2005-04-20

Improving and Extending Behavioral Animation Through Machine Learning

Jonathan J. Dinerstein
Brigham Young University - Provo

IMPROVING AND EXTENDING BEHAVIORAL ANIMATION THROUGH MACHINE LEARNING

by

Jonathan Dinerstein

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

April 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Jonathan Dinerstein

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Parris K. Egbert, Chair
Date	Bryan Morse
Date	Dan Ventura
Date	Michael Goodrich
Date	Kevin Seppi

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Jonathan Dinerstein in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Parris K. Egbert
Chair, Graduate Committee

Accepted for the Department

David Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce
Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

IMPROVING AND EXTENDING BEHAVIORAL ANIMATION
THROUGH MACHINE LEARNING

Jonathan Dinerstein
Computer Science
Doctor of Philosophy

Behavioral animation has become popular for creating virtual characters that are autonomous agents and thus self-animating. This is useful for lessening the workload of human animators, populating virtual environments with interactive agents, etc. Unfortunately, current behavioral animation techniques suffer from three key problems: (1) deliberative behavioral models (i.e., *cognitive models*) are slow to execute; (2) interactive virtual characters cannot adapt online due to interaction with a human user; (3) programming of behavioral models is a difficult and time-intensive process.

This dissertation presents a collection of papers that seek to overcome each of these problems. Specifically, these issues are alleviated through novel machine learning schemes. Problem 1 is addressed by using fast regression techniques to quickly approximate a cognitive model. Problem 2 is addressed by a novel multi-level technique composed of custom machine learning methods to gather salient knowledge with which to guide decision making. Finally, Problem 3 is addressed through programming-by-demonstration, allowing a non-technical user to quickly and intuitively specify agent behavior.

ACKNOWLEDGMENTS

Thanks to my advisor Dr. Parris K. Egbert, who always had time to answer my questions. The freedom he allowed me in selecting a research topic has proven critical to my success and education.

Thanks also to Dr. Dan Ventura and Dr. Michael Goodrich, who helped guide the machine learning and artificial intelligence aspects of my research. Their feedback on my hypotheses and writing style have been invaluable.

I am grateful to Dr. Bryan Morse and Dr. Kevin Seppi for their support as committee members. Their time spent on my behalf is greatly appreciated

Many thanks to my family, immediate and extended, who have been supportive during my education. Their faith in me will never be forgotten.

I am especially grateful to my wife, Bethanne, who pretended to listen whenever I wanted to tell her about my research ideas. Her support has made the achievement of this degree possible.

Contents

Part I

Introduction

Part I provides the background information and motivation for the remainder of the dissertation. It contains a single chapter.

Chapter 1 provides an introduction to the research contained in this dissertation. It introduces synthetic agents, machine learning for agents, and behavioral animation. Chapter 1 also discusses three key weaknesses of behavioral animation, and briefly outlines the solutions to these problems that are presented in this dissertation.

Chapter 1

Introduction and Motivation

Simulation of real-world environments is necessary for many applications of computers today. These applications include training simulators, computer games, special effects for film, immersive virtual environments, etc. Accurate simulation of the intended environment is critical if such an application is to be effective and fruitful.

The real world is full of autonomous biological creatures that are proactive, that are goal-fulfilling, and that may interact with one another. These creatures not only include humans but also birds, dogs, fish, insects, etc. To adequately and correctly simulate a specific environment (whether designed by human imagination or patterned after the real world), a simulation must involve those creatures that are expected to exist in that environment.

Unfortunately, it is often implausible for a human designer to explicitly dictate the behavior of a virtual creature. This results from the fact that the simulated environment may evolve in unexpected or diverse ways. Thus it is often necessary to create autonomous virtual creatures that are capable of automatically and intelligently responding to events in their environment. Through this approach, virtual worlds can be populated with intelligent and compelling characters. This sort of agent simulation might also be a useful tool for rapid prototyping of agents that will later be physically constructed (e.g., robots).

A popular approach to the creation of synthetic autonomous agents that inhabit virtual worlds is *behavioral animation* [Reynolds 1987]. A character is given perception, decision making, and motor control skills. The character responds to events in its environment through motion (or another external activity), thereby changing the state of its

environment. There are two primary character decision making schemes: *reactive* (simply responding to the current state) and *cognitive* (deliberating over the set of candidate choices by predicting the outcome of each). Behavioral animation overlaps with several important fields, including artificial intelligence, multi-agent systems, machine learning, computer graphics, and computer-human interfaces.

The objective of this dissertation is to enhance behavioral animation through machine learning. Specifically, techniques are presented that address the following limitations in current methods:

1. Cognitive models are very slow to execute, distinctly limiting their usefulness.

2. Interactive virtual characters cannot adapt on-line due to interaction with a human user — their behavior is static.

3. There are no simplified techniques for creating behavioral models. Traditionally, a model must be explicitly designed and programmed by a skilled developer.

Solving these three problems will allow for faster creation of more effective and useful autonomous virtual characters. Before giving the details about our solutions, we first give a brief overview of relevant background and related work.

1.1 Autonomous Agents and Machine Learning

An *agent* [Stone and Veloso 1997; Weiss 1999] is an entity that is capable of sensing its environment, making choices, and then performing actions that carry out those choices. Thus an agent can cause changes in the state of its environment. Real-world examples of agents include humans and animals.

A *synthetic agent* is an agent created by humans. The most common form today is software agents, though robotic agents are gaining in popularity.

A number of theories for the design and programming of synthetic agents have been proposed [Brooks 1986; Newell 1990; Rao and Georgeff 1995]. Many aspects of these theories are based on our current understanding of cognitive science [Matthews 1997; Nadel 2003]. However, the development of effective agent AI has remained difficult and costly.

For this reason, it has been widely proposed that machine learning may be a more effective approach than explicit designing of agent behavior (e.g., [Stone 2000]).

A machine learning approach to creating agent AI is compelling because it is hypothesized that humans gain nearly all skills through learning [Meltzoff and Moore 1992; Byrne and Russon 1998]. Thus it seems plausible that effective synthetic agent behavior can be learned (semi)automatically. Many agent-oriented machine learning techniques have been proposed, including: Q-learning [Watkins and Dayan 1992], minimax-Q [Littman 1994], agent/user modeling [Bui et al. 1996; Gmytrasiewicz and Durfee 2000], and emotion-guided reinforcement learning [Gadanho 2003]. One reason there are so many learning methods is because, as proven in [Schaffer 1994], no given approach is best for all problems.

Unfortunately, none of these learning techniques explicitly address the problem domain of interest in this dissertation: embodied agents that inhabit virtual worlds. This problem domain has unique constraints and requirements. Thus there is need for custom learning methods, tailored for autonomous virtual characters.

1.2 Behavioral Animation

Most synthetic agents that exist today are software agents. This results from the fact that software agents are far less expensive to develop and deploy than physical agents (e.g., autonomous robots). Moreover, there currently exists a large demand for software agents.

One popular and well-known category of software agent is *autonomous virtual characters*. These are synthetic agents that live in virtual worlds and have bodies that are displayed through computer graphics. These characters appear in computer-simulated or computer-generated environments such as training simulators, computer games, special effects for film, etc. These characters may interact with a human user and/or other autonomous characters.

A number of behavioral animation systems have been developed (see for example [Tu and Terzopoulos 1994; Blumberg and Galyean 1995; Perlin and Goldberg 1996; Isla et al. 2001; Monzani et al. 2001]). These techniques have produced impressive results,

but are limited in three aspects. First, they only perform reactive decision making, not deliberative (i.e., *cognitive*) decision making. Second, they have no ability to learn, and therefore are limited to pre-specified behavior. Third, the behavioral model for a character must be designed and implemented explicitly, which has proven challenging, especially for complex behavior.

Cognitive modeling [Funge et al. 1999] was recently introduced to provide virtual characters with deliberative decision making. This is performed through a tree search, constructing a plan (i.e., sequence of actions) that most fully achieves the character's current goal. Relatively little work has been performed thus far in cognitive modeling as compared to behavioral modeling. It has been shown that cognitive modeling for goal-based computer animation can result in astonishingly realistic and rich animations [Funge et al. 1999]. However, while effective, the tree search is very slow. As a result, only a few intelligent virtual characters can be used simultaneously. Further, only short, sub-optimal plans can be formulated, and only a small set of candidate actions can be considered if the tree search is to be performed quickly enough for real-time animation. These limitations greatly reduce the potential applications of this technique. As a result, cognitive modeling has seen little use in practice but has remained popular from a theoretical viewpoint.

Learning has only begun to be explored in behavioral animation ([Yoon et al. 2000; Evans 2002; Tomlinson and Blumberg 2002]). A notable example of behavioral learning is given in [Blumberg et al. 2002], where a technique is presented by which a virtual dog can be interactively taught by the user to exhibit desired behavior. This technique was inspired by dog training techniques and is designed around a master-slave relationship paradigm. It is related to reinforcement learning, and uses immediate explicit feedback from the human user. This technique has been shown to work extremely well. However, it has no support for high-level reasoning to accomplish complex tasks (i.e., it is reactive), and is not a good fit when the virtual character is an opponent or peer of the human user. Also, it cannot learn in the absence of a human user providing explicit feedback.

As listed in Section 1, there are three primary outstanding problems in the field of

behavioral animation: (1) cognitive models are prohibitively slow for interactive environments; (2) characters lack the ability to adapt; and (3) designing and programming behavioral/cognitive models is a technical, time-consuming endeavor. This dissertation presents solutions to these problems. These solutions utilize novel machine learning schemes that are tailored for the requirements and constraints of autonomous virtual characters.

1.3 Thesis Statement

The performance issue (Problem 1) is solved through the use of regression to quickly approximate complex cognitive models, thus making the process of animating intelligent virtual characters less CPU intensive. The adaptation issue (Problem 2) is solved by the use of our fast multi-level online learning system, thus allowing interactive characters to better cooperate with or compete against a unique human user. Finally, the modeling issue (Problem 3) is solved by the use of our learning-by-observation or programming-by-demonstration technique whereby an animator need only act out the desired behavior of a character to construct a behavioral model.

1.4 Overview of Dissertation

The remainder of this dissertation (with the exception of the final chapter) consists of a collection of papers that have been either published or submitted for publication in journal or conference proceedings. Each chapter, therefore, has its own abstract, introduction, and conclusions. There is also a small amount of intentional overlap between these chapters, so that each is a self-contained paper. The references for these papers are listed in the following section of this introduction and also appear at the beginning of the chapter to which each applies.

Part II presents two chapters that introduce our solution for slow execution performance of cognitive models. Specifically, Chapter 2 presents a technique for rapid approximation of a cognitive model through regression (i.e., function approximation). Samples of the behavior of the cognitive model (in the form of *state* $\rightarrow$ *action* pairs) are collected and then generalized using an artificial neural network. Chapter 3 presents continuing work on

this topic, providing a more formal basis for our technique and suggesting that a k-nearest neighbor (k-nn) approach be used for regression. Approximating the cognitive model with k-nn is interesting because the system learns very quickly, is robust, and there are well-established methods for automatically performing feature selection.

Part III presents two chapters on interactive adaptation for autonomous virtual characters. Chapter 4 introduces a technique for incremental action prediction. Specifically, the character records observations of the behavior of the human user. A model is created from these observations. While learning is taking place, this model is used to predict the future behavior of the user, allowing the character to intelligently choose actions to perform. Chapter 5 extends this work, presenting a multi-level adaptation technique. Each layer is composed of a separate learning method. These learning methods influence (from low to high level) the character's action selection, task selection, and goal selection. An imitation method is also presented whereby the character can imitate novel behavior performed by the human user.

Part IV presents two chapters on simplified construction of behavioral models through programming by demonstration (i.e., learning by observation). Chapter 6 introduces a technique for learning policies from human example. This technique is related to existing programming-by-demonstration methods in the robotics and agents literature but is specifically applied to behavioral animation and includes a novel conflict elimination algorithm. Chapter 7 discusses how autonomous virtual character behavior can be specified and synthesized in a data-driven manner. Sequences of actions are automatically captured from human demonstration. This data is then used to synthesize novel behaviors by "cutting and pasting" disjoint segments of the demonstrated action sequences. This data-driven approach is interesting because it has been empirically shown to be very scalable, intuitive, and powerful.

