Cognitive Modeling for Computer Games

John Funge
Microcomputer Research Lab
Intel Corporation

www.cs.toronto.edu/ funge

Abstract

One of the major challenges associated with making compelling virtual worlds for computer games is populating them with intelligent characters. Most of today's characters are mindless automatons that follow an inflexible set of simple rules. Unfortunately, when we start to make characters try and think for themselves we can run into difficulties. For example, if we are not extra careful about how we represent a character's knowledge we can deprive them of "common sense" as they reason about the effects of their actions. Another important issue we consider is how to provide a convenient mechanism to allow the programmer to strike a balance between lots of fast pre-programmed behavior and more expensive run-time decision making. In addition, by using interval arithmetic to integrate sensing into our underlying theoretical framework, we enable characters to generate plans of action even when they find themselves in highly complex, dynamic virtual worlds. All these ideas are made concrete in the form of a cognitive modeling language (CML) we have developed. CML is used to express what a character knows, how that knowledge is acquired, and how it can be used to plan actions. It has enabled us to quickly, and easily, create various "intelligent" characters that inhabit a variety of different virtual worlds.

Introduction

Characters that are autonomous contribute heavily to the success of games like Doom. Unfortunately, gamers soon tire of the predictable computer controlled characters. They prefer the thrill of going head to head with fellow human avatars in "Deathmatch" versions. Computer characters with the wits to better match their human adversaries would therefore be in high-demand. In response, our work explores *cognitive modeling* for computer games. Cognitive models govern what a character knows, how that knowledge is acquired, and how it can be used to plan actions. Cognitive models are applicable to directing the new breed of highly autonomous, quasi-intelligent characters that are beginning to find use in game production. Moreover, cognitive models can play subsidiary roles in controlling cinematography and lighting.

We decompose cognitive modeling into two related subtasks: *domain knowledge specification* and *control specification*. One of the key benefits of such a separation is that code is easier to maintain. Once we have the correct knowledge about the world setup, we need only modify the control part when we want to generate new behavior. Characters can combine knowledge in ways that we did not think of in advance and they can use their knowledge to work out their own strategies for obtaining their goals. The approach is reminiscent of the classic AI formula that tries to promote modularity of design by separating out knowledge from control.

control + knowledge = behavior

Domain knowledge specification, or just domain specification, involves giving a character knowledge about its world and how it can change. Control specification involves directing the character to try and behave in a certain way within its world. Like other advanced modeling tasks, both of these steps can be fraught with difficulty unless developers are given the right tools for the job. To this end, we developed a cognitive modeling language, CML.

CML provides an intuitive way to give characters, and also cameras and lights, knowledge about their domain in terms of actions, their preconditions and their effects. We can also endow characters with a certain amount of"common sense" within their domain and we can even leave out tiresome details from their instructions. The missing details are automatically filled in at run-time by an integral reasoning engine which decides what the character must do to achieve the specified behavior.

Traditional AI style planning certainly falls under the broad umbrella of this description, but the distinguishing features of CML are the intuitive way domain knowledge can be specified and how it affords a game developer familiar control structures to focus the power of the reasoning engine. This forms an important middle ground between regular logic programming (as represented by Prolog) and traditional imperative programming (as typified by C). Moreover, this middle ground turns out to be crucial for cognitive modeling in computer games. In rapid-prototyping, reducing development time is, within reason, more important than fast execution. The game designer may therefore choose to rely more heavily on the reasoning engine. When run-time efficiency is also important our approach lends itself to an incremental style of development. We can quickly create a

working prototype that can be gradually made more efficient by including more and more heuristic control information to narrow the focus of the reasoning engine.

Cognitive Modeling

The *situation calculus* is a well known formalism for describing changing worlds and provides the theoretical underpinnings for CML. Fortunately, from the CML user's point of view, the underlying theory is *completely hidden*. In particular, a user is *not* required to type in axioms written in first-order mathematical logic. Instead, CML provides an intuitive interaction language that resembles natural language, but has a clear and precise mapping to the underlying formalism.

Domain Specification

Domain specification involves giving a character knowledge about how its world changes. The underlying idea is that the world is conceived of consisting of a sequence of *situations*. Each situation is a "snapshot" of the state of the world. Any property of the world that can change over time is known as a *fluent* and a fluent's value can change by performing *actions*.

