A Command Language for Taskable Virtual Agents

Pat Langley, Nishant Trivedi, and Matt Banister
Computer Science and Engineering
Arizona State University
Tempe AZ 85287 USA

Abstract

In this paper, we report progress on making synthetic characters more taskable. In particular, we present an English-like command language that lets one specify complex behaviors an agent should carry out in a virtual environment. We also report compilers that translate English commands into a formal notation and formal statements into procedures for ICARUS, an agent architecture that supports reactive execution. To demonstrate the benefits of such taskability, we have integrated ICARUS with TWIG, which provides a simulated physical environment with humanoid agents. We use the command language to specify three complex activities, including responding to an object contingently, collecting and storing a set of objects, and negotiating with another agent in order to purchase an item. We also discuss related work on controlling synthetic characters, along with paths for additional research on taskability.

Introduction

Although many modern digital games include nonplayer characters that are controlled by computer software, there is wide agreement that richer characters would make such interactive systems more compelling and entertaining. To this end, there is a growing body of research on using techniques from artificial intelligence to support more sophisticated and interesting agents that operate in games and similar simulated environments. Some results from this work have already made the transition into commercial games, and this trend seems likely to continue.

Nevertheless, specifying the behaviors that agents should exhibit remains a tedious and painstaking process. Cognitive architectures like Soar (Laird et al., 1987) provide high-level constructs that reduce some of the effort, but creating knowledge in these frameworks is still time consuming. Authoring tools for synthetic characters (e.g., Perlin & Goldberg, 1996) also alleviate some issues, but again require considerable work before a character behaves in the desired ways. Knowledge-lean approaches that carry out heuristic search or use simple reactive controllers help in some contexts, but they do not produce the rich, structured behavior we associate with human intelligence.

One ability that these approaches and the resulting agents lack is something that humans do readily: accept and carry out complex instructions. Despite the increasing reliance of synthetic characters on domain knowledge that constrains and guides their behavior, they cannot interpret and follow commands. Langley et al. (2009) refer to this as *taskability* and note its absence as an important limitation of existing work on intelligent agents. More taskable agents would make compelling companion characters in digital games and other virtual environments.

In this paper, we propose a novel approach to enabling taskable systems. The central idea involves a command language that lets one specify complex behaviors the agent should carry out. Task statements are composable, so that one can produce complex commands by combining simpler ones. We couple the language with compilers that translate commands into structures that an agent architecture can execute in a virtual environment. Together, the formalism and the compilers support taskable agents that accept and follow complex instructions.

In the next section, we describe a command language that lets one specify instructions for complex tasks. After this, we review ICARUS, a cognitive architecture, and TWIG, a procedural animation system, that we have integrated to demonstrate the formalism's operation. Our approach is not limited to either system, but they provide a context in which to illustrate our claims. Next we report the results of three ICARUS runs with taskable TWIG agents that exhibit complex behaviors in response to our commands. We conclude by discussing related work on guiding agent activities and suggesting directions for future research.

A Command Language for Complex Tasks

The main contribution of our research is a command language for taskable agents that follow instructions about how to interact with the environment and other agents. The language can specify complex behaviors by combining a small set of command predicates with domain concepts and actions. The framework lets one describe activities in terms of application conditions, termination criteria, and orderings on subactivities. Because elements are composable, one can create complex commands in terms of simpler ones and thus express a wide range of rich agent behaviors.

The command language uses a constrained English syntax to describe activities. For example, the statement

If you are a child and D is a doll,
Then Until you are at doll D
Holds you should move toward D.

demonstrates the use of two paired sets of command predicates. One pair, *If / Then*, specifies one or more conditions for carrying out a subcommand. The other, *Until / Holds*, indicates the halting criteria for a subcommand. The expressions *doll* and *at doll* refer to known conceptual predicates, while *move toward* refers to a known domain action. The words *is*, *a*, *are*, and *should* are 'stop' words included for readability, while the remaining terms, *D* and *you*, denote pattern-match variables.