Part V contains a single conclusion chapter for this dissertation. This final chapter also proposes possible directions for future work.

1.5 Publications

Chapters 2–7 are based on a collection of papers that have either been published or submitted for publication in refereed journals or conferences. Following is a list of references for these publications in the order in which they appear in this dissertation.

Part II. Fast Construction and Approximation of Cognitive Models Through Regression

Jonathan Dinerstein, Parris K. Egbert, Hugo de Garis, and Nelson Dinerstein. "Fast and learnable behavioral and cognitive modeling for virtual character animation". *Journal of Computer Animation and Virtual Worlds*, **15**(2):95–108, 2004. (Chapter 2).

Jonathan Dinerstein and Parris K. Egbert. "Improved behavioral animation through regression". In *Proceedings of Computer Animation and Social Agents*, pp. 231–238, 2004. (Chapter 3).

Part III. Online Adaptation for Interactive Characters

Jonathan Dinerstein, Dan Ventura, and Parris K. Egbert. "Incremental action prediction for interactive autonomous agents". *Computational Intelligence*, **21**(1):90–110, 2005. (Chapter 4).

Jonathan Dinerstein and Parris K. Egbert. "Fast multi-level adaptation for interactive autonomous characters". To appear in *ACM Transactions on Graphics*, 2005. (Chapter 5).

Part IV. Creating Behavior Through Demonstration

Jonathan Dinerstein, Trent Crow, and Parris K. Egbert. "Intelligence capture — Automatic behavioral animation from human example". Submitted to *Journal of Graphics Tools*, June 2004. (Chapter 6).

Jonathan Dinerstein, Parris K. Egbert, Dan Ventura, and Michael Goodrich. "Data-driven programming and control for autonomous virtual characters". Submitted to *SIGGRAPH*, January 2005. (Chapter 7).

Part II

Fast Construction and Approximation of Cognitive Models Through Regression

Part II addresses Problems #1 and #3 listed in Chapter 1: slow execution speed of cognitive models and difficulty of designing/programming behavioral and cognitive models.

Chapter 2 presents a technique for rapid approximation of a cognitive model through regression (i.e., function approximation). Samples of the behavior of the cognitive model (in the form of *state* $\rightarrow$ *action* pairs) are collected and then generalized using an artificial neural network. Chapter 2 was published in the *Journal of Computer Animation and Virtual Worlds* and can be referenced as follows:

> Jonathan Dinerstein, Parris K. Egbert, Hugo de Garis, and Nelson Dinerstein. "Fast and learnable behavioral and cognitive modeling for virtual character animation". *Journal of Computer Animation and Virtual Worlds*, **15**(2):95–108, 2004.

Chapter 3 presents continuing work on this topic, providing a more formal basis for our technique and suggesting that a k-nearest neighbor algorithm be used for regression. Approximating the cognitive model with k-nn is interesting because it learns very quickly, is robust, and there are well-established methods for automatically performing feature selection. Chapter 3 was published under the following reference:

> Jonathan Dinerstein and Parris K. Egbert. "Improved behavioral animation through regression". In *Proceedings of Computer Animation and Social Agents*, pp. 231–238, 2004.

Chapter 2

Fast and Learnable Behavioral and Cognitive Modeling for Virtual Character Animation

Journal of Computer Animation and Virtual Worlds, Vol. 15, No. 2, pp. 95–108, 2004.

Abstract: Behavioral and cognitive modeling for virtual characters is a promising field. It significantly reduces the workload on the animator, allowing characters to act autonomously in a believable fashion. It also makes interactivity between humans and virtual characters more practical than ever before. In this paper we present a novel technique where an artificial neural network is used to approximate a cognitive model. This allows us to execute the model much more quickly, making cognitively empowered characters more practical for interactive applications. Through this approach, we can animate several thousand intelligent characters in real-time on a PC. We also present a novel technique for how a virtual character, instead of using an explicit model supplied by the user, can automatically learn an unknown behavioral/cognitive model by itself through reinforcement learning. The ability to learn without an explicit model appears promising for helping behavioral and cognitive modeling become more broadly accepted and used in the computer graphics community, as it can further reduce the workload on the animator. Further, it provides solutions for problems that cannot easily be modeled explicitly.

Keywords: computer animation, synthetic characters, behavioral modeling, cognitive modeling, machine learning, reinforcement learning.

2.1 Introduction

Virtual characters are an important part of computer graphics. These characters have taken forms such as synthetic humans, animals, mythological creatures, and non-organic objects that exhibit life-like properties (walking lamps, etc). Their uses include entertainment, training, simulation, etc. As computing and rendering power continue to increase, virtual characters will only become more commonplace and important.

One of the fundamental challenges involved in using virtual characters is animating them. It can often be difficult and time consuming to explicitly define all aspects of the behavior and animation of a complex virtual character. Further, the desired behavior may be impossible to define ahead of time if the character's virtual world changes in unexpected or diverse ways. For these reasons, it is desirable to make virtual characters as autonomous and intelligent as possible while still maintaining animator control over their high-level goals. This can be accomplished with a *behavioral model*: an executable model defining how the character should react to stimuli from its environment. Alternatively, we can use a *cognitive model*: an executable model of the character's thought process. A behavioral model is reactive (i.e., seeks to fulfill immediate goals), whereas a cognitive model seeks to accomplish long-term goals through *planning*: a search for what actions should be performed in what order to reach a goal state. Thus a cognitive model is generally considered more powerful than a behavioral one, but can require significantly more processing power. As can be seen, behavioral and cognitive modeling have unique strengths and weaknesses, and each has proven to be very useful for virtual character animation.

However, despite the success of these techniques in certain domains, some important arguments have been brought against current behavioral and cognitive modeling systems for autonomous characters in computer graphics.

First, cognitive models are traditionally very slow to execute, as a tree search must be performed to formulate a plan. This speed bottleneck requires the character to make sub-optimal decisions and limits the number of virtual characters that can be used simultaneously in real-time. Also, since a search of all candidate actions throughout time is performed, it is necessary to use only a small set of candidate actions (which is not practical for all problems, especially those with continuous action spaces). Note that behavioral

models are currently more popular than cognitive models, partially because they are usually significantly faster to execute.

Second, for some problems, it can be very difficult and time consuming to construct explicit behavioral or cognitive models (this is known as the *curse of modeling* in the artificial intelligence field). For example, it is not uncommon for behavioral/cognitive models to require weeks to design and program. Therefore, it would be extremely beneficial to have virtual characters be able to automatically learn behavioral and cognitive models if possible, alleviating the animator of this task.

In this paper, we present two novel techniques. In the first technique, an artificial neural network is used to approximate a cognitive model. This allows us to execute our cognitive model much more quickly, making intelligent characters more practical for interactive applications. Through this approach, we can animate several thousand intelligent characters in real-time on a PC. Further, this approach allows us to use optimal plans rather than sub-optimal plans.

The second technique we introduce allows a virtual character to automatically learn an unknown behavioral or cognitive model through reinforcement learning. The ability to learn without an explicit model appears promising for helping behavioral and cognitive modeling become more broadly used in the computer graphics community, as this can further reduce the workload on the animator. Further, it provides solutions for problems that cannot easily be modeled explicitly.

In summary, this paper presents the following original contributions:

- A novel technique for fast execution of a cognitive model using neural network approximation.
- A novel technique for a virtual character to automatically learn an approximate behavioral or cognitive model by itself (we call this *off-line character learning*).

We present each of these techniques in turn. We begin by surveying related work. We then give a brief introduction to cognitive modeling (as it is less well known than behavioral modeling) and neural networks. Next we present our technique for using neural networks to rapidly approximate cognitive models. We then give a brief introduction to reinforcement

learning, and then present our technique for off-line character learning. Next we present our experience with several experimental applications and the lessons learned. Finally, we conclude with a summary and possible directions for future work.

2.2 Related Work

Previous computer graphics research in the area of autonomous virtual characters includes automatic generation of motion primitives [van de Panne and Fiume 1993; van de Panne et al. 1994; Sims 1994; Grzeszczuk and Terzopoulos 1995; Grzeszczuk et al. 1998; Hodgins and Pollard 1997; Gleicher 1998]. This is useful for reducing the work required by animators. More recently, Faloutsos et al. [2001] present a technique for learning the pre-conditions from which a given *specialist controller* can succeed at its task, thus allowing them to be combined into a general purpose motor system for physically based animated characters. Note that these approaches to motor learning focus on learning how to move to minimize a cost function (such as the energy used). Therefore, these techniques do not embody the virtual characters with any decision-making abilities. However, these techniques can be used in a complementary way with behavioral/cognitive modeling in a multi-level animation system. In other words, a behavioral/cognitive model makes a high-level decision for the character (e.g., "walk left"), which is then carried out by a lower-level animation system (e.g., skeletal animation).

A great deal of research has also been performed in control of animated autonomous characters [Reynolds 1987; Tu and Terzopoulos 1994; Blumberg and Galyean 1995; Perlin and Goldberg 1996]. These techniques have produced impressive results, but are limited in two aspects. First, they have no ability to learn, and therefore are limited to explicit pre-specified behavior. Secondly, they only perform behavioral control, not cognitive control (where *behavioral* means reactive decision making and *cognitive* means reasoning and planning to accomplish long-term tasks). On-line behavioral learning has only begun to be explored in computer graphics [Burke et al. 2001; Yoon et al. 2000; Tomlinson and Blumberg 2002]. A notable example is [Blumberg et al. 2002], where a virtual dog can be interactively taught by the user to exhibit desired behavior. This technique is based on

reinforcement learning and has been shown to work extremely well. However, it has no support for long-term reasoning to accomplish complex tasks. Also, since these learning techniques are all designed to be used on-line, they are (for the sake of interactive speed) limited in terms of how much they can learn.

To endow virtual characters with long-term reasoning, cognitive modeling for computer graphics was recently introduced [Funge et al. 1999]. Cognitive modeling can provide a virtual character with enough intelligence to automatically perform long-term, complex tasks in a believable manner.

The techniques we present in this paper build on the successes of traditional behavioral and cognitive modeling with the goal of alleviating two important weaknesses: performance of cognitive models, and time-consuming construction of explicit behavioral and cognitive models. We will first present our technique for speeding up cognitive model execution through approximation. We will briefly review cognitive modeling and neural networks, and then present our new technique.

2.3 Introduction to Cognitive Modeling

Cognitive modeling [Funge et al. 1999; Funge 1999; Terzopoulos 1999; Funge 2000] is closely related to behavioral modeling, but is less well known, so we now provide a brief introduction. A *cognitive model* defines what a character knows, how that knowledge is acquired, and how it can be used to plan actions. The traditional approach to cognitive modeling is a symbolic approach. It uses a type of first-order logic known as "the situation calculus," wherein the virtual world is seen as a sequence of situations, each of which is a "snapshot" of the state of the world.

The most important component of a cognitive model is planning. *Planning* is the task of formulating a sequence of actions that are expected to achieve a goal. Planning is performed through a tree search of all candidate actions throughout time (see Figure 2.1). However, it is usually cost prohibitive to plan all the way to the goal state. Therefore, any given plan is usually only a partial path to the goal state, with new partial plans formulated later on.

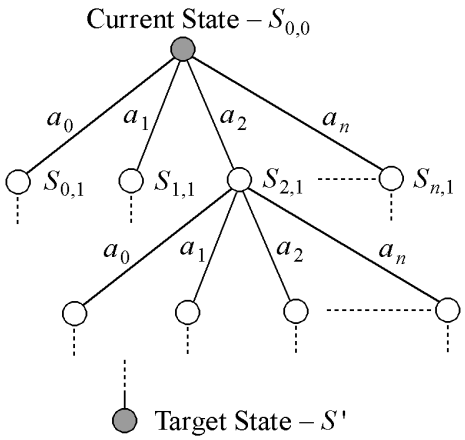

Figure 2.1: Planning is performed with a tree search of all candidate actions throughout time. To perform planning in real-time without dedicated hardware, it is usually necessary to greatly limit the number of candidate actions and to only formulate short (sub-optimal) plans.

