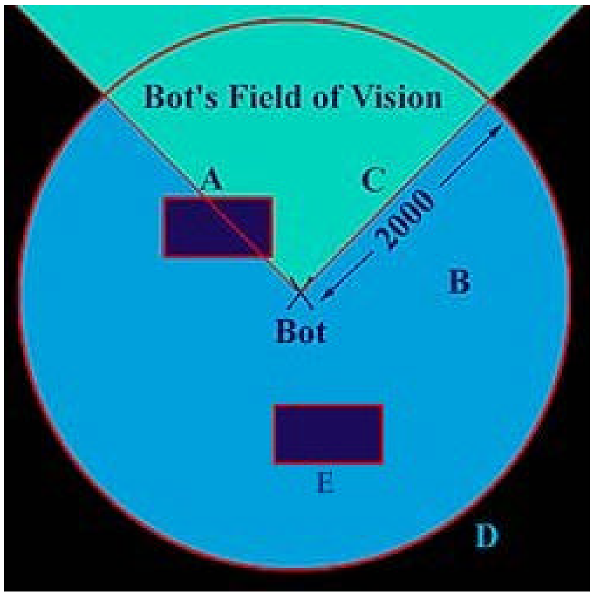

Figure 2: An intelligent agent, or bot, in Quake II will receive sensor information about an opponent at position C but won't sense the opponents in positions A (no line of sight), B (out of field of view), D (out of sight range and field of vision) or E (no line of sight and out of field of vision).

immediately forget that the enemy exists. A script-based machine will fare slightly better because it will at least have an attack-enemy script selected; but it won't have any sensor information about the enemy with which to implement the script. The Soar architecture can easily store the most recent sensor information about the enemy in the internal memory and, if the enemy disappears, fall back on these memories. Furthermore, after the enemy disappears Soar operators can modify the internal memories about the enemy based on projections of the enemy's behavior.

Finally, both stimulus-response machines and script-based machines are inflexible in that they generally only have one way to respond to each situation. Soar's hierarchical operator representation can easily support multiple tactics to achieve each goal and multiple behaviors to implement each tactic. Each operator is represented by a set of selection conditions, tests on sensor information and internal memory, and a set of conditional actions. When an operator needs to be chosen at a level of the hierarchy all the suitable operators with matching selection conditions are considered. Another form of knowledge, called search control knowledge, is used to assign priorities to the candidate operators. Once an operator is chosen, it remains the current operator at that level until its selection conditions are no longer met. While an operator is selected its actions can be executed if the action's conditions are also met. Multiple tactics or behaviors can be implemented by creating multiple operators with similar selection conditions but different actions that result in the same result. For example, the "Attack" goal from the Quake II example above (see figure 1) can be achieved via a number of different tactic operators and each tactic operator could be implemented by a variety of behavior operators.

The Interface is Key

Enemy Sensor Information

^name	[string]
^classname	[string]
^skin	[string]
^model	[string]
^health	[int]
^deadflag	[string]
^weapon	[string]
^team	[string]
^waterlevel	[int]
^watertype	[string]
^velocity	
^x	[float]
^y	[float]
^z	[float]
^range	[float]
^angle-off	
^h	[float]
^v	[float]
^aspect	
^h	[float]
^v	[float]
^sensor	
^visible	[bool]
^infront	[bool]

Movement Commands

^thrust	[forward/off/backward]
^sidestep	[left/off/right]
^turn	[left/off/right]
^face	[degrees]
^climb	[up/off/down]
^aim	[degrees]
^look	[up/off/down]
^jump	[yes/no]
^centerview	[yes/no]
^run	[on/off]
^facetarget	[on/off]
^movetotarget	[on/off]
^leadtarget	[on/off]

Weapon Control Commands

^change	[weapon]
^continuousfire	[on/off]
^fireonce	[yes/no]

Misc. Commands

^dropnode	[yes/no]
^disconnect	[yes/no]
^wave	[int]
^say	[string]
^say_team	[string]
^selecttarget	[target]

Figure 3: Samples of the sensor information and actions implemented by the Quake II interface. The sensor information the inference engine receives about an enemy entity is shown on the left. On the right are many of the external commands the inference engine can issue.

One of the lessons learned as a result of the Soar/Games project is the importance of a carefully designed interface between the inference machine and the environment in which the agent lives. The interface extracts the necessary information from the environment and encodes it into the format required by the inference machine. Each new game requires a new interface because the details of the interaction and the content of the knowledge extracted vary from game to game. For example, the interface to Descent 3 must give the agent the ability to move or rotate in all six degrees of freedom, while Quake II requires only four degrees of freedom (plus a jump command). Similarly, Quake II requires one set of weapon control commands while Descent 3 requires two sets because the game includes primary and secondary weapons. Each game includes it's own special features which require customized interface programming to support.

However, each of the interfaces we've designed has shared two common principles. The first is that the interface should mimic the human's interface as closely as possible. Thus, the inference machine gets all the information available to the human player and no additional information. For example, as shown in figure 2, an opponent in Quake II must meet three requirements to be sensed. First, the opponent must be in the agent’s sight range. Second, the opponent must be in the agent’s visual field, which corresponds to the visual field displayed on the screen. Finally, there must be an unblocked line of sight between the agent and the opponent. When an opponent meets all three requirements the interfaces sends sensor information about that opponent to the inference machine. The second principle is that the interface should access the game’s data structures directly and avoid the difficult problems involved in modeling human vision. Thus, the interface shouldn’t attempt to extract sensor information from the image displayed on the screen alone.