The English syntax maps onto an analogous formal syntax that uses list structures. For instance, the corresponding formal command would be

```
(if (child ?you) (doll ?d)
   (until (at-doll ?you ?d)
          (*move-toward ?you ?d)))
```

As before, the *if* predicate denotes conditions for carrying out a subcommand, while *until* specifies conditions (in this case, only one) for terminating a subcommand. Differences include using parentheses as delimiters rather than pairs of command words, omitting stop words, marking variables with question marks, and marking actions with asterisks.

The formal notation also clarifies the embedded nature of commands, with the *until* clause occurring within the *if* statement and with the **move-toward* action occurring within the *until* clause. Both the *if* and *until* predicates take two or more arguments, with the final one being either a domain action or another command. Earlier arguments specify a set of conditions that must match consistently (with shared variables having the same values) for the statement to apply. For *if* statements, these indicate conditions for carrying out the final action or subcommand; for *until* statements, they specify the conditions for halting the subcommand.

Another function of commands is to specify subtask orderings. For example, consider the more complex statement

If D is a doll,
Then First Until you are at doll D,
Holds you should move toward D,
Next Until you are grasping D,
Holds you should grasp object D.

The use of *First* and *Next* here indicates that the agent should attempt to grasp the doll only after getting near that object. Such ordering constraints are common in natural language instructions, since it is often necessary to complete one subtask before starting another.

We can also specify the same ordering in our formal syntax; in this case, we would write

```
(if (doll ?d)
   (before (until (at-doll ?you ?d)
                  (*move-toward ?you ?d))
           (until (grasping ?you ?d)
                  (*grasp-object ?you ?d))))
```

Here the *before* predicate indicates that its first argument should be carried out before the second. Both arguments are subtasks in the same formalism, but they might also have been domain actions. In many ways, *before* is a simpler relation than *if* and *until*, since it involves no conditions, and thus adds no domain concepts or pattern-match variables.

However, these do not exhaust the types of commands we need to support. We must also express contingent courses of action. As an example, consider the command

If D is a doll and T is a tree,
Either If you are nearer to D than T,
Then you should move toward D,
Or If you are nearer to T than D,
Then you should move toward T.

Here the *Either* predicate indicates the start of the first alternative, while *Or* marks its end and the start of the second option. Although not shown in this example, the syntax allows three or more alternatives, with *Or* initiating each one.

Again, we can state equivalent instructions in the formal command syntax, with the English statement translating to

```
(if (doll ?d) (tree ?t)
   (or (if (nearer ?you ?d ?t)
           (*move-toward ?you ?d))
       (if (nearer ?you ?t ?d)
           (*move-toward ?you ?t))))
```

In this case, the *or* predicate has two arguments, but it can take three or more, each of which must be a subcommand or an expression with a domain action like **move-toward*.

The above statements are interpreted imperatively, as indication that the agent should carry out the commands. However, both the English and the formal languages also include a nonimperative *define* predicate that lets one encapsulate procedures for later reuse. This predicate takes two arguments, the activity's name and arguments followed by its specification. The ability to define named procedures has two important advantages. One is that it enables specification of recursive activities that could not be stated otherwise. The other is that it lets one define generalized behaviors, such as obtaining and delivering an arbitrary object, then repeatedly give this command imperatively with different entities as arguments. This capability greatly reduces the effort needed to produce complex behavior.

Of course, the command language itself would be useless without some means to convert statements into executable procedures. This equates to translating command expressions into the format used by some agent architecture. We have developed two such compilers; we will not report their details here, except to note that both are written in Common Lisp. The first translates commands in the constrained English syntax into list structures of the formal command notation. The second compiles commands in the formal syntax into procedures for an agent architecture that we review in the next section. Once the formal notation compiler has processed a nondefinitional command, it treats the statement as an imperative, calling on the architecture to execute the compiled version. Upon completion, it awaits further instructions and executes them upon arrival.