The animator has high-level control over the virtual character since she can supply it with a goal state. Note that to achieve real-time performance, it is necessary to have the goal hard-coded into the cognitive model. This is because it is necessary to implement custom heuristics to speed up the tree search for planning (for further details see [Funge et al. 1999]). Therefore, either an animator and programmer must collaborate, or the programmer must also be the animator.

This traditional symbolic approach to cognitive modeling has many important strengths. It is explicit, has formal semantics, and is both human readable and executable. It also has a firm mathematical foundation and is well established in AI theory. However, it also has some significant weaknesses with respect to application in computer graphics animation. Since planning is performed through a tree search, and the branching factor is the number of actions to consider, the set of candidate actions must be kept very small if real-time performance is to be achieved. Also, to keep real-time performance, we are limited to short (sub-optimal) plans. Another performance problem that is unique to computer graphics is the fact that the user may want to have many intelligent virtual characters interacting in real-time. In most situations, on a commodity PC, this is impossible to achieve with the traditional symbolic approach to planning. Another limitation is that it is not possible to

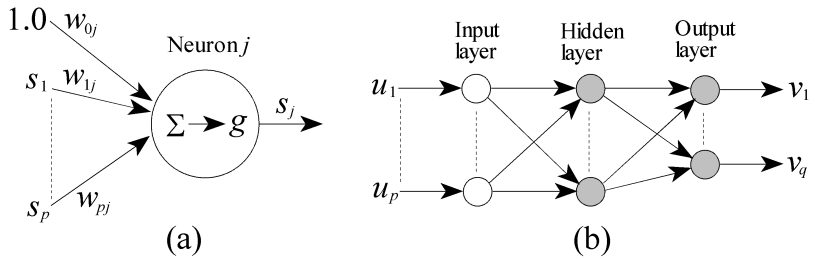

Figure 2.2: (a) Mathematical model of a neuron j. (b) A three-layer feedforward neural network of p inputs and q outputs.

have a virtual character automatically learn a cognitive model by itself (which could further reduce the workload on the animator, and provide solutions to very difficult problems).

2.4 Introduction to Artificial Neural Networks

Note that there are many machine learning techniques, many of which could be used to approximate an explicit cognitive model. However, we have chosen to use neural networks because they are both compact and computationally efficient. In this section we briefly review a common type of artificial neural network [Haykin 1999]. A more thorough introduction can be found in [Grzeszczuk et al. 1998]. There are many libraries and applications publicly available[1] (free and commercial) for constructing and executing artificial neural nets.

A *neuron* can be modeled as a mathematical operator that maps $\mathbb{R}^p \rightarrow \mathbb{R}$. Consider Figure 2.2a. Neuron j receives p input signals (denoted s_i). These signals are scaled by associated connection weights w_{ij}. The neuron sums its input signals

$$z_j = w_{0j} + \sum_{i=1}^{p} s_i w_{ij} = \mathbf{u} \cdot \mathbf{w}_j,$$

where $\mathbf{u} = [1, s_1, s_2, ..., s_p]$ is the input vector and $\mathbf{w}_j = [w_{0j}, w_{1j}, ..., w_{pj}]$ is the connection weight vector. The neuron outputs a signal $s_j = g(z_j)$, where g is an activation function:

$$s_j = g(z_j) = 1/(1+e^{-z_j}).$$

[1] e.g., SNNS (ftp.informatik.uni-tuebingen.de/pub/SNNS) and Xerion (ftp.cs.toronto.edu/pub/xerion).

A *feedforward* artificial neural network (see Figure 2.2b), also known simply as a neural net, is a set of interconnected neurons organized in layers. Layer l receives inputs only from the neurons of layer $l-1$. The first layer of neurons is the *input layer* and the last layer is the *output layer*. The intermediate layers are called *hidden layers*. Note that the input layer has no functionality, as its neurons are simply "containers" for the network inputs.

A neural network "learns" by adjusting its connection weights such that it can perform a desired computational task. This involves considering input-output examples of the desired functionality (or *target function*). The standard approach to training a neural net is the *backpropagation training algorithm* [Rumelhart et al. 1986]. Note that it has been proven that neural networks are universal function approximators (see [Hornik et al. 1989]).

An alternative approach that we considered was to use the *continuous k-nearest neighbor* algorithm [Mitchell 1997]. Unlike neural nets, the k-nearest neighbor algorithm provides a local approximation of the target function, and can be used automatically without the user carefully selecting inputs. Also, k-nearest neighbor is guaranteed to learn the target function to the quality of the examples that it has been provided (whereas no such guarantee exists with neural nets). However, k-nearest neighbor requires the explicit storage of many examples of the target function. Because of this storage issue, we opted to use a neural net approach.

2.5 Fast Animation using Neural Network Approximation of Cognitive Models

The novel technique we now present is analogous to how a human becomes an expert at a task. As an example, let's consider typing on a computer keyboard. When a person first learns how to type, she must search the keyboard with her eyes to find every key she wishes to press. However, after enough experience, she learns (i.e., memorizes) where the keys are. Thereafter, she can type more quickly, only having to recall where the keys are. There is a strong parallel between this example and all other tasks humans perform. After

enough experience we no longer have to implicitly "plan" or "search" for our actions; we simply recall what to do.

In our technique, we use a neural net to learn (i.e., memorize) the decisions made through planning by a cognitive model to achieve a goal. Thereafter, we can quickly recall these decisions by executing the trained neural net. Training is done off-line and then the trained network is used on-line. Thus, we can achieve intelligent virtual characters in real-time using very few CPU cycles.

We now present our technique in detail, first discussing the structure of our technique, followed by how to train the neural network, and then finally how to use the trained network in practice.

2.5.1 Structure

A cognitive model with a goal defines a *policy*. A policy specifies what action to perform for a given state. A policy is formulated as

$$a = \mu(i),$$

where i is the current state and a is the action to perform. This is a non-context-sensitive formulation, which covers most cognitive models. However, if desired, context information can also be supplied as input (e.g., the last n actions can be input). We train our feedforward neural net to approximate a specific policy μ. We denote the neural net approximation of the policy $\hat{\mu}$ (see Figure 2.3a). Note that the current state (network input) and action (output) will likely be vector-valued for non-trivial virtual worlds and characters. Further, a logical selection and organization of the input and output components can help make the target function as smooth as possible (and therefore easier to approximate). Selecting network inputs will be discussed in more detail later. Also note that the input should be normalized and the output denormalized for use. Specifically, the normalized input components should have zero means and unit variances, and the normalized output components should have 0.5 means and be in the range [0.1, 0.9]. This ensures that all inputs contribute equivalently, and that the output is in a range the neural net's activation function can produce.

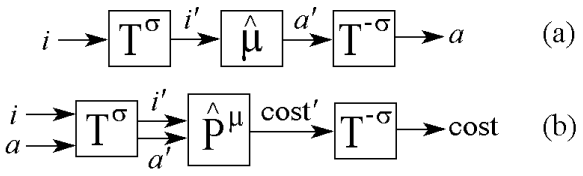

Figure 2.3: (a) Neural net approximation of a policy μ. The network input is the current state, the output is the action to perform. T^{σ} and $T^{-\sigma}$ normalize the input and denormalize the output, respectively. (b) Neural net approximation of a priority function.

An important question is how many hidden layers (and how many neurons in each of those hidden layers) we need to use in a neural net to achieve a good approximation of a policy. This is important because we want a reasonable approximation, but we also want the neural net to be as fast to execute as possible (i.e., there is a speed/quality tradeoff). We have found empirically that, at minimum, it is best to use one hidden layer with the same number of neurons as there are inputs. If a higher quality approximation is desired, it is useful to use two hidden layers, the first with $2p+1$ neurons (where p is the number of inputs), and the second with $2q+1$ neurons (where q is the number of outputs). We have found empirically that any more layers and/or neurons than this usually provides little benefit. Note that the state and action spaces can be continuous or discrete, as all processing in a neural network is real-valued. If discrete outputs are desired, the real-valued outputs of the network should simply be quantized to predefined discrete values.

Even though cognitive models (i.e., policies) produce good animations in most cases, there are some cases in which they can appear too predictable. This is due to the fact that cognitive models are fundamentally deterministic (mapping states to actions). We now introduce an alternative form of our technique that addresses this problem. First note that, in some cases, it may be interesting to not always perform the same action for a given state (even if that action is most desirable). Occasional slight randomness in the decision making of an intelligent virtual character, performed in the right manner, can dramatically improve the aesthetic quality of an animation when predictability cannot be tolerated. However, it is not enough to simply choose actions at random, as this makes the virtual character appear very unintelligent. Instead, we do this in a much more believable fashion with a modification of the structure of our technique (see Figure 2.3b). We formulate it as a

priority function:

$$\text{priority} = P^{\mu}(i,a).$$

The priority function represents the value of performing any given action a from the current state i under a policy μ. The priority can simply be an ordering of the best action to the worst, or can represent actual value information (i.e., how much an action helps the character reach a goal state). Using a priority function allows us to query for the best action at any given state, but also lets us choose an alternative action if desired (with knowledge of that action's cost). For example, by using the known priorities of all candidate actions from the current state, we can select an action probabilistically. Thus our virtual character is able to make intelligent, but non-deterministic, decisions for all situations. However, note that while this non-deterministic technique is useful, we focus on standard policies in this paper. This is because they are simpler, faster, and correspond to the standard approach to cognitive modeling (i.e., always using the best possible action in a given state).

2.5.2 Training the Neural Network

We train the neural net using the backpropagation algorithm with examples of the cognitive model's decisions (i.e., policy). A naive approach is to randomly select many examples from the entire state space. However, this is wasteful because we are usually only interested in a small portion of the state space. This is because, as a character makes intelligent decisions, it will find itself traversing into only a subset of all possible states.

As an example, consider a sheepdog that is herding a flock of sheep. It is illogical for the dog to become afraid of the sheep and run away. It is equally illogical for the sheep to herd the dog. Therefore, such states should never be experienced in practice. We have found empirically that by ignoring uninteresting states, the neural net's training can focus on more important states, resulting in a higher quality approximation. However, for the sake of robustness, it may be desirable to also use a few randomly selected states that we never expect to encounter (to ensure that the neural net has at least seen a coarse sampling of the entire state space).

To focus on the subset of the state space of interest, we generate examples by running many animations with the cognitive model. At each iteration of an animation, we have a current state and the action decided upon, which are stored for later use as training examples. We have found that using a large number of examples is best to achieve a well-generalized trained network. Specifically, we prefer to use between 5,000 and 20,000 examples. Note that this is far more than is normally used when training neural nets, but we found that the use of so many examples helps to ensure that all interesting states are visited at least once (or at least a very similar state is visited). Finally, note that if a small time step is used between actions, it may be desirable to only keep an even subsampling of the examples generated through animation. This is because, with a small time step, it is likely that little state change will occur with each step and therefore temporally adjacent examples may be virtually identical.

We used a backpropagation learning rate of $\eta \cong 0.1$ and momentum of $\gamma \cong 0.4$ in all our experiments. Training a neural net took about 15 minutes on average using a 1.7 GHz PC. In all of our experiments, an appropriate selection of inputs to the neural net resulted in a good approximation of a cognitive model.

2.5.3 Choosing Salient Variables and Features

Training a neural network is not a conceptually difficult task. All that is required is to supply the backpropagation algorithm with examples of the desired behavior we want the network to exhibit. However, there is one well-known challenge that we need to discuss: selecting network inputs. This is critical as too many inputs can make a neural net computationally infeasible. Also, a poor choice of inputs can be incomplete or may define a mapping that is too rough for a neural net to approximate well. General tips for input selection can be found in [Haykin 1999], so we only briefly mention key points and focus our current discussion on lessons we have learned specific to approximation of cognitive models.