The possibility of performing an action is determined by user specified *preconditions*. For instance, in a simple maze example it is possible to move in some compass direction d, provided the cell we are moving to is free, and has not been visited before.

action *move*(d) **possible when** c = adjacent(position,d) &&
Free(c) && !member(c,visited);

Specifying the effect of an action is just as straightforward. Returning to the maze example, we can use CML to specify the initial position as the start point of the maze, and the effect of moving to a new cell to update the position fluent accordingly.

initially position = start;
occurrence *move*(d) **results in** position = adjacent(p_{old},d)
when position = p_{old};

Unfortunately, specifying the effects of actions turns out to be complicated by the problem of how to specify what does not change. If characters are going to start thinking for themselves they need to possess the common sense notion that unless told otherwise they should assume things stay the same. This is known as the *frame problem* and we adopt the approach to dealing with it described in (Reiter 1991). That is, for each fluent we automatically collect up all the action effects specified by the user and make the assumption that these are the only effects. For each fluent this assumption is represented internally by what is known as a *successor-state axiom*.

Control specification

With what we have described so far it would be possible to do regular planning. Unfortunately, it often turns out to be unrealistic to expect a character to generate a plan from scratch. Even for simple problems the search space is simply too large. What is needed is a way to narrow the search space. This is where complex actions come in (Levesque *et al.* 1997). A complex action is an action that is made up of other actions using familiar programming constructs such as "while" loops and less familiar ones like "nondeterministic choice". Complex actions are implemented in CML and allow the user to write what amounts to sketch plans. These sketch plans allow the user to give the character an outline of what to do. At one extreme a highly detailed outline will heavily restrict the character's behavior and at the other extreme the user can choose to just provide high-level goals. At run-time the character has the flexibility to decide on how to best fill in any missing details and need not be overwhelmed by having to solve the whole problem on its own. Until more progress is made on planning from scratch our approach seems like a reasonable compromise between preprogrammed behavior and full run-time decision making.

Many examples of using complex actions to construct sketch plans for animated agents are given in (Funge 1998a). Numerous other examples, written in a closely related language called Golog (alGOL in LOGic), can be found at (Cognitive Robotics Group 1998). As a quick simple example of a very loose sketch plan, the following CML code can be used to find a path through a maze:

while (position != exit)
pick(d) *move*(d);

Just like a while-loop in a regular programming language, this while loop expands out into a sequence of statements. The difference is that in CML it can, potentially, expand out into all possible sequences of actions. In the case of a maze, this means that it will expand out into all possible paths through the maze. The integral runtime reasoning engine can then automatically search for a particular path through the maze. We call this style of programming nondeterministic. Nothing random is occurring, we just have the ability to cover multiple possibilities in one statement.

Sensing

In any reasonably complicated virtual world sensing is going to play a key role. This is especially so in computer games as the person playing the game interacts with the virtual world in a way that is highly unpredictable. The solution is to periodically sense aspects of the world and then re-plan. If we want the character itself to anticipate its need to intermittently re-plan then we need a way of representing the character's ignorance/knowledge about certain aspects of its world. Then when the character anticipates that it will no longer know something important it can plan to sense and re-plan.

Representing ignorance/knowledge within the situation calculus turns out to be a hard problem. Although there are some theoretical results, as far as we are aware, no previous approaches to sensing within the situation calculus are amenable to easy implementation. To address this problem we introduced the notion of interval-valued epistemic (IVE) fluents (Funge 1999) The idea is to use intervals to represent a character's level of ignorance/knowledge about aspects of its world. For example, if we have a fluent temp then the interval $\langle 10, 40 \rangle$ can be used to represent that the character

knows the temperature is between 10 and 40 degrees. Narrower intervals correspond to more knowledge and wider intervals to greater ignorance. We can write effect axioms to state how the intervals grow and shrink with sensing actions and with time. In this way a character can anticipate that after a certain time period, or event, it will be ignorant of some aspect of its world, and it can plan to sense and re-plan.

Case Studies

So far we have created three complete fully-fledged working examples, the details of which can be found in (Funge 1998a).

