Robot Architectures for Believable Game Agents

Ian D. Horswill and Robert Zubek

Institute for the Learning Sciences, Northwestern University
1890 Maple Ave., Suite 300
Evanston, IL 60201
{ian,rob}@cs.nwu.edu

Abstract

Computer game character design and robotics share many of the same goals and computational constraints. Both attempt to create intelligent artifacts that respond realistically to their environments, in real time, using limited computation resources. Unfortunately, none of the current AI architectures is entirely satisfactory for either field. We discuss some of the issues in believability and computational complexity that are common to both fields, and the types of architectures that have been used in the robotics world to cope with these problems. Then we present a new class of architectures, called *role passing architectures* which combine the ability to perform high level inference with real-time performance.

Introduction

As both players and developers will readily admit, game agents do not behave as realistically as we would want. Even though first-person action games have greatly advanced since the times of Wolfenstein 3D, the precursor of the genre, game agents have not improved nearly as much, especially in emulating real human and animal behavior. What we would like to see instead are creatures that do not just perform their roles, but which are more believable in the sense of responding to situations in a way that humans and animals might respond. We want agents that:

- Hide behind corners to avoid being shot
- Look for good ambush sites
- Run away from more powerful enemies
- Are concerned about their own survival

In short, we would like game agents that act more like us.

In this paper we would like to discuss how advancements in robot programming could help with developing believable agents, and we present a robot architecture that can help game developers with agent programming.

What's hard about believability?

The problem with many game agents is that they're simultaneously too smart and too stupid. On one hand, they have very limited reasoning capabilities. On the other hand, the reasoning capabilities they do have may take advantage of super-human sensory or motor skills.

None of this is surprising given the practicalities of game design. No one is going to use an exponential-time planner to select actions when the system has to run in real-time and 95% of the CPU is reserved for rendering. Also, AI infrastructure and real-time game infrastructure don't mix very well.

Modeling human limitations

Both humans and, to varying degrees, other animals, have sophisticated cognitive abilities. However, those abilities are severely limited both in speed and attention. While people are very good at planning long-range strategies, they can't do it while simultaneously fighting off a horde of angry monsters. Human perception is very limited. We can't see behind ourselves. We can't see around corners. We're much better at detecting moving objects than static objects. Perceptual attention is also severely limited. When fighting one enemy, we are much less likely to even be aware of another enemy approaching. Finally, human short-term memory is extremely limited. When people are distracted from one task, they often forget their previous tasks completely.

Real military doctrine is based in large part on exactly these kinds of limitations. On a tactical level, soldiers do everything possible to avoid giving away their positions. On a strategic level, misdirection of the enemy is central to maneuver warfare.

Consequently, modeling of attentional limitations is crucial to believability in real-time simulations of time-limited behavior, such as combat.

How robotics can help

Unfortunately, today's robots don't share the problem of being too smart. They're just too stupid. However, the reasons for their limitations are essentially the same as the issues with believability.

Just as agents in computer games, robots need to be responsive to changes in the external environment. That

means they need to be constantly resensing their environments and reevaluating their course of action. However, there are serious limitations on the degree to which this is possible. Just as rendering can eat up arbitrary amounts of CPU time in games, vision can eat arbitrary amount of CPU time on robots. The high cost of sensory processing means that robots have to be extremely selective about what they sense and how often they update it. It also means that in practice relatively little CPU is left over for decision making.

Because of that, robot architectures may be particularly suitable for game AI, as they operate within both the computational constraints typical of a game engine, and perceptual constraints imposed by the physical world that we want to emulate.

A quick overview of robot architectures

Historically, robot architecture design has been approached from two angles – behavior-based system and symbolic systems.

Behavior-based architectures

Most autonomous robots use some variety of behavior-based architecture, at least for sensory-motor control. Behavior-based architectures are also being used increasingly in computer games, such as Crash Bandicoot and Jedi Knight.

Behavior-based systems consist of a set of simple, task-specific sensory and motor processes (i.e. behaviors) running in parallel, usually with some mechanism to arbitrate between them. The basic sensory and motor components are typically kept quite simple, so as to help guarantee real-time performance. For example, the subsumption architecture (Brooks 1986) limits the programmer to finite-state machines communicating scalar values over fixed communication channels.

Behavior-based systems offer a number of compelling advantages:

- They're fast and cheap
- They give hard real time guarantees
- They interface easily to most sensors and effectors
- They require little or no infrastructure to support them, so they're convenient to program on a bare machine with no O.S.
- They're easy for new programmers to understand

However, behavior-based systems also have serious limitations. For the most part, the only control structures they support are parallelism and infinite looping. More importantly, they only support very simple representations. Typical behavior-based architectures only allow users to pass simple scalar values (Brooks 1986, Maes 1991) or fixed-size vectors (Arkin 1987) between behaviors. In effect, this limits behavior-based systems to propositional logic, and makes it impossible for them to represent predicate logic (logic with variables) and quantified inference.