The inputs should be salient variables (no constants), which have a strong impact in determining the answer of the function. Further, if possible, features should be used. *Features* are transformations or combinations of state variables. This is useful for not only

reducing the total number of inputs but also making the input-output mapping smoother. Through experience, we have discovered some useful features that we now present.

When approximating cognitive models, many of the potential inputs represent raw 3D geometry information (position, orientation, etc). We have found that it is very important to make all inputs rotation and translation invariant if possible. Specifically, we have found it very useful to transform all inputs so that they are relative to the local coordinate system of the virtual character. That is, rather than considering the origin to be at some fixed point in space, transform the world such that the origin is with respect to the virtual character. This not only makes it unnecessary to input the character's current position and orientation, but also makes the mapping smoother.

We have also found it useful to, in some cases, separate critical information into distinct inputs. For example, if a cognitive model relies on knowing the direction and distance to an object in its virtual world, this information could be presented as a scaled vector (dx, dy, dz). However, we have found that in many cases it is better to present this information as a normalized vector with distance (x, y, z, d), as the decision-making may be dramatically different depending on the distance. In other words, if a piece of information is very important to the decision-making of a cognitive model, the mapping will likely be more smooth if that information is presented as a separate input to the neural net. Thus we need to balance the desire to keep the number of inputs low with clearly presenting all salient information.

Finally, note that choosing good inputs sometimes requires experimentation to see what choice produces the best trained network, as input selection can be a difficult task. However, recall that if storage is not a concern, k-nearest neighbor can be used instead of a neural network and (as described in [Mitchell 1997; Wilson and Martinez 2000]) can automatically discover those inputs that are necessary to approximate the target function.

Several practical examples of selecting good inputs for neural networks to approximate cognitive models are given in the results section of this paper.

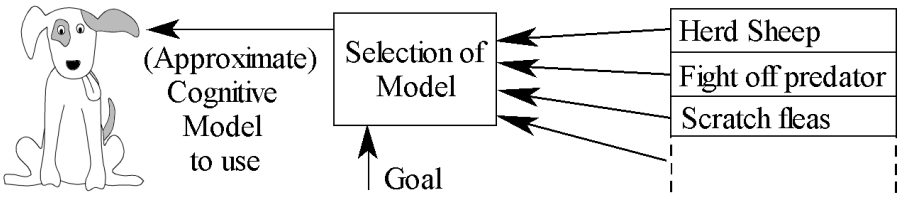

Figure 2.4: Example of a set of (approximate) cognitive models, integrated into a synthetic brain architecture for a virtual character — in this case, a sheepdog. The character's current goal determines which cognitive/behavioral model will be used.

2.5.4 Using the Neural Network

After a neural net is trained off-line, it can be used on-line to rapidly recall what action is best to take for any given state. If the network generalizes properly during training, it can produce high-quality approximations for states that were not explicitly represented in the training set. Further, a neural net of a reasonable size is very fast to execute, usually requiring far less than a microsecond. In fact, it can be executed in a fixed amount of time, unlike explicit planning with a cognitive model since a tree search is used (which can degenerate worst-case to visiting every node in the tree). This fixed-time feature makes neural net approximation more applicable to interactive computer graphics animation than using explicit cognitive models.

Since our neural net is trained to approximate a single policy, it is only useful for one cognitive model and goal. There is a similar limitation in the traditional technique for cognitive modeling [Funge et al. 1999], since the goal must be implemented directly into the cognitive model. In order to overcome this one-goal limitation, we must be able to use a set of models, where each model has its own goal (see Figure 2.4).

This is done by associating more than one neural net (or explicit cognitive model with integrated goal) with a virtual character. Each of these (approximate) cognitive models are independent, only one is used at a time, and the selection of which model to use depends on the character's current internal goal. In other words, the virtual character's brain has one or more (approximate) cognitive models, each capable of controlling its behavior to accomplish a specific goal. In fact, there can be more than one (approximate) cognitive model for any given goal, such that greater variety and/or robustness can be achieved.

Note that it is possible to use both neural nets and explicit cognitive models in the same character's brain if desired.

The neural networks produced by our technique (a set of approximate cognitive models) can be used in most recent synthetic brain architectures for virtual characters (e.g., "C1" and "C4" by the Media Lab at MIT [Blumberg and Galyean 1995; Blumberg et al. 2002]). Most brain architectures are modular and layered, with the cognitive/behavioral model to use at any given time selected based on the character's current goal and internal state. The model then operates in a modular fashion with the rest of the synthetic brain. Thus our technique naturally fits with these existing brain architectures, and we can achieve highly autonomous virtual characters with a variety of goals and behaviors.

2.5.5 Discussion

There is a great deal of preexisting validation for the approach we take in our technique, both in terms of AI theory and previous research. First, note that the difficulty of approximating a function with a neural net is directly related to how smooth the function is (this is analogous to the difficulty of fitting a polynomial to a curve). Note that a policy μ (if well formulated with vector-valued input and output) is virtually always a smooth function, because two similar states usually require similar or identical actions. Therefore, μ is an ideal candidate for neural net approximation.

Of course, since a neural net only approximates a policy, we are not guaranteed exactly correct results. However, a *properly trained* neural net should never make a "gross" error, as it is trained to minimize the mean-squared-error. In other words, if a mistake is made, it should be a small one. Since our goal is believable animation (which does not require exactness), a good approximation of a policy is sufficient. Besides, it is likely that we can achieve better results with a neural net approximation than an explicit cognitive model anyway. This is because, for planning to be done in real-time using an explicit cognitive model, short sub-optimal plans must be used. However, since in our technique we train a neural net off-line, we can use high quality, optimal plans as the training examples, leading to better real-time results.

Another benefit of using a neural net approximation is that, since planning does not

have to be done in real-time, we can use large (or continuous) state and action spaces. To support continuous state spaces, we simply need to have a sufficient set of training examples to demonstrate the solution space. As discussed previously, we have found that using 5,000 to 20,000 examples is sufficient. It is also possible to support continuous action spaces by finely discretizing the continuous action space. This provides a finite branching factor for planning, but also lets us generate training examples that are nearly continuous in nature.

The primary weakness of our technique is the fact that care must be used when selecting the net's inputs (i.e., it is not obvious how to design a neural net to approximate an explicit cognitive model). This means that a new skill must be acquired to effectively use our technique, even if publicly available neural net software is used to create and train the nets. Therefore, it may be preferable to use the *k*-nearest neighbor algorithm to provide an approximation of the cognitive model (see [Mitchell 1997; Wilson and Martinez 2000] for how inputs can be automatically selected). However, this alternative approach requires the explicit storage of many examples of the target function, and therefore should only be used if storage is not of concern.

2.6 Off-Line Character Learning

In this section we introduce off-line character learning for autonomous virtual characters. By *off-line character learning*, we mean a character automatically learning an unknown behavioral or cognitive model (i.e., learning to perform a task on its own). This is interesting because it can alleviate a large part of the animator's workload.

We have developed a novel technique to perform off-line character learning, using a tree search to compute discrete examples of a policy. These examples are then generalized into an approximate behavioral/cognitive model realized through a neural network. We will briefly review reinforcement learning (machine learning without a teacher) to lay a foundation for our discussion, and then introduce our technique for off-line character learning.

2.6.1 Background

In *reinforcement learning*, the machine learning of an input-output mapping is performed through continued interaction with an environment in order to maximize a scalar index of performance. This performance index is called a *fitness function*. Some of the earliest research in computer graphics involving reinforcement learning sought to have virtual characters automatically learn how to walk, swim, or jump optimally [van de Panne and Fiume 1993; van de Panne et al. 1994; Sims 1994; Grzeszczuk and Terzopoulos 1995]. As an example, the fitness function for walking was the distance traveled in a unit of time. However, while interesting and useful, this type of learning does not provide characters with decision-making abilities. Behavioral learning in computer graphics has only begun to be explored (e.g., [Blumberg et al. 2002]).

The goal of reinforcement learning is to automatically learn an optimal policy, μ^*. By optimal, we mean that the policy always maps the current state to the best possible action according to the fitness function. The challenge is to find μ^* automatically and quickly. There are several techniques to do this (excellent surveys are given in [Kaelbling et al. 1996; Sutton and Barto 1998]). However, the most popular and general approach is known as *Q-learning* [Watkins and Dayan 1992].

In Q-learning, the agent (virtual character) learns by exploring its state-action space. This is done by trying different actions for each state to learn the fitness of all important state-action pairs. This state-action fitness information can then be used to determine which action is optimal for any given state. This is the optimal policy μ^*. Because there is often a prohibitively large table of state-task values to store, we must approximate it. This can be done with a neural net as discussed in [Kaelbling et al. 1996; Sutton and Barto 1998].

We have performed several experiments using Q-learning for character learning (i.e., to automatically learn an unknown behavioral or cognitive model), where the only information we gave our virtual character was a fitness function. This approach has proven to be very difficult. First, to get stable results, we have to approximate the state-action value table to a high accuracy. We have found that this can require a very large neural net. For this reason, learning the state-action values can take days on a current PC. Second, as

discussed in [Sutton and Barto 1998], Q-learning can be very difficult to get to work in practice, especially since it can require visiting every state-action many times.

It is our opinion that the difficulties that accompany Q-learning for our desired application make it an undesirable approach. For this reason, we have developed an alternative approach to character learning based on planning-based reinforcement learning, but with several novel particulars. We now present this technique.

2.6.2 Off-Line Character Learning Through Searching

The technique for off-line character learning we have developed is designed for stability, simplicity, and speed. Thus, we believe it will be more useful to the computer graphics community than a technique based on Q-learning. Note that this technique is not guaranteed to find an optimal policy (which explicit Q-learning is), but explicit Q-learning is usually impractical anyway because of a large state-action value table. Also, we will show that our technique consistently produces good results, is computationally bounded, takes far less time than Q-learning, and is simple to implement and use. It also has a firm foundation in AI theory, as it is related to techniques presented in Chapter 9 of [Sutton and Barto 1998].

To generate training examples for our optimal policy neural net, we take an approach similar to traditional planning. Starting at a current state i, we use a tree search (as in Figure 2.1) to formulate a plan. The tree search continues until the minimum cost path to a specified depth of the tree is found (the cost of each state-action is determined by the reciprocal of the fitness function). Thus no terminal state (ending to the animation) is required, and computational time is bounded. Since this tree search is done off-line, we can search many levels deep in the tree (e.g., 25 levels). This allows us to be very confident in the partial plan we have formulated. Once the plan has been formulated, we keep the action a chosen for the initial state i as the training example (i.e., $\mu(i) = a$). We can then reuse the latter portions of this plan to find solutions for states that we transition into, speeding up the process dramatically.

For performing the tree search, we have found it useful to use either the popular A* algorithm or a best-first branch-and-bound algorithm. We prefer A*, as it has proven to

be the fastest in our experiments. However, A* requires an *admissible heuristic* (a conservative estimate of the total cost to reach a goal state), which is not difficult to design but does require some experience to do so effectively. The advantage to a best-first branch-and-bound search is that it is generic, and thus can be used for any fitness function without modification. See [Russell and Norvig 2003] for more general information on tree searching.

The tree search depth limit to use is an important question, as it limits how far ahead the character will consider its actions. On one hand it is useful to limit this depth to make the off-line learning algorithm as fast as possible, but on the other hand we want the character to succeed at its task. Setting the tree search depth is obviously task specific. A heuristic we have found to work well in practice is to set the depth limit based on minimum reaction time required by the actor. In other words, the character needs to consider far enough into the future that it will have sufficient time to prepare appropriately for upcoming situations. For example, a virtual spaceship pilot needs to start turning well in advance if she is to dodge a large asteroid in her path.

The biggest challenge we have encountered (with both this technique and character learning using Q-learning) is designing a fitness function that produces exactly the results we want. It is important to note that the fitness function is the only control we have over the unknown behavioral or cognitive model our character learns. We will discuss this issue in detail in the next subsection.