1. A camera controller (described below).
2. A territorial T-Rex that can herd some uncooperative and more agile Raptors toward a narrow passageway and out of its territory. To generate this behavior using CML all we had to tell the T-Rex were two things: First, that the Raptors are afraid of it and will run in the opposite direction if it gets to close; and second that every time it re-plans its goal is to get as many Raptors heading in the right direction as possible. From this simple knowledge base the T-Rex is able to behave like a good sheepdog and automatically plan motion paths that get "in and around" the Raptors to frighten them in the right direction. Moreover, it automatically chooses paths that avoid deflecting Raptors already heading in the right direction.
3. An intelligent merman that can use its wits to hide behind obstacles to avoid being eaten by a larger and faster shark. This example employs a sophisticated underwater physics simulation (see (Tu & Terzopoulos 1994)) and, unlike the previous two examples, does not currently run in real-time.

The last two examples were created in collaboration with Tu and Terzopoulos by using their low-level reactive behavior system to provide a higher level API to interface with CML . The camera example, however, did not require the reactive behavior system and so in this paper we shall use it as a complete working example of how CML is used in practice.

Cinematography

At first, it might seem strange to be advocating building a cognitive model for a camera. We soon realize, however, that it is the knowledge of the cameraperson, and the director, who control the camera that we want to capture with our cognitive model.

We shall begin by introducing the fluent Talking(A, B) that is true whenever character A is talking to character B. Domain knowledge can now be given to characters intuitively, in terms of actions, their preconditions and their effects. In [He96], the authors discuss one particular formula for filming two characters talking to one another. The idea is to flip between "external" shots of each character, focusing on the character doing the talking. In this paper our focus is on high-level camera control but for completeness we give some low-level rules that cover the geometry of camera placement. To make these low-level rules easy to follow we include simplified versions of those we used in our actual implementation.

We introduce an action *external* that takes two arguments, characters A and B, and places the camera so that A is seen over the shoulder of B. A precondition for this action states that we only want the camera pointing at A, if we are already filming A, and it has not got boring yet; or we not filming A, and A is talking, and we have stayed with the current shot long enough.

action *external*(A,B) **possible when** (!Boring && Filming(A)) || (Talking(A,B) && !Filming(A) && !TooFast;

Similarly, one effect of the external camera action is that the camera is looking at character A.

occurrence *external*(A,B) **results in** lookAt = p **when** scene(A(upperbody,centroid)) = p;

Another effect is that the camera is located above B's shoulder:

occurrence *external*(A,B) **results in** lookFrom = $p + k_2 *$ up $+k_3 *$ normalize$(p - c)$ **when** scene(B(shoulder,centroid)) = p && scene(A(upperbody,centroid)) = c;

where k_2 and k_3 are some suitable constants.

We refer the reader to (Funge 1998a) for the remainder of the domain specification (about 10 simple auxiliary definitions) . Now we want to write a controller for the camera to film a conversation. To break up the monotony, we want our camera to ensure that shots are interspersed with reaction shots of the other character. In (He, Cohen, & Salesin 1996), the formula is encoded as a finite state machine. Because we separate out the domain knowledge from tehcontrol information we can use complex actions to capture the same formula with far more elegance and brevity. In particular, assuming the background domain knowledge described above, the controller that will move the camera to look at the character doing the talking, with occasional respites to focus on the other character's reactions, is as follows.

```
setCount;
while (0 < silenceCount) {
    pick(A,B) external(A,B);
    tick;
}
```

Figure 1 shows some frames from an animation called "Cinemasauras". The animation shows how our approach has been applied to camera control. The action consists of a T-Rex and a Raptor having a "conversation".