Some readers may question whether the ability to state instructions in constrained English makes agents any more taskable than simply writing programs in a traditional procedural language. However, note that although our command language can specify desired behavior in great detail, one can also give very abstract instructions that refer to known procedures. Commands may also be nondeterministic and rely on the agent architecture to make choices among alternative expansions. Taken together, these features support strong taskability in the sense we described earlier.

Review of the ICARUS Architecture

For a synthetic character to take advantage of command statements, it must be able to represent the meanings of those commands and have some way to interpret them. The research community has explored multiple paradigms for producing agent behavior in simulated environments. We favor using a cognitive architecture (Langley et al., 2009) that makes strong commitments to the representations, performance mechanisms, and learning processes that underlie intelligent behavior. Our chosen architecture is ICARUS, which has controlled synthetic characters in a number of virtual environments (Choi et al., 2007; Li et al., 2009). Although we believe our approach to taskability generalizes beyond this framework, our demonstrations to date depend on it, so we should review it briefly.

ICARUS shares important features with other cognitive architectures like Soar (Laird et al., 1987), such as using symbolic list structures, matching long-term knowledge against short-term elements in a recognize-act cycle, and combining goal-driven processing with stimulus-driven behavior. The most basic process in ICARUS is conceptual inference, which provides an agent with information about its situation in the environment. On each cognitive cycle, the environment deposits descriptions of perceived objects into a perceptual buffer, against which ICARUS matches conceptual rules to produce inferences that it deposits in a belief memory. The inference process uses lower-level beliefs to produce higher-level ones, generating a set of literals that comprise the agent's understanding of its current situation. ICARUS repeats the process on each cycle, adding new beliefs and eliminating ones that are no longer justified.

Whereas conceptual inference produces beliefs about the world, a second ICARUS process – skill execution – attempts to alter the environment. This mechanism relies on a memory for procedures or 'skills' that describe how to achieve goals. Each skill specifies a name and arguments, a set of preconditions, a set of expected effects, and a set of ordered subskills or executable actions. Execution begins with a top-level intention and, on each cycle, takes one step downward through the skill hierarchy, selecting a subskill with conditions satisfied by current beliefs. This continues until the system reaches a primitive skill, in which case it executes the associated actions. ICARUS may execute the same intention on successive cycles or, upon achieving desired effects, shift to later subskills. This process continues until it completes the top-level intention or finds no relevant subskills are applicable. The mixture of bottom-up inference with top-down execution makes ICARUS a teleoreactive architecture.

Given the first example command in the previous section, the two compilers would produce the single ICARUS skill

```
((skill-1 ?you ?d)
 :conditions  ((child ?you) (doll ?d))
 :actions     ((*move-toward ?you ?d))
 :effects     ((at-doll ?you ?d)))
```

which incorporates both the initiation conditions and the halting criterion in a single structure. In contrast, the second example command would produce three ICARUS skills

```
((skill-4 ?you ?d)
 :conditions  ((doll ?d))
 :subskills   ((skill-2 ?you ?d) (skill-3 ?you ?d)))
((skill-2 ?you ?d)
 :actions     ((*move-toward ?you ?d))
 :effects     ((at-doll ?you ?d)))
((skill-3 ?you ?d)
 :actions     ((*grasp-object ?you ?d))
 :effects     ((grasping ?you ?d)))
```

that are organized in a two-level hierarchy, with the first, *skill-4*, referring to *skill-2* and *skill-3* in that order.

The TWIG Environment

We also require an environment in which to develop and demonstrate our ideas on taskability. For this purpose, we selected TWIG (Horswill, 2008), a low-fidelity simulator that supports the creation, rendering, and animation for a basic set of virtual characters and objects. In essence, it provides a stage, a set of humanoid actors, and a set of passive props with which they interact. Although relatively simple compared to many simulation environments, TWIG has the advantage of providing a reasonably broad range of functionality in a reasonably small and fast implementation.