These limitations can seriously hinder the programmer. Because there's no good way of expressing parameters, Maes' solution to the blocks world (1991) involves having separate behaviors for every possible block motion. Behavior-based dialog system by Hasegawa et al. (1997) used separate behaviors for every possible utterance by every possible speaker. While this kind of behavior duplication is fine for small problems, it becomes prohibitive for large problems.

Symbolic architectures

Symbolic AI architectures, on the other hand, are typically Turing-complete. They allow full parameter passing, general control structures, including recursion, and more or less unrestricted representations, including manipulation of arbitrary dynamic tree structures. This flexibility allows the programmer to implement very general reasoning operations, including generative planning from first principles.

Of course, this flexibility comes at the cost of being unable to make real-time guarantees, or even guarantees of termination. First-order logic inference is equivalent to simulating a Turing machine, so depending on the specific problem, reasoning could take arbitrary time and space, or fail to terminate entirely. Furthermore, drawing inferences is generally an exponential-time operation.

A more down to Earth problem is that symbolic AI systems typically assume all information to be manipulated by the system is available in a centralized database. This means that the sensory systems have to continually keep the database up to date, often without any kind of back-channel form the reasoner to tell them what information is relevant to system's current task. Worse, in very dynamic environments such as combat games, the reasoner must continually re-update any inferences it has made based on the changing sensory data.

Many of the computational problems of inference come from the presence of variables in the rules. If patterns in the rules can contain variables, then those patterns must be matched using some form of tree- or graph-matching against the entries in the database. In the case of multiple matches, the rule may need to be fired on all of them. Although a number of excellent techniques exist for optimizing the matching processes, chained rule application is still exponential in the general case. If each rule can match 4 different entries in the database, then a 5 step sequence has 1024 possible matches.

Implementing logical inference using compact bit-vectors

Inference can, of course, be fast in special cases. One of the principal problems in knowledge representation is finding good special cases that are both fast and useful. We have developed a family of inference engines for real-time systems that allow us to compile all inference operations into straight-line code consisting mostly of bit-mask instructions. The principle limitations of the technique are that:

- It only supports predicates of one argument. That is, it supports predicates like near(X), but not distance(X,100).
- It doesn't support term expressions. This means it allows arguments to predicates to be object names, as in near(fido) or variable names, as in near(X), but not complex expressions, as in near(owner(fido)).
- It only supports a fixed, small set of object names chosen at compile time. Our current implementation allows 32 object names.

The last of these is mitigated by allowing the object names to be *indexical*, meaning that the system is allowed to use the names to mean different objects in different situations. This effectively means that you never use specific names like "fido", you use generic *role names* like TARGET, DESTINATION, THREAT, *etc*. From a programmer's standpoint, these roles are effectively just another variable binding mechanism. However, they are implemented very differently – roles are bound by the tracking components of the perceptual system and not by the inference system.

By limiting the set of roles in advance we can represent the complete extension of a unary predicate in a single memory word, one bit per role. For example:

	Agent	**Patient**	**Source**	**Destination**
in_range(x)	N	Y	N	Y
aiming_at(x)	N	Y	N	N
can_shoot(x)	N	Y	N	N

In robotics, the major appeal of this representation is that it is easy for sensory-motor systems to generate and update in real time. However, it's simplicity and speed is also convenient for computer games.

Suppose we want the creature to have a set of simulated sensors that report when it is near an object, when it is within firing range, when it is facing the object, etc. Each of these sensors would report its data as a bit-vector showing which roles were in range (or: nearby, facing, …). At run time, the game would keep a table mapping roles to the internal game objects to which they are bound. The game can then compute simulated sensor readings using the following C code:

```
typedef struct {
  /* which roles the predicate
     is true of */
  unsigned long true;
  /* which roles for which we know
     whether it's true */
  unsigned long know;
} predicate;


predicate sensor() {
  /* don't know anything yet */
  predicate reading = { 0, 0 };

  for (int role=0;
       role<total_roles; role++)
    if (role_binding[role] &&
        can_see(role_binding[role]) {
     /* we can sense the predicate
         for this role */
     reading.know | = 1<<role;
     if(sensor_internal_implementation(
              role_binding[role]))
        reading.true |= 1<<role;
    }

  return reading;
}
```

where role_binding[] is a table mapping the i^{th} role to its internal game object, can_see() is a function telling whether a given game object is in the agent's field of view, and sensor_internal_implementation() performs the actual sensing operation on game objects. The procedure allows the game to efficiently determine the set of objects for which the agent knows the truth of the predicate and also its truth for those objects for which it's known.