Note that our technique relies on the assumption that the task being learned is non-context sensitive. Q-learning and other reinforcement learning techniques make this same assumption, plus more (they are Markovian). This non-context sensitive assumption is usually not a problem for us, as non-context sensitive policies have been shown in the literature to be capable of performing very complex tasks. However, we have found that being context sensitive can be very useful for virtual characters from the perspective of portraying emotion (e.g., “happy,” “angry,” “afraid,” etc). A simple approach we use to learn context sensitive policies is to supply a different fitness function for each context. A different policy is then learned from each of these fitness functions (one for each context).

These policies are then placed in our character's brain, with the selection of which neural net to use determined by the character's current internal state.

2.6.3 Designing Fitness Functions for Character Learning

As mentioned in the previous subsection, correctly designing the fitness function is a critical task because it is the only control we have over the unknown policy that our virtual character will learn. Indeed, fitness function creation is known to be a non-trivial task [Mitchell 1997; Sutton and Barto 1998].

We have found that a good fitness function for character learning should have the following features. First, it should be smooth and continuous (i.e., similar states having a similar fitness), which helps avoid temporal aliasing in the animation. Second, it should have a term restricting drastic actions (e.g., very sharp turns in a spaceship), which helps achieve more realistic and aesthetically pleasing animation. Third, there should be a term specifically rewarding actions that, from a task-specific standpoint, are desirable even though they may not represent the best choice with respect to reaching a goal state as quickly as possible.

One final tip: we have found it useful to experiment with different fitness functions to see which produces the most desirable results. Often it is difficult to determine ahead of time the exact fitness function that will achieve the desired behavior for a character. This can be overcome by making small incremental improvements in the fitness function and repeatedly testing it.

2.6.4 Discussion

Our planning-based character learning technique has proven to be fast, simple, and robust (see Section 2.7 for details). We have also succeeded in automatically learning behavioral and cognitive models for difficult and interesting tasks. In our experiments, designing fitness functions required on average a few minutes of work; off-line learning required one hour on average per policy using a 1.7 GHz processor. This proved significantly faster than constructing explicit behavioral and cognitive models, which often take several

days or even weeks for experienced designers to design and program. Further, we were able to learn approximate models for tasks that we were unable to devise explicit models for.

Note that our planning-based character learning technique will only learn a sub-optimal policy, the quality of which is based on the search depth limit used. However, optimality is probably not necessary to achieve intelligent-appearing behavior in a virtual character (i.e., good behavior usually looks just as realistic, and perhaps sometimes more so, than perfect behavior). Further, note that in practice Q-learning is not optimal either, as it can only achieve optimality after visiting every state-action an infinite number of times.

The difference between learning a behavioral model versus a cognitive model is merely the tree search depth limit: a short look-ahead results in reactive behavior, whereas a long look-ahead maximizes long-term utility. Therefore, in the approach we take, these two types of models are realized in exactly the same way and are differentiated with merely the adjustment of a parameter.

2.7 Experimental Results

We implemented our techniques (rapidly approximating an explicit cognitive model and off-line character learning) and used them to perform a series of experiments. We report our findings in this section. These experiments were designed to cover, in a general way, all major distinguishing aspects of behavioral and cognitive tasks (e.g., temporally coarse- or fine-grain action selection, continuous or discrete state and action spaces, simple or complex state information, etc). We used single-precision floating point in our neural nets, which doubles the performance of the division and power operations with very little loss in quality. All animations were rendered in real-time using OpenGL on a 1.7 GHz PC with an ATI Radeon 8500 (commodity) video card. Quicktime videos are available from http://rivit.cs.byu.edu/a3dg/publications.php.

Figure 2.5: Snapshots of a skeleton herding a group of humans. All characters are articulated figures. The skeleton is controlled by a cognitive model that selects high-level actions, which are carried out by a skeletal animation system. We first constructed an explicit cognitive model, and then had our character automatically learn a model through off-line character learning. The neural net approximation was significantly faster to execute than the explicit cognitive model (about 1 μs vs. 0.2 sec). Also, character learning required less time than for us to program an explicit model (a few hours vs. a few days).

2.7.1 Herding a Group of Characters

The experiment of herding is interesting for putting our work in perspective, as it has been used to test many previous techniques (e.g., [Funge et al. 1999], where a large dinosaur herded smaller ones). Also, this experiment is a good test case for behavioral/cognitive models that are applied at a high level in the animation hierarchy, with temporally coarse-grain action selection. This is important, as some of the most impressive results to date in the literature have been achieved with high-level, temporally coarse-grain models.

In our experiment (see Figure 2.5), the character performing the herding is a skeleton, and the characters being herded are humans. These characters are articulated figures, whose motor control is performed through skeletal animation. The virtual world is split up into a grid of cells (i.e., the state space is discreet). The skeleton and humans are located in one cell at a time. They can each move to an empty adjacent cell only (i.e., the action space is discreet). If the skeleton gets too close to the humans, they will run away in the opposite direction. They also remain in a group (or "flock") whenever possible. The skeleton's task is to move all humans to a goal location in the virtual world. Once at the goal location, the humans cannot leave and are thereafter ignored by the skeleton. The humans are controlled by a simple reactive system, whereas the skeleton is empowered with a cognitive model.

Note that the skeleton's cognitive model performs temporally coarse-grain action selection, as the actions require a notable amount of time to perform (e.g., about a second). Thus actions are selected occasionally, with lower levels in the animation hierarchy carrying them out (e.g., motor control).

The inputs we chose for the neural net were the distances from the skeleton to the nearest member of up to two groups of humans. This was represented as two (i, j) vectors. The only other inputs were the distance vector to and size of the nearest obstacle, and the direction toward the goal location. Thus we had seven inputs, resulting in a very small neural net that was extremely fast to execute.

We implemented this experiment with both an explicit cognitive model we programmed and with an automatically learned model (the fitness function measured how close on average all the humans were to the goal location). Though the approach taken to solve the problem was different between the two models, they both solved it well. However, the explicit model took three days to program, whereas the fitness function only took a few minutes. Thus we see that character learning can not only simplify the process of creating a behavioral/cognitive model, but also dramatically shorten the development time. Also, note that the neural net required less than a microsecond to execute, whereas the explicit cognitive model required about 0.2 seconds on average to compute an action.

2.7.2 Spaceship Pilot and Asteroids

In this experiment, the virtual character was a spaceship pilot (see Figure 2.6). The pilot's task was to maneuver the spaceship through an asteroid field (along the Z-axis), flying from one end to the other as quickly as possible with no collisions. To ensure that this would be a significant problem, we limited the maneuverability of the spaceship so that the pilot would have to plan his path through space well in advance. We also placed the asteroids close together. The animation ran at 15 frames per second, with an intelligent action computed for each frame. Thus the model was applied at a fairly low level in the animation hierarchy, and action selection was temporally fine-grain.

The virtual pilot had two controls over the spaceship: yaw (rotation around the Y-axis) and pitch (rotation around the X-axis). The controls were real-valued (i.e., the action space

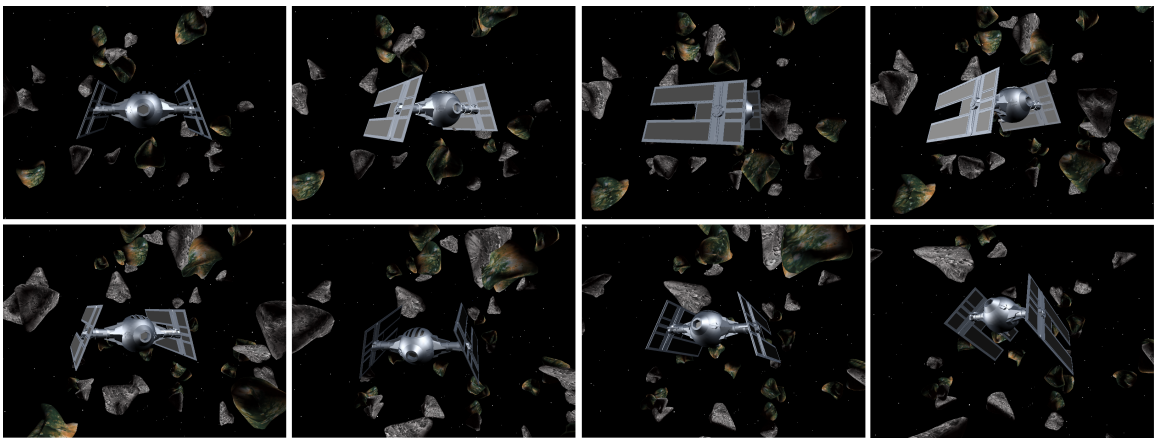

Figure 2.6: A cognitively empowered spaceship pilot intelligently maneuvers within an asteroid field. The pilot's goal is to cross the asteroid field (forward motion) as quickly as possible with no collisions. Due to the temporally fine-grain action selection in this experiment, neural net approximation was necessary to achieve real-time performance.

was continuous). Also, the spaceship could be at any location and orientation (i.e., the state space was continuous). The inputs we selected for the neural net were the spaceship's current orientation (θ, ϕ), and the rotation-invariant normalized direction (i, j, k) and distance (d) to the three nearest asteroids. Thus there were 14 inputs total. The network had two outputs, which determined the change in the spaceship's orientation.

We first programmed an explicit cognitive model. Since the ideal action space was continuous, we had to discretize it dramatically to achieve real-time performance (there were only 9 possible actions the pilot could do). We also had to limit planning to a tree depth of 5 levels. The final result was a poor animation, due to the fact that the pilot could not plan far enough ahead to adequately maneuver the spaceship around the asteroids. Also, because the discretization of the action space was so coarse, the motion was not very smooth.

In our next experiment we improved the planning of our explicit cognitive model, taking advantage of the fact that we could perform our tree search off-line and then train a neural net to "memorize" the correct actions. This significantly improved the results, as we were able to formulate much better plans and also use a more fine grain discretization of the action space. Using a fast neural net approximation of this improved (but slower)

explicit cognitive model, the spaceship pilot was able to dodge all asteroids, and the motion was smooth and aesthetically pleasing. This neural net animation utilized very little CPU: about a microsecond for each execution. Comparatively, the high-quality explicit model we produced required approximately 0.5 seconds to compute an action, and thus was not able to maintain interactive rates.

Finally, we tried automatically learning a cognitive model (off-line character learning). The fitness function was determined by how direct of a route the pilot took, without hitting any asteroids. Specifically, the reward was equal to the forward component of the spaceship's motion vector, and a very large penalty was given for hitting an asteroid. We achieved excellent results, learning a cognitive model that crossed asteroid fields faster than our explicit cognitive model and did so in a visually pleasing manner. The most challenging part of this task was in determining the fitness function that allowed us to achieve the exact "look" that we wanted the pilot's actions to have.

2.7.3 Spaceship Battle

Our next experiment involved a battle between two spaceships. The pilot's controls were identical to the previous experiment, except that the pilot could also fire a laser. The inputs were the relative orientations of the two spaceships, their relative positions, and any nearby lasers to avoid (all according to the pilot's local coordinate frame). The fitness function was very simple: a reward for avoiding the other spaceship's nose (where the laser was mounted), a reward for shooting the other spaceship, and a punishment for being shot. We did not attempt to program an explicit behavioral/cognitive model because we had no idea how one should go about piloting a spaceship in combat (note that studying *how* to accomplish a task is usually necessary before one can program explicit AI to accomplish that task, thus our technique for character learning relieved us of this burden in this experiment).