The current TWIG implementation supports child and adult actors, which differ mainly in their size, along with inanimate balls, dolls, trees, and chairs. A child or adult can approach any object, including another agent; they can also pick up and drop dolls, sit on and rise from chairs, and kick balls. An agent can interact directly with another nearby actor by hugging it or fighting with it. TWIG also lets agents communicate in text that it displays in speech bubbles. The environment comes with low-level reactive behaviors that support basic activities like moving to an object.[1]

Before we could demonstrate our approach to taskability, we had to integrate ICARUS and TWIG in ways that supported their interaction. Because TWIG is implemented in C# and ICARUS is written in Lisp, we used message passing over TCP/IP streams to let them communicate. ICARUS is responsible for high-level inference and goal-directed activity, while TWIG handles low-level reactive behavior. Because ICARUS uses percepts to drive conceptual inference, TWIG provides a description of the environment on each cycle. Whenever ICARUS is ready to execute an action associated with a primitive skill, it tells TWIG to initiate the appro-

[1] TWIG itself provides basic taskability in the form of scripts, but these do not support the complex, extended activities we believe are necessary to mimic intelligence in synthetic characters.

priate reactive behavior. If TWIG completes this behavior successfully, then ICARUS receives the updated state on the next cycle and turns to another skill.

We also wanted to model social interaction, including constrained forms of dialogue, between ICARUS-controlled TWIG agents. The basic environment handles physical interaction, but we needed some way for agents to exchange messages about topics like buying objects. We adapted the existing TWIG messaging mechanism to this end. The initiating ICARUS agent sends a directive that TWIG should address a message to the recipient with details about speaker, listener, and content. TWIG passes the information to all agents as percepts that contain relevant information. For instance, the percept for an offer to buy or sell an object includes ids for the offer, object, buyer, and seller, along with a price and deadline. The percept is available until this deadline passes.

Experimental Demonstrations

In order to demonstrate that our command language supports taskability, we used it to specify three tasks for ICARUS agents operating within the TWIG environment. In this section, we describe each command's intent, show its English and formal specifications, and discuss the behavior that the compiled skills produced. In our experience, the command language makes it substantially easier to describe complex behaviors than writing ICARUS programs directly.

Scenario 1: Contingent Dolls

The initial scenario involves a combination of contingent, conditional, and ordered activities. We instructed the agent to obtain a nearby doll if it was not currently holding one, but to instead take any doll that it is holding to some tree. The English specification of this command was

If D is a doll,
Then Either If not you are grasping D,
Then First Until you are at doll D
Holds you should move toward D,
Next If you are at doll D,
Then Until you are grasping D
Holds you should grasp object D,
Or If you are grasping D and T is a tree,
Then First Until you are at tree T
Holds you should move toward T
Next you should drop object D.

Using knowledge about command predicates, concepts and actions, and stop words, the compiler translates this statement into an equivalent structure in the formal notation,

```
(if (doll ?d)
   (or (if (not (grasping ?you ?d))
          (before (until (at-doll ?you ?d)
                         (*move-toward ?you ?d))
                  (if (at-doll ?you ?d)
                     (until (grasping ?you ?d)
                            (*grasp-object ?you ?d)))))
       (if (grasping ?you ?d) (tree ?t)
          (before (until (at-tree ?you ?t)
                         (*move-toward ?you ?t))
                  (*drop-object ?you ?d)))))
```

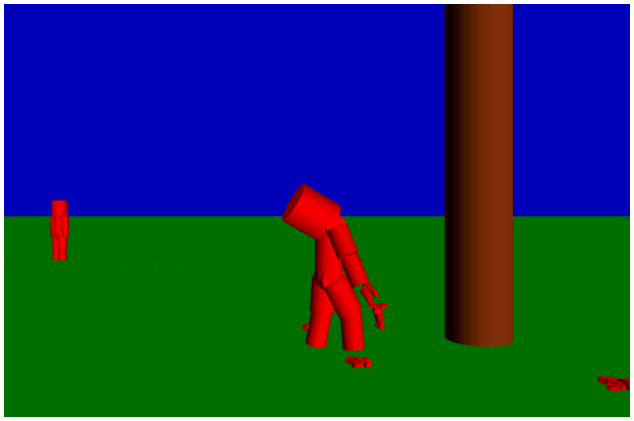

Figure 1: The doll-collecting agent depositing one of its finds near the tree.