One of the common complaints about game AI is that the agents are allowed to cheat. Cheating can take the form of using extra information that the human player doesn’t have or being given extra resources without having to perform the actions required to acquire them. Requiring the intelligent agents to use the same sensor information, follow the same rules and use the same actions as the human players eliminates cheating and results in realistic agents. All of the sensor information and actions available through the Quake II interface are also available to a human player (see figure 3). The cost is that these realistic agents will require more knowledge and better tactics and behaviors to challenge human opponents. Hopefully, using a pre-existing knowledge base will free the AI programmers to develop the complex tactics and knowledge necessary to implement challenging agents that don’t cheat. Because these agents don’t cheat, but instead play smarter, they’ll be more similar to human opponents and more fun to play against.

The Knowledge is Key

The final component of our AI engine is the knowledge base of game independent goals, tactics and behaviors. As an example, the knowledge base for the DARPA project included almost 500 operators that allowed the agents to fly more than ten different types of missions including air to air combat, air to ground combat and patrols. In our AI engine design, this knowledge base doesn't include game specific information but instead focuses on goals, tactics and behaviors that apply to any game within a genre. For example, in the first person perspective genre, the circle-strafing tactic would be a component of the behavior knowledge base. To apply the AI engine to a specific game, a small amount of game dependent information is added which would allow the circle-strafing tactic to be applied differently according to the game dynamics. Descent 3’s flying agents might circle-strafe in three dimensions, while Quake agents would circle-strafe in only two dimensions. The job of the AI programmer would then be to tailor the general knowledge base to the game being developed and add additional game specific knowledge and personal touches.

When the general knowledge base is being developed it is important to keep the five agent requirements (reactive, context specific, flexible, realistic, easy to develop) in mind. Some of these requirements are mainly supported by features of the inference machine and/or interface. The knowledge base simply needs to ensure that it makes use of these features. For example, a knowledge base that takes advantage of the hierarchical goal structure and internal memory of the Soar architecture by including persistent operators and internal memories will result in agents with context specific behavior. Encoding many high level operators, some of which apply to any situation, gives the agent flexibility in its choice of tactics. Similarly, encoding many low-level operators that implement the tactics in more than one way, gives the agent flexibility in its choice of behaviors. Flexible agents, with a choice of responses to any situation, won't react the same way twice making the game more fun and more replayable.

Realism is one of the main areas in which intelligent agents in games tend to fall short. Frequently, the agents take actions a human player would never take or miss actions that would be obvious to a human player. Unrealistic behavior can be very distracting and usually is the cause of complaints that agents are "stupid." Frequently, the cause of unrealistic behavior is unrealistic or missing knowledge in the knowledge base. When creating and testing a knowledge base it is important to constantly ask "What would a human do?" and tailor the knowledge to match.

Future Directions

Currently the Soar architecture has been interfaced to five different games. Three of these games are fairly simple variations on Pac-man and tank combat action games. The two commercial games, Quake II and Descent 3, are more complex and have involved creating more sophisticated interfaces and knowledge bases. A simple agent for Quake II has been developed that uses around 15 operators in a three level hierarchy to battle human opponents. While this simple Quake-bot isn't an expert player it does easily beat beginners and provides a challenging opponent for intermediate Quake II players. A simple Descent 3 agent has also been developed that seeks out and destroys monsters in the Descent 3 levels.

The immediate future plans for the Soar/Games project is to finish a more complex and complete implementation of the AI engine. The Soar architecture has recently been updated to version 8, which includes changes to improve reactivity and make Soar's learning mechanism easier to use. A full rewrite of the interface to Quake II will be complete by the end of February and the interface to Descent 3 is also being rewritten. Simple Quake II and Descent 3 knowledge bases have already been developed and tested and a more complex knowledge base, which will be used by both games, is currently being designed. Additionally, a speech interface to cooperative agents in Quake II is being developed which will allow a human player to act as an officer, giving voice commands coordinating the actions of a platoon of intelligent agent soldiers.

Once the initial implementation is complete, some of the ongoing research at the University of Michigan AI lab can be tested in the context of the games. One major area of research is automatically learning new knowledge from natural interactions with human experts such as instruction and observation. A very early experiment has shown that the KnoMic (Knowledge Mimic) system can learn new operators for the Quake II knowledge base based on observations of an expert playing the game. A similar system, which learns from expert instruction, also seems promising. A related research project at Colby college is using the Soar/Games AI engine to develop socially motivated agents that seek to satisfy internal needs and drives through social interaction and cooperation. One of the advantages of the Soar/Games project is that so many different areas of AI research, such as opponent modeling, agent coordination, natural language processing and planning, have the potential to be easily showcased in the context of computer games.

Acknowledgements

The research presented here and Soar/Games project would not exist without the hard work of a number of undergraduates and graduate students at the University of Michigan. Steve Houchard has worked tirelessly on the interface to Quake II. Kurt Steinkraus implemented and constantly improves the Soar side of the interface. Russ Tedrake developed the Descent 3 interface. Joe Hartford developed the initial interface to Quake II. Josh Buckman assisted Russ with the Descent 3 side of the project. Thanks to Intel for donating a number of the machines that have been used during the project. Finally, we’d like to thank Outrage Entertainment for giving us the chance to work with Descent 3 while it is being developed. Their cooperation and many hours of assistance made this project possible.