Note that a challenge for our character in learning this task is that we must know how one pilot will behave for the other pilot to learn how to combat him. However, the problem of learning in competitive environments has been thoroughly explored (e.g., [Reynolds 1994]). We did this by iteratively learning better cognitive models for both pilots by having them compete. In other words, after one pilot learns a new model (and is therefore better at

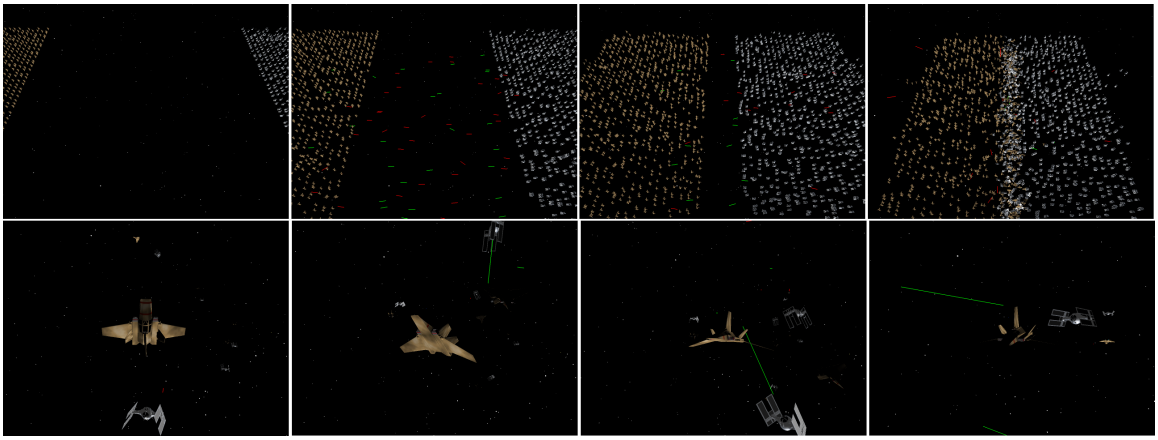

Figure 2.7: With the high computational speed of our technique, it is possible to perform animations of an "epic" scale in real-time. Here we have 2,000 spaceship pilots in combat. Our pilots are self taught (i.e., character learning), so this animation took very little human effort to create.

his task), we then use him as an example for the other pilot to learn a new model. However, it is also possible (and perhaps simpler in some situations) to simply construct a trivial reactive model for one pilot and then learn a superior cognitive model for the other pilot.

As in our previous experiments, we achieved good results with our learned cognitive model. In addition, the neural net execution time was very fast (approximately one microsecond per iteration). Due to this performance, we were able to achieve a spaceship battle of "epic proportions" on our PC (see Figure 2.7). Specifically, we had two thousand of these spaceship pilots locked in combat in real-time. Interestingly, the bottleneck was not executing the neural nets, but rendering all the spaceships. To achieve real-time performance, we had to use very simplified meshes. We believe this large-scale real-time animation ability for highly intelligent virtual characters is one of the most important contributions we make in this paper.

2.8 Conclusions and Future Work

In this paper, we have presented two novel techniques. First, we use machine learning (in our system usually neural networks) to quickly approximate cognitive models. This

allows us to achieve performance never before possible (several thousand intelligent autonomous characters in real-time on a PC). Further, because training is done off-line, we can use much larger action spaces and higher quality plans than previously possible.

The second technique we have introduced is off-line character learning. Through this method, a character can automatically learn an unknown behavioral or cognitive model on its own with nothing more than a fitness function to guide it, alleviating the animator from the workload of programming an explicit model. This also allows us to model tasks for which it would be difficult or virtually impossible to develop an explicit model.

However, there are some weaknesses in our approach that are important to recognize. First, since a neural net only approximates a mapping, we are not guaranteed exactly correct results. However, since a neural net is trained to minimize the mean-squared-error of the training examples, we are guaranteed that the network will make no "gross" errors *if it has been trained properly*. Another issue of our technique is that a net's inputs must be chosen with care. It is important that salient variables and features in the state be identified and used as the inputs. However, this problem can be avoided or reduced through the use of k-nearest neighbor, where inputs can be selected automatically (but at the cost of extra storage). Next, note that when performing off-line character learning, it can be difficult to design a fitness function that results in the exact behavioral/cognitive model that is desired. However, we have found it to be easy and quick to achieve a good model. Finally, note that some behavioral/cognitive models have many salient variables. Approximating these models could require a very large, slow neural net, and therefore it may be necessary to use an alternative machine learning technique (which suffers less from the curse of dimensionality with respect to performance).

There are some exciting areas open for future work. For example, we have only presented a technique for off-line character learning. On-line learning in the literature has thus far been limited to behavioral models (short-term utility). Is it possible to learn a cognitive model on-line in an interactive application? This could take interactive computer graphics to a whole new level, especially in the entertainment market. This could also be interesting if an animator could interactively train a virtual character for cognitive learning, rather than using a fitness function.

Chapter 3

Improved Behavioral Animation Through Regression

Proceedings of Computer Animation and Social Agents, pp. 231–238, 2004.

Abstract: Behavioral and cognitive modeling have become popular for creating autonomous, self-animating virtual characters. However, existing techniques have some weaknesses. In particular, cognitive models are usually very computationally expensive, limiting their usefulness. Also, behavioral and cognitive models can behave unexpectedly since it may be impossible to exhaustively test the model for the entire input space (especially if the input space is continuous).

In this paper we present a general technique for approximating behavioral and cognitive models through regression with machine learning. This provides several benefits, such as fast execution in fixed time, and generalization using a finite set of known behavior examples. We examine the usefulness of alternative machine learning techniques for our problem of interest, and compare their strengths and weaknesses. We also present a custom method for automatic input selection, which helps simplify the process of machine learning to approximate a behavioral/cognitive model.

3.1 Introduction

Computer animation is often a costly endeavor, requiring a large amount of work from human animators. In the past decade, some notable research has been performed in developing techniques to reduce this workload through automation. One such automatic technique for computer animation is *behavioral animation* [Reynolds 1987] (see Figure 3.1). In behavioral animation, virtual characters are designed to be autonomous agents, intelligent enough to animate themselves at a high level. Specifically, a character selects its own actions, which are carried out by a motor-control module. Several powerful and popular behavioral animation systems have been introduced [Reynolds 1987; Funge et al. 1999; Monzani et al. 2001; Devillers et al. 2002; Blumberg et al. 2002]. Excellent surveys of behavioral animation techniques and related topics can be found in [Millar et al. 1999; Pina et al. 2000].

There are two primary approaches to behavioral animation. A *behavioral model* [Reynolds 1987] is an executable model defining how the character should react to the current state of its environment. Alternatively, a *cognitive model* [Funge et al. 1999] is an executable model of the character's thought process, allowing it to deliberate over its possible actions (e.g., through a tree search). Thus a cognitive model is generally considered more powerful than a behavioral one but can require significantly more processing power. As can be seen, behavioral and cognitive modeling have unique strengths and weaknesses, and each has proven to be very useful for virtual character animation.

However, despite the success of these techniques in certain domains, there are two notable limitations which we address in this paper. First, cognitive models are traditionally very slow to execute, as a tree search must be performed to formulate a plan. Thus the character can only make sub-optimal decisions, and the number of virtual characters that can be used simultaneously in real-time is limited, and it is necessary to use only a small set of candidate actions. Second, behavioral and cognitive models can act unexpectedly, producing undesirable behavior in certain regions of the state space. This is because it may be impossible to exhaustively test the model for the entire state space (especially if the state space is continuous). This can be worrisome for end-user applications involving autonomous virtual characters, such as training simulators.

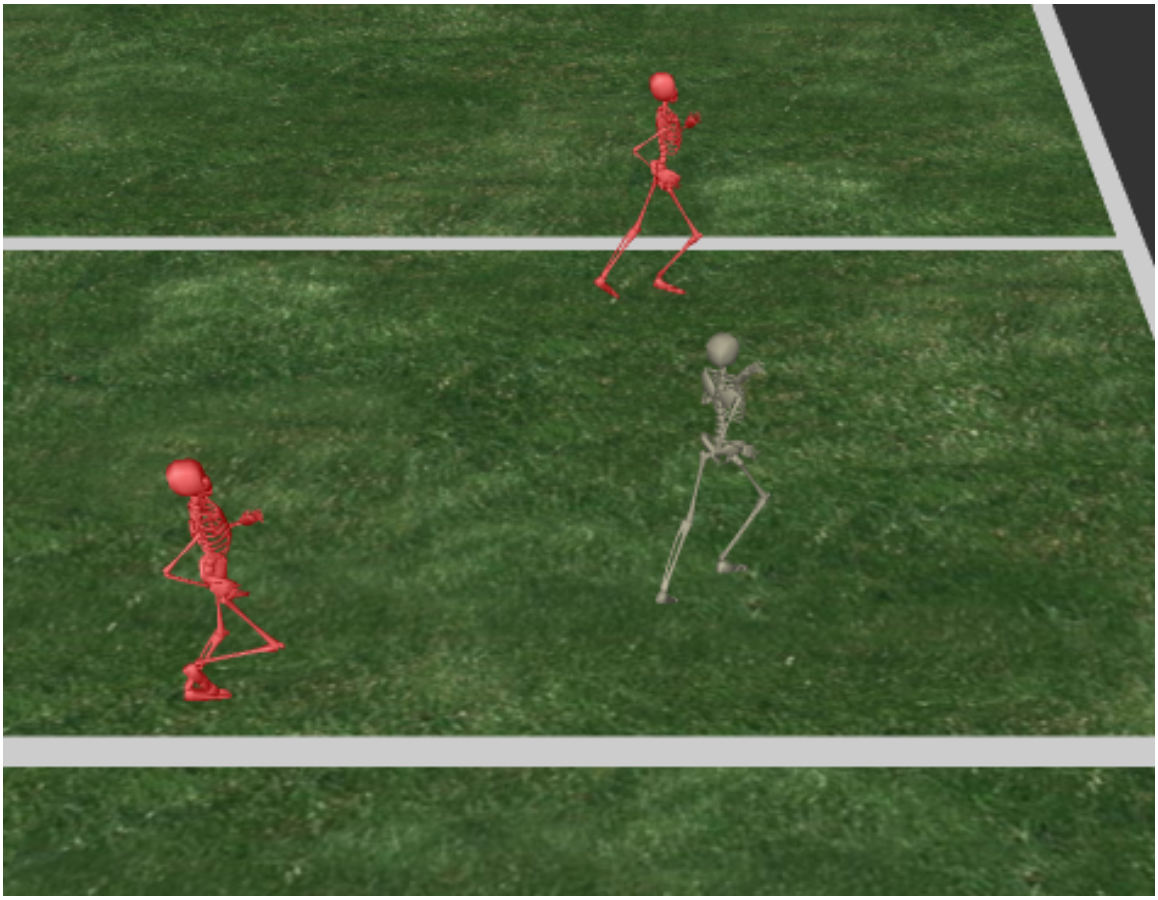

Figure 3.1: Example of behavioral animation in an interactive virtual world. A human user is playing rugby against a group of characters.

In this paper, we build off our previous work reported in [Dinerstein et al. 2004b]. We introduced a novel technique for rapidly approximating behavioral/cognitive models through regression with artificial neural networks. The purpose of that technique was to help eliminate the problems listed above, and it succeeded to some degree. However, there are many interesting and powerful machine learning methods, each with unique strengths and weaknesses. As a result, since the technique in [Dinerstein et al. 2004b] uses only neural networks for regression, it is inherently limited and one-sided. Also, that paper does not address the problem of *input selection*, an important initial step in machine learning which often requires more programmer time and effort than any other step.

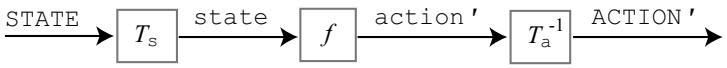

Figure 3.2: Overview of our model regression method.

Our contributions in this paper include a general technique for approximating behavioral/cognitive models through any machine learning technique that supports a real-vector-valued formulation of inputs and outputs. We also discuss the respective strengths and weaknesses of several popular machine learning techniques for approximating behavioral/cognitive models through regression. We have found that local regression techniques (e.g., case-based reasoning) are often more useful than global regression techniques (e.g., neural nets) for our problem of interest, as character behavior is often not a simple, smooth mapping. This is especially true when sub-optimal inputs have been selected. We also present a custom method for automatically performing input selection, designed specifically for regression of behavioral/cognitive models. This method greatly simplifies and speeds up the machine learning process for the programmer, and often selects better inputs than the programmer as well.