The second compiler in turn translates this command into six ICARUS skills, along with an imperative top-level intention, that produce the desired behavior. If the agent is not holding the specified doll, it moves toward it, halts on getting close enough, and picks it up. However, if the agent is already holding a doll when the task starts, it moves toward a nearby tree and, upon reaching it, drops the object.

Scenario 2: Collectable Dolls

Our second scenario is more complex in that it involves an agent interacting with five dolls and a tree. We instructed the agent to approach any doll it observes, pick it up, carry the doll to the tree, and drop it there, continuing this process until all dolls are at the tree. Because this was a recursive procedure, we used *define* to name the top-level task as *put all dolls in stash* with two English commands

Define you put all dolls in stash T as
If T is a tree and D is a doll not in stash T,
Then Until all dolls in stash T
Holds First you take doll D to T,
Next you put all dolls in stash T.

Define you put all dolls in stash T as
If T is a tree,
Then Until D is a doll in stash T
Holds you take doll D to T.

The first compiler translates these statements into two analogous formal commands, specifically

```
(define (put-all-dolls-in-stash ?you ?t)
   (if (tree ?t) (doll-not-in-stash ?d ?t)
      (until (all-dolls-in-stash ?t)
             (before (take-doll ?t ?you ?d)
                     (put-all-dolls-in-stash ?you ?t)))))
(define (put-all-dolls-in-stash ?you ?t)
   (if (tree ?t)
      (until (doll-in-stash ?d ?t)
             (take-doll ?t ?you ?d))))
```

We also used *define* to specify a subtask, *take doll*, that occurs in both the recursive and base cases of the procedure.

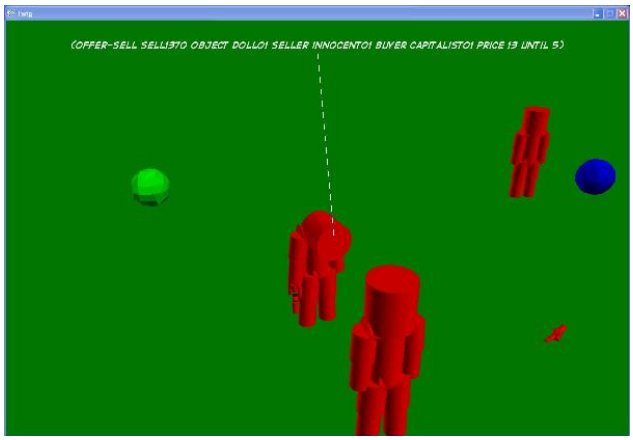

Figure 2: Counteroffer by the innocent to the capitalist in the doll purchase scenario.

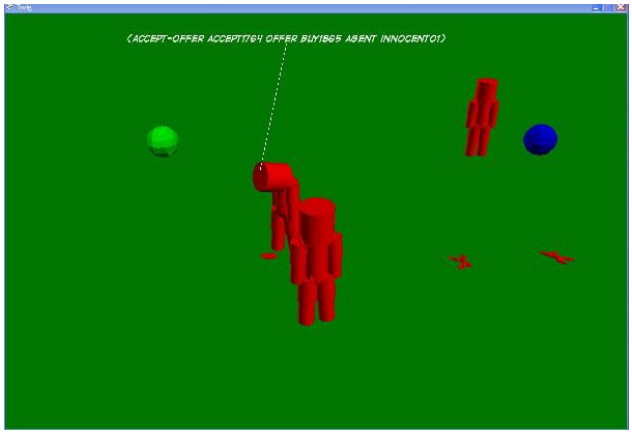

Figure 3: Final agreement between the capitalist and innocent in the doll purchase scenario.