Having implemented primitive sensing, we can now implement forward-chaining, universally quantified inference rules, such as:

for all x, P(x) if Q(x) and R(x)

using simple bit-mask operations. For example, in C we would translate the rule:

for all x, can_shoot(x) if in_range(x) and aiming_at(x)

as:

```
can_shoot.true = in_range.true &
aiming_at.true;
```

which computes the inference rule for all values of x simultaneously using only one machine instruction plus

any loads and stores that may be necessary. It can also compute the know bits using:

```
can_shoot.know = in_range.know &
aiming_at.know;
```

although in this particular case the agent presumably always knows whether it is aiming at a given object, so a good compiler will optimize this to:

```
can_shoot.know = in_range.know;
```

It is straightforward to extend the representation to handle reasoning about goals and knowledge goals. See (Horswill 1998) for details.

Role passing

This technique is the basis of a class of architectures, called *role passing* architectures, which we believe combine (much of) the utility of symbolic reasoning architectures with the performance of behavior-based systems.

A role-passing system represents the agent's task and environment in terms of a set of situations and bindings of external objects to roles. Situations classify the current task and environment. For example, the agent might have a *search* situation, which would mean the agent was engaged in searching, and a *intruder containment* situation, which would mean the agent was attempting to prevent an intruder from moving outside some specified perimeter. Either, both, or none of those situations might be active at any given time. When the *search* situation is active, the agent will use the inference rules associated with it to solve the search task, and similarly with *intruder containment*. If both are active, it will run both rules and try to achieve both goals. The specifics of what intruder is being kept within what perimeter, and of what object is being searched for, are specified by role bindings.

A role passing system is composed of a set of inference rules, expressed in a simple forward-chaining rule language, and a set of sensory-motor systems that track designated objects, follow them, grab them, shoot at them, etc. The inference system tells the sensory-motor systems what to follow or grab by passing it a bit-vector specifying the role of the object to be followed or grabbed. On each cycle of its control loop, the system resenses all the properties of all the objects bound to roles, reruns all of its inference rules, and feeds the output of the inference rules back to the sensory-motor systems to retarget them. By rederiving inferences on every cycle, the system can continually adjust its activity as the environment changes. This is important to avoid the kinds of non-fluencies that are common in plan-based systems where a set of firing conditions are checked at the time the plan is initiated, but are not rechecked during the course of the plan. A classic example in robotics is a robot trying to deliver and package to a destination. If the package falls out of the robot's gripper enroute, many robot systems won't notice and will continue the delivery operation without reacquiring the package. Parallel examples are easy to construct in computer game design, such as monsters that continue to chase other monsters, even when the latter slip into pits of lava.

Using role passing techniques, we estimate that current mid-to-high end PCs could easily run a rule base of 1000 inference rules at 100 Hz (100 complete revisions of the system's inferences per second) using less than 1% of the CPU. However, this assumes that the perceptual systems (in the case of robots) or the graphics engine (in the case of games) could keep up with that speed, which is probably unrealistic.

Conclusion

Computer games and robotics share many of the same goals and computational constraints. We believe that role passing provides many of the benefits of symbolic reasoning systems, such as:

- Limited parameter passing, both to predicates and to behaviors
- Explicit subgoal decomposition
- Universally quantified inference, and
- Explicit reasoning about the agent's state of knowledge

while simultaneously the simplicity and real-time performance of behavior-based systems. On robot systems, the performance of role passing systems comes from taking into account the inherent limitations of the sensory-motor and short-term memory systems. Since these can only track a few objects simultaneously anyhow, we can simplify the inference system by having it operate directly upon the set of objects in short term memory, performing inferences on all of them, in parallel. This simplifies interfacing and also allows the system to stream data from the sensory systems, through the inference system, to the motor systems.

We are currently working implementing role passing architectures in Unreal. We are writing behavior-based sensory-motor controllers for the game and are porting our role passing compiler to support UnrealScript. Our intent is to use role passing to implement intelligent characters that:

- Find environmental features for ambush, hiding from enemies, and so on,
- Show animal-like hunting behavior (e.g. stalking prey), and
- Adopt interesting group strategies.

References

Arkin, R. C. 1987. Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior. In Proceedings of the IEEE Conference on Robotics and Automation, 264-71. Raleigh, NC: IEEE Press.

Brooks, R. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation,* RA-2 (1): 14-23.

Hasegawa, T., Nakano, Y. I., Kato, T. 1997. A Collaborative Dialog Model Based on Interaction Between Reactivity and Deliberation. In Proceedings of the First International Conference on Autonomous Agents, 83-87. Marina del Rey, CA: ACM Press.

Horswill, I. D. 1998. Grounding Mundane Inference in Perception. *Autonomous Robots* 5: 63-77.

Maes, P. 1989. How To Do the Right Thing, Memo No. 1180, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.