3.2 Regression of Behavioral and Cognitive Models

3.2.1 Formalism

We now present our general technique for approximating behavioral/cognitive models through machine learning (see Figure 3.2). For an introduction to machine learning, see [Mitchell 1997].

A behavioral or cognitive model uses the virtual character's perception of the current state of its virtual world to select the next action to perform. More formally, a behavioral/cognitive model performs a $\texttt{state} \rightarrow \texttt{action}$ mapping. By representing states and actions as real-vector-valued points of fixed dimensionality n and m ($\texttt{state} \in \mathbb{R}^n$ and $\texttt{action} \in \mathbb{R}^m$) we have a mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$. Thus our regression problem is:

$$f : \texttt{state} \in \mathbb{R}^n \rightarrow \texttt{action}' \in \mathbb{R}^m, \tag{3.1}$$

where action′ signifies that it is approximate. This real-vector-valued formulation is important as it is a very general format, and therefore useful for our needs in creating a general model approximation technique.

Not all behavioral/cognitive models use real-vector-valued representations for their states and actions. However, most alternative state and action representations can be converted to a real-vector-valued form through simple, custom transformations (and vice versa):

$$T_{\mathtt{s}} : \mathtt{STATE} \rightarrow \mathtt{state}, \tag{3.2}$$

$$T_{\mathtt{a}} : \mathtt{ACTION} \rightarrow \mathtt{action}, \tag{3.3}$$

$$T_{\mathtt{a}}^{-1} : \mathtt{action} \rightarrow \mathtt{ACTION}, \tag{3.4}$$

where caps signify the external format of states and actions (see Figure 3.2).

It is important, for the sake of generalization, that our real-vector-valued states and actions be organized such that similar states usually map to similar actions. More formally:

$$(\|\mathtt{state}_1 - \mathtt{state}_2\| < \varepsilon_a) \Rightarrow (\|\mathtt{action}_1 - \mathtt{action}_2\| < \varepsilon_b), \tag{3.5}$$

where $\| \cdot \|$ is the l2-norm, and ε_a and ε_b are small scalar thresholds. Certainly, this constraint need not always hold, but the smoother the mapping the simpler it will be to learn. Moreover, if possible, it can be useful for the mapping to be C^0 and C^1 continuous. Of course, the importance of these constraints vary depending on the machine learning technique used. We will discuss this in more detail later in this paper.

Regardless of the machine learning technique utilized, n (the input dimensionality) must be kept as small as possible. This is due to the "curse of dimensionality", a famous thesis in machine learning stating that the difficulty of learning a mapping increases exponentially for each additional input. Therefore, the behavioral/cognitive model we wish to approximate should require as little information about the current state of the virtual world as possible, and this information should be presented to the machine learner in a compact form. If the state space representation cannot be compressed enough to effectively learn the desired behavior, it may be necessary to modify the model we wish to approximate to better meet the requirements of machine learning. This is usually possible, but there are some types of decision-making that fundamentally require a lot of state information (e.g., the game of chess) and therefore may never be good candidates for regression.

To approximate a behavioral/cognitive model, a finite set of discrete `state` $\rightarrow$ `action` examples of the model's decision making needs to be assembled. This can be done by running internal, undisplayed animations using the behavioral/cognitive model and recording a subset of its input-output pairs. The selected machine learning technique can then use this set of examples to perform regression. Regardless of the learning method used, these examples should represent all regions of the state space, illustrating the entire scope of decision-making the character may engage in. Moreover, it can be useful to vary the density of the examples according to the importance of each region of the state space (i.e., how often that region is visited by the character).

Behavior regression, as formulated in Equation 3.1, can only provide a deterministic and Markovian approximation. We have found in our experiments that these limitations are usually acceptable, and perhaps even preferable since it helps keep the regression problem tractable. Nevertheless, there will be situations where a behavioral/cognitive model cannot be cast as a deterministic Markovian process. A non-Markovian formulation of our technique is:

$$\texttt{action}' = g(\texttt{state}, \texttt{context}), \tag{3.6}$$

where `context` can be either the character's internal state or the last few actions performed. However, we usually do not need to explicitly input context, because if there are a few discrete contexts (e.g., a small number of emotional states or goals) we can use a separate machine learner to approximate each individually:

$$\begin{aligned} \texttt{action}' &= f_1(\texttt{state}), \\ &\vdots \\ \texttt{action}' &= f_p(\texttt{state}), \end{aligned}$$

where each learned function f_i corresponds to one context. For most behavioral/cognitive models, though, the decision making is Markovian and we can simply use a single instance of Equation 3.1.

To achieve a non-deterministic approximation, we use the following formulation of our technique:

$$\texttt{value} = h(\texttt{state}, \texttt{action}), \tag{3.7}$$

where `value` is the expected utility of performing `action` in the current situation `state`. After utilities are computed for a (sub)set of actions, they are ranked and one is selected probabilistically. Thus a character can stochastically select actions in an intelligent manner. However, the input dimensionality of Equation 3.7 is higher than that of Equation 3.1, so we prefer to use deterministic regression whenever possible.

Our technique is actually quite scalable, since a behavioral/cognitive model can be approximated by several separate machine learners, each of which learn a distinct subset of the state-action mapping. For example, decision-making in different regions of the state space may rely on different state information, and therefore these machine learners can use different state formulations (reducing the dimensionality). Similarly, if a virtual character has several distinct candidate goals, these can be learned separately. To allow for smooth switching between learners during animation, the actions recommended by each can be blended for a period of time. Since our actions are real-vector-valued, this can be performed through a weighted vector average.

3.2.2 Our implementation:

In our experiments, we created our virtual worlds and characters with the needs of regression in mind. Therefore, we defined the external state and action spaces to be real-vector-valued. As a result, no action transformations $T_{\mathtt{a}}$ or $T_{\mathtt{a}}^{-1}$ are required. However, we do use a transformation $T_{\mathtt{s}}$ to convert a complete external state `STATE` into a compact internal form `state`. Specifically, only information pertinent to the current character is retained, and this information is converted into a compact set of features. Most features are constructed by making angles and distances between characters/objects translation and rotation invariant according to the current character's frame of reference. We will discuss creation of features in more detail later in this paper.

To ensure that the input dimensionality is tractable, we use approximate state information whenever possible. While such state approximations limit the accuracy of learning a mapping, they can significantly reduce the dimensionality and thereby make learning tractable.

To ensure that we have a representative set of `state` $\rightarrow$ `action` examples, we run

several internal (i.e., non-displayed) animations. We regularly sample and record the behavioral/cognitive model's input-output pairs (e.g., every fifth decision). Since we record examples over several animations, we are likely to get data for most of the state space and the density of the data corresponds to those regions most often visited by the character. In our experiments, we used between 5,000 to 65,000 examples. The number necessary varies depending on how smooth the mapping is and the machine learning method used.

Of course, our technique for regression of behavioral/cognitive models only replaces the decision-making module of a model. Other important modules such as perception and motor control need not be altered.

3.3 Comparison of Machine Learning Techniques

Our technique for approximation of behavioral/cognitive models can be used with any machine learning method, providing that it allows real-vector-valued inputs and outputs. This is the case with the most popular and common machine learning methods for regression.

In the following subsections, we consider several notable machine learning techniques. We examine the strengths and weaknesses of each technique, as they pertain to approximation of behavioral and cognitive models. Our findings are drawn from several experiments, as well as theoretical considerations. Our experimental test beds are listed in Figure 3.3, and encompass a wide range of virtual characters, environments, and target behaviors. The results of our experiments are summarized in Table 3.1.

3.3.1 The artificial neural network (NN):

The artificial neural network or NN is a *global* regression technique (i.e., the entire network contributes in computing an answer). It was the machine learning method of choice in our previous work [Dinerstein et al. 2004b].

NN proved to work well in many of our experiments, as detailed in Table 3.1. However, because it is a compact and global technique, it only worked well when the mapping to learn was fairly smooth and continuous. As a result, it often took several days to design

(a) (b) (c)

Figure 3.3: We used several test beds in our experiments. These include: (a) 3D asteroid field navigation; (b) flocking and herding; (c) virtual rugby. We used cognitive models for asteroid navigation and virtual rugby, and a behavioral model for flocking behavior.

an effective, compact state representation to use as input for the neural net. However, even with a near-optimal state representation, some of the more complex character behavior was never learned well since it was still too irregular or of too high dimensionality. Thus scalability is a big issue when using a NN. To combat this, we often found it necessary to train several NN's for a single behavioral/cognitive model, each NN covering a distinct subset of the state space. Nevertheless, once adequate regression was achieved, the resulting animations were smooth and pleasing due to good generalization.

NN requires few training examples ($\sim$ 5,000 to 15,000) since it generalizes well when learning. This can be a useful property, since generating examples can be computationally expensive. Moreover, due to powerful generalization, NN tends to blend out noise, mistakes, and aliasing in the decision-making examples. However, this powerful generalization also means that unique local behavior is likely to be blended out. Moreover, behavior discontinuities are likely to be smoothed.

3.3.2 The support vector machine (SVM):

The support vector machine or SVM is another *global* regression technique, and is related to NN. The only primary difference with respect to our needs between SVM and NN is that SVM training is guaranteed to achieve global minimum mean-squared-error, whereas NN training (backpropagation) can converge to a local minimum. We used the radial basis kernel and epsilon-regression training method in our experiments.

		k-nn	NN	SVM	Other
Asteroids	***Execution time***	16 μsec	2 μsec	5 μsec	6 μsec
	NMSE	0.00042	0.0021	0.0015	0.0038
	Storage	1.6 MB	1.2 KB	17 KB	$\sim$ 1 MB
Flocking	***Execution time***	13.8 μsec	1.5 μsec	4.8 μsec	5.9 μsec
	NMSE	$8.1_{E^{-5}}$	0.0013	0.00093	0.0025
	Storage	0.88 MB	0.4 KB	4.4 KB	$\sim$ 1 MB
Rugby	***Execution time***	15 μsec	1.8 μsec	5 μsec	5.9 μsec
	NMSE	0.00039	0.0021	0.002	0.0035
	Storage	1.1 MB	0.6 KB	13 KB	$\sim$ 1 MB

Table 3.1: Typical performance results of our behavioral/cognitive model approximation technique, utilizing different machine learning algorithms. *NMSE* denotes normalized mean-squared-error (i.e., output $\in [0,1]$) between the explicit model and learned approximation. We used a 1.7 GHz PC with 512 MB RAM in these experiments. In comparison, our asteroid navigation cognitive model required approximately 0.5 seconds to compute a decision.

SVM proved to have similar strengths and weaknesses to traditional NN's. In particular, with both of these machine learning techniques it proved necessary to use an effective, compact state space formulation. The only notable benefit we found to using SVM over NN was that the approximation error was usually somewhat smaller. However, visually, the results usually appeared indistinguishable from a traditional NN in most cases. This is likely due to the fact that our real-vector-valued formulation of actions is somewhat tolerant of noise and error. In our experience, SVM requires two to three times as many training examples as NN.

3.3.3 Continuous *k*-nearest neighbor (*k*-nn):

Continuous *k*-nearest neighbor is probably the most well-known *local* machine learning technique, and is an example of case-based reasoning. Unlike NN and SVM which are compact, *k*-nn keeps a library of all examples of the target mapping it has been provided. To compute an output for a given input, the *k* examples closest to the input (according to the Euclidean metric) are found and their associated outputs are distance-weighted and averaged.

K-nn has proven in our experiments to be very simple to use and work remarkably

well for regression of behavioral/cognitive models. This is primarily due to the fact that, unless the programmer carefully designs the state space representation to provide a smooth and simple state-action mapping, it is likely that the mapping will be quite rough. As a result, compact techniques like NN and SVM will fail to learn such a mapping well, whereas *k*-nn has no such trouble. However, since *k*-nn uses explicit examples, a suboptimal state space representation may have a higher dimensionality than necessary, requiring an exponentially-increasing number of examples to populate each additional dimension. Nevertheless, even with more input axes than necessary, storage requirements are still usually quite reasonable (e.g., usually < 2 MB). But *k*-nn does require more training examples than NN and SVM ($\sim$ 15,000 to 60,000) due to poorer generalization.