The English and formal versions of this command were

Define you take doll D to T as
First you move toward D, Next you pick up D,
Next you move toward T, Next you drop object D.

and

(define (take-doll ?you ?d ?t)
*(before (*move-toward ?you ?d) (*grasp-object ?you ?d)*
*(*move-toward ?you ?t) (*drop-object ?you ?d)))*

Overall, the two compilers generated three ICARUS skills that produce the intended behavior when given the imperative command *You put all dolls in stash tree1*. In response, the agent first walks to one of the dolls, then grasps it, walks to the tree, and drops the object. After this, it repeats the procedure with a second doll and continues the process until it has transferred all of them to their new home. Figure 1 shows a screenshot near the end of this activity.

Scenario 3: Capitalism in Action

Our final scenario is even more complex, in that it involves economic interaction among two agents, each following its own instructions. This TWIG environment includes two distinct types of agents. Capitalists want to amass wealth, so they seek to buy dolls at the lowest price possible and sell them at a higher rate. Innocents are naive agents that pick up dolls and, if approached by a capitalist making an offer, negotiate about the price. The capitalist initiates negotiations, but the innocent can make counteroffers.

In this case, the command statements specified the negotiation protocol that each agent should follow. We tasked a capitalist to approach an innocent if she has a doll and offer to buy it. If the innocent agrees on the price, she transfers the doll in exchange for the money; if not, she makes a counteroffer that reduces the selling price. If the capitalist finds this low enough, he agrees and the transfer occurs; if not, he makes a counteroffer that raises the amount. This bargaining continues until they agree upon a price or either agent reaches its threshold, in which case their interaction ends.

We lack space to present the formal specification for this task, but the command statement was approximately 30 lines for the capitalist and 20 for the innocent. These compiled into eight ICARUS skills for the former and seven skills for the latter. The agents behave as desired when operating with the compiled instructions. The capitalist approaches a doll-holding innocent and makes an initial offer. Since this is below its threshold, the innocent makes a counteroffer with a higher price, as shown in Figure 2. The capitalist finds this too high, so he makes his own counteroffer. This bargaining continues for two rounds, at which point the agents agree on a price and exchange the doll, as Figure 3 illustrates.

Related Research

There has been considerable research on endowing synthetic characters, in digital games and elsewhere, with sophisticated abilities, but relatively few efforts on making these agents taskable. André et al. (1998) reported a method for creating life-like agents by writing directives in a script-like formalism which was then compiled into executable machine code. Vosinakis and Panayiotopoulos (2003) described an approach closer to our own, proposing a language that specifies agent behavior as hierarchical combinations of sequential, parallel, and conditional actions. However, neither utilized an agent architecture to interpret compiled behaviors, and neither supported an English-like syntax.

In another early effort, Perlin and Goldberg's (1996) Improv aimed to let authors create layered, non-repetitive motions for synthetic characters, along with smooth transitions between them. The system included a behavior engine that interpreted groups of simple action sequences about how agents make decisions, communicate, and change. Blumberg et al.'s (1995) framework let users specify the agent behavior at an abstract motivational level that indicates generic goals, an intermediate task level that specifies procedures for achieving goals, and a detailed level that controls agent actions directly. Mateas and Stern's (2002) ABL provided a formalism not only for specifying complex behavior of individual agents, but also the joint activities needed for interactive drama. Our command language lacks some of these features, but it supports activities of equal or greater complexity, and its emphasis on taskability makes it distinctive.

Other researchers have developed abstract formalisms like Herbal (Cohen et al., 2005) and HLSR (Jones et al., 2006) to ease the process of cognitive modeling. These take the form of high-level programming languages that generalize common features of architectures like ACT-R and Soar. One can describe agent behaviors in an architecture-independent syntax, which is then compiled into structures in a selected architecture. Our approach takes this abstraction even higher by introducing an English-like command language that compiles into an intermediate formalism similar to HLSR and Herbal. This should make our approach accessible even to those with limited programming expertise.