Future Work

Real-time Decision Making

To enhance the applicability of our approach to computer games would entail adopting a more comprehensive strategy for real-time decision making. In (Funge & Tu 1997; Funge 1998b) CML was combined with the reactive behavior system described in (Tu & Terzopoulos 1994). The result was a system that tried to come up with an "intelligent" course of action, but could always fall back on a reactive system if that were not possible. In particular, a time constraint

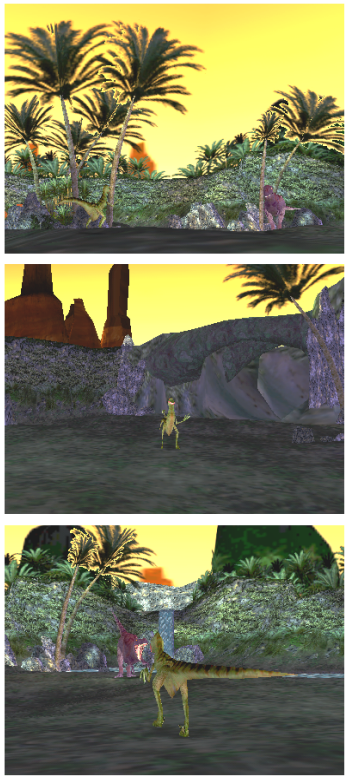

Figure 1: Cinematography.

can cause the reasoning system to fail. To introduce a hard real-time constraint we envisage moving to an architecture in which the reasoning engine acts as a server that runs as a separate process. Requests can be submitted to the server and if it responds in time the results can be incorporated into the characters behavior. While the character is waiting for a response, or if a response is not forthcoming a simple reactive system will take-over.

Learning

Learning is a large and complex field that could impact our work in numerous ways. For example, instead of having the user enter them manually, it should be possible to learn some of the domain knowledge action-effect axioms. Another possibility is to use the reasoning engine to generate a training set for learning a faster production-rule style controller. This could even take place "on-line" so that the character constantly gets faster and smarter as time goes on. The behavior capture work described in (van Lent & Laird 1998) and the neural network learning described in (Grzeszczuk, Terzopoulos, & Hinton 1998) give us reason to believe that something along the lines of what we propose is feasible.

Applications

Perhaps one of the most interesting and challenging possible future applications would be to create intelligent characters that can assist the game player. For example an intelligent wingman in a flight simulator game. Other challenges could be to create intelligent characters that do not try too hard to defeat their human adversaries, but instead try to "lose well".

Conclusion

In order to create compelling intelligent characters for computer games it is helpful to promote a firm understanding of some of the underlying knowledge representation issues that can arise. This understanding is likely to come through examples that relate to the kind of problems faced by game programmers in their own projects. Therefore we have developed a number of applications of CML to specifying the behavior of the kinds of quasi-intelligent characters that will populate the computer games of tomorrow. Some animations and selected frames from those animations are available at (Animations and Images 1998).

Acknowledgements

Thanks to Eugene Fiume for suggesting the application to cinematography, and to Angel Studios for the dinosaur API.

References

Animations and Images, 1998. `www.cs.toronto.edu/~funge`.

Funge, J., and Tu, X. 1997. Making them behave. In *SIGGRAPH 97 Visual Proceedings*. ACM SIGGRAPH.

Funge, J. 1998a. *Hardcore AI for Computer Games and Animation*. Siggraph 98, Orlando, FL: Course Notes #10.

Funge, J. 1998b. *Making Them Behave: Cognitive Models for Computer Animation*. Department of Computer Science, University of Toronto, Toronto, Canada: PhD thesis.

Funge, J. 1999. Representing knowledge within the situation calculus using interval-valued epitemic fluents. *Journal of Reliable Computing* 5(1).

The Cognitive Robotics Group, 1998. `www.cs.toronto.edu/~cogrobo`. Contains copies of numerous relevant papers.

Grzeszczuk, R.; Terzopoulos, D.; and Hinton, G. 1998. Neuroanimator: Fast neural network emulation and control of physics-based models. In Proceedings of SIGGRAPH '98 (Orlando, FL). ACM SIGGRAPH.

He, L.; Cohen, M. F.; and Salesin, D. 1996. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of SIGGRAPH '96 (New Orleans, LA)*. ACM SIGGRAPH.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31.

Reiter, R. 1991. The frame problem in the situation calculus. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press.

Tu, X., and Terzopoulos, D. 1994. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of SIGGRAPH '94 (Orlando, Florida)*. ACM SIGGRAPH.

van Lent, M., and Laird, J. 1998. Behavior capture: Motion is only skin deep. In *In Lifelike Computer Characters 98*.