Directions for Future Work

Although our experience to date suggests that our command language supports more taskable agents, it also seems clear that it would benefit from extensions. For instance, the current formalism focuses almost entirely on what actions the agent should carry out. However, we might also desire instructions that specify constraints on the environmental state, such as *do not come within ten feet of the tree when you are grasping a doll.* We should augment the language to handle such statements and revise the compiler to translate them into conditionalized constraints on agent behavior. We should also add abilities for specifying goals the agent should achieve and for defining the domain concepts that appear in commands. Finally, we should support the parallel, coordinated actions that arise in some complex activities.

Naturally, we should also demonstrate the framework's generality by using it to generate a wider variety of ICARUS behaviors in the TWIG environment. We should also devote some energy toward adapting it to other virtual environments in which agents must accept and execute commands. One likely candidate involves search-and-rescue scenarios for human-robot interaction, where a human commander repeatedly gives commands to a robot that is looking for survivors. In addition, the framework's ability to state abstract, imperative commands suggests its use for directing agent behavior in interactive drama. Finally, we have argued that our command language will prove useful for other agent architectures, but this means writing compilers from the formalism to their internal structures, which we will leave to others.

Closing Remarks

In the preceding pages, we posed the challenge of creating taskable synthetic agents that accept and carry out complex instructions. In response, we described a command language with a clear syntax that embeds simpler tasks within more complicated ones. We also reviewed ICARUS, a cognitive architecture, and TWIG, an animation framework, which together let us show that our formalism produces intended behavior in synthetic characters. We demonstrated this taskability on three scenarios that varied in complexity.

We examined the literature on high-level formalisms for specifying agent behavior, finding that our approach is not entirely novel but that it has unique features. We also discussed limitations of our command language and ways to improve it in future work. Developing a framework for specifying complex behaviors is not the only requirement for believable synthetic characters, but it may well be an essential component, and we encourage other researchers to join us in addressing this important problem.

Acknowledgements

This work was funded in part by ONR Grant N00014-07-1-1049. We thank Ian Horswill for assistance with the TWIG environment and Glenn Iba for his contributions to ICARUS.

References

André, E., Rist, T., & Müller, J. (1998). Integrating reactive and scripted behaviors in a life-like presentation agent. *Proceedings of the Second International Conference on Autonomous Agents* (pp. 261–268). Minneapolis: ACM.

Blumberg, B., & Galyean, T. (1995). Multilevel direction of autonomous creatures for real-time virtual environments. *Proceedings of the 22nd Annual Conference on Computer Graphics* (pp. 47–54). Los Angeles: ACM Press.

Choi, D., Könik, T., Nejati, N., Park, C., & Langley, P. (2007). A believable agent for first-person shooter games. *Proceedings of the Third Annual Artificial Intelligence and Interactive Digital Entertainment Conference* (pp. 71–73). Stanford, CA: AAAI Press.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). Herbal: A high-level language and development environment for developing cognitive models in Soar. *Proceedings of the Fourteenth Conference on Behavior Representation in Modeling and Simulation* (pp. 133-140).

Jones, R. M., Crossman, J. A. L., Libiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. *Proceedings of the Seveneth International Conference on Cognitive Modeling*. Mahwah, NJ: Lawrence Erlbaum.

Horswill, I. (2008). Lightweight procedural animation with believable physical interaction. *Proceedings of the Fourth Conference on Artificial Intelligence and Interactive Digital Entertainment*. Stanford CA: AAAI Press.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Langley, P., Laird, J. E., & Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, *10*, 141–160.

Li, N., Stracuzzi, D. J., Cleveland, G., Konik, T., Shapiro, D., Molineaux, M., & Aha, D. W. (2009). Constructing game agents from video of human behavior. *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Mateas, M., & Stern, A. (2002). A behavior language for story-based believable agents. *Papers from the Spring Symposium on Artificial Intelligence and Interactive Entertainment*. Stanford, CA: AAAI Press.

Perlin, K., & Goldberg, A. (1996). Improv: A system for scripting interactive actors in virtual worlds. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (pp. 205–216). ACM.

Vosinakis, S., & Panayiotopoulos, T. (2003). A task definition language for virtual agents. *Journal of WSCG*, *11*, 512–519.