We have found that $k = 3$ works well, as this keeps regression quite local but generalizes sufficiently to provide smooth animation. We use a kd-tree to make the lookup of cases fast. We scale the input space axes (as described in [Mitchell 1997]) to minimize the mean-squared-error.

While *k*-nn has proven capable of performing adequate regression of rough mappings, such regression of rough mappings may result in jittery animation because of incorrect generalization of cases. Moreover, *k*-nn does not generalize powerfully like NN, so it is more prone to jittering due to noise or mistakes in the behavior examples. To combat these problems, we have found it potentially useful to temporally filter actions recommended by *k*-nn to eliminate high frequencies. For example, with a cognitive model, we examine the character's plan to determine whether the character's next action contradicts the following action:

$$\text{if } \frac{\texttt{action}_1}{\|\texttt{action}_1\|} \bullet \frac{\texttt{action}_2}{\|\texttt{action}_2\|} < \gamma \approx 0.4,$$
$$\text{then average } \texttt{action}_1 \text{ and } \texttt{action}_2.$$

3.3.4 Other machine learning techniques:

We also tried several other machine learning techniques, but the results were not interesting enough to warrant individual attention. They either produced poor results or non-remarkable results at the cost of using an unusual technique. Therefore, we now briefly summarize the rest of our findings.

Because we achieved such good results with *k*-nn, we also tried a few other local regression techniques. First, we tried a lookup table of adaptive resolution. The results were no better than *k*-nn, but the software was significantly more complex. We also tried replacing the simple weighting metric in *k*-nn with a radial basis function, but the accuracy was not notably superior.

Alternative forms of NN's and different SVM kernels performed much like the sigmoidal NN and radial-basis SVM we discuss above. A linear perceptron (i.e., single-layer NN) could not adequately approximate the desired behavior in any of our experiments. Regression with a decision tree performed quite poorly, producing choppy animation.

3.4 Input Selection for Behavior Regression

Input selection (often called *feature selection*) [Guyon and Elisseeff 2003] is a well-known problem in machine learning. As discussed previously, all machine learning techniques suffer from the curse of dimensionality. Therefore, it is essential to carefully select and use only those candidate inputs that are necessary for the system to learn the target function. However, this is a difficult task, since it is often unclear which inputs are critical to adequately define a mapping. Thus it is attractive to use an automatic input selection technique.

For regression of behavioral and cognitive models, we often have many candidate inputs. This is due to the fact that any given variable contained in the full state of the virtual world could be a useful input. Therefore, we need an input selection technique that will robustly handle large sets of candidate inputs, some of which are partially redundant and many of which are of no value.

Several automatic input selection techniques have been developed by the machine

learning and statistics communities [Guyon and Elisseeff 2003]. However, these existing techniques (in their traditional forms) are not a good fit for our needs. This is due to several factors. For example, the most well-known technique, Principal Component Analysis (PCA), does not consider the target output and therefore cannot differentiate between valid input data and noise. Many of these techniques are designed for classification rather than regression, some only reject noisy inputs, or are not robust when there are many candidate inputs, etc. As a result, we have developed our own custom method for input selection, which we present in this section.

First, note that we do not address the problem of *feature creation* in our input selection method. A *feature* is a high-level concept, constructed from raw, low-level variables. Features are usually better inputs than raw variables, because they can define a more learnable target function (i.e., smooth enough and of a sufficiently low input dimensionality). However, there are no mature and well-established theories or techniques for automatic feature creation [Thornton 2003]. Thus feature creation has traditionally been left to the programmer. We follow this standard approach, and require that the programmer first specify a complete set or superset of useful inputs (features). Then our input selection technique automatically chooses a (suboptimal) minimal subset of inputs. This is useful because it is often not clear which inputs are needed to minimally but accurately represent a mapping. For a detailed discussion on creating features for behavioral animation, see [Dinerstein et al. 2004b].

Our approach to input selection proceeds as follows:

1. The programmer provides a (super)set of the inputs necessary to learn the target state-action mapping.

2. Determine linear correlation between the candidate inputs and output by computing the Pearson correlation coefficient: $\mathscr{R}(i) = (cov(X_i, Y))/(\sqrt{var(X_i)var(Y)})$. Reject all candidate inputs X_i where $\mathscr{R}(i)^2 < \alpha \approx 0.005$.

3. Perform *forward selection* [Guyon and Elisseeff 2003] on all remaining candidate inputs. Specifically, start with no inputs. Then loop through all of the inputs, testing each to determine which one provides the most improvement in mean-squared-error.

The winning input is selected for use. Iterate until no additional inputs can be added that decrease the mean-squared-error.

4. (*Optional*) — Perform principal component analysis to project the selected inputs onto a manifold of lower dimensionality.

The reason we list PCA as optional is because we have found it is not useful if the programmer has supplied a set of good features. This is because the features may represent non-redundant information.

Our approach is interesting because we first perform a fast and simple rejection test using $\mathscr{R}(i)^2$, which seeds and expedites the more complex forward selection algorithm. This also helps make selection more robust by initially rejecting inputs with no clear statistical correlation to the output.

Our custom input selection method is a combination of existing techniques: correlation, forward selection, and PCA. This combined approach is effective because we leverage the strengths of each technique, while side-stepping many of their weaknesses.

So that the Pearson correlation $\mathscr{R}(i)^2$ can be computed quickly, we use the approximation detailed in [Guyon and Elisseeff 2003].

3.5 Summary and Discussion

We achieved our best results by performing regression with k-nn, using $k = 3$. This is because local regression makes it simple to accurately approximate any given behavioral/cognitive model, as long as enough state-action mapping examples can be gathered to cover the state space. While k-nn is slower to execute than compact techniques (like NN), it can still usually be computed in under 20 microseconds on a 1.7 GHz PC. Because its execution is near fixed-time (using a balanced kd-tree for case lookup), and it generalizes using known behavior examples, our model approximation technique with k-nn provides a solution to the two problems listed in the introduction. Note that novel paths through the state space are possible, and thus novel behavior sequences, but no immediate behavior except blending of local cases is possible.

However, we did find two circumstances under which compact regression techniques were more useful. First, if there are few state-action examples of the desired behavior compared to the input dimensionality, the state space may not be adequately populated with examples to use k-nn. Second, if there is notable noise in the state-action examples, it can cause high-frequency dithering in an animation when using k-nn. In contrast, with a compact technique like NN or SVM, such noise is usually averaged out during training.

An interesting benefit of our regression technique is that, since the decision-making examples are generated off-line, they can be of very high quality. In other words, the character has a lot of CPU time with which to make its decisions. As a result, our technique allows a character to exhibit significant intelligence on-line with the use of little CPU.

Our regression technique does have some weaknesses which are important to discuss. First, due to generalization of state-action examples, it is difficult or impossible to guarantee that a character will never generalize cases in such a way that unrealistic behavior is the result. Nevertheless, this weakness has not proven a significant problem in our case studies. Second, our technique cannot be used to approximate any behavioral/cognitive model because of the curse of dimensionality. Behaviors that require a large amount of state information are not good candidates for our technique.

Our custom input selection method greatly simplifies for the programmer the process of machine learning a behavioral/cognitive model. While our input selection method does not create features (an open problem), it does quickly and accurately select a minimum set of inputs from a superset, in a way oriented toward the requirements of regression of behavioral/cognitive models. Although our method is suboptimal (like most input selection techniques), it has performed nearly optimally in our experiments.

Part III

Online Adaptation for Interactive Characters

This part addresses Problem #2 listed in Chapter 1: lack of online adaptation for autonomous virtual characters.

Chapter 4 introduces a technique for incremental action prediction. Specifically, the character records observations of the behavior of the human user. A model is created from these observations. While learning is taking place, this model is used to predict the future behavior of the user, allowing the character to adaptively choose actions to perform. This chapter was published in the journal *Computational Intelligence* and can be referenced as follows.

> Jonathan Dinerstein, Dan Ventura, and Parris K. Egbert. "Incremental action prediction for interactive autonomous agents". *Computational Intelligence*, **21**(1):90–110, 2005.

Chapter 5 builds off of the results given in Chapter 4, presenting a multi-level adaptation technique. Each layer is composed of a separate learning method. These learning methods influence (from low to high level) the character's action selection, task selection, and goal selection. An imitation method is also presented whereby the character can imitate novel behavior performed by the human user. Chapter 5 has been accepted for publication in *ACM Transactions on Graphics*. It can be referenced as follows.

> Jonathan Dinerstein and Parris K. Egbert. "Fast multi-level adaptation for interactive autonomous characters". *ACM Transactions on Graphics*, **24**(2), 2005.

Chapter 4

Fast and Robust Incremental Action Prediction for Interactive Agents

Computational Intelligence, Vol. 21, No. 1, pp. 90–110, 2005.

Abstract: The ability for a given agent to adapt on-line to better interact with another agent is a difficult and important problem. This problem becomes even more difficult when the agent to interact with is a human, since humans learn quickly and behave non-deterministically. In this paper we present a novel method whereby an agent can incrementally learn to predict the actions of another agent (even a human), and thereby can learn to better interact with that agent. We take a case-based approach, where the behavior of the other agent is learned in the form of state-action pairs. We generalize these cases either through continuous k-nearest neighbor, or a modified bounded minimax search. Through our case studies, our technique is empirically shown to require little storage, learn very quickly, and be fast and robust in practice. It can accurately predict actions several steps into the future. Our case studies include interactive virtual environments involving mixtures of synthetic agents and humans, with cooperative and/or competitive relationships.

Keywords: autonomous agents, user modeling, agent modeling, action prediction, plan recognition.

4.1 Introduction

The use of intelligent software agents is becoming increasingly pervasive. This is especially true in interactive software, where one or more human users may interact with the system at any time. Examples of such agents include training simulator and computer game characters, virtual tutors, etc. However, an important limitation of most agents in interactive software is that they are usually static. In other words, the agent's behavior does not adapt on-line in response to interaction with a human user.

Effective and rapid on-line adaptation of an agent's behavior would be extremely useful for many applications. For example, consider a training simulator where a virtual character is an opponent to a human user (see Figure 4.1). By learning on-line through interaction with the human, the character could adjust its tactics according to those of the human it is competing against and thereby become a more difficult, customized opponent.

One primary reason for the lack of use of on-line learning for interactive agents is that learning from and about a human is difficult, due to non-deterministic behavior and a non-stationary policy. Also, humans learn quickly and are impatient, so an agent must also learn quickly to provide a stimulating and effective experience for the user. These issues are discussed in more detail in Section 4.3.

In this paper, we present a novel machine learning method for fast adaptation of interactive agents. Specifically, our technique allows an agent to learn to predict the future actions of a human user (or another synthetic agent). We take an observation-based approach which operates through case-based reasoning. As the interaction proceeds, the agent records state-action pairs of the human's behavior. Each state-action is treated as a single case. The case library represents a non-deterministic mapping of states to actions. We generalize these state-action cases using either continuous k-nearest neighbor or a modified minimax search. These alternate approaches to generalization allow us to adjust a confidence/caution tradeoff as we see fit.

The agent can predict several steps into the future by iteratively predicting using the previously predicted state. That is, for each time step into the future, we predict each agent's next action; the predicted actions are then used to predict the next state, which is in turn used for the next iteration of action prediction. Finally, once the desired length of

prediction has been computed, the agent uses this information to help in selecting its own actions. Note that our technique can learn quickly, being case-based, and can naturally handle non-determinism in the behavior it is learning. Also, old cases are deleted so that it can learn non-stationary behavior. Further, our technique can be used for either cooperative or competitive relationships between an agent and human, since our learning technique provides the agent with very general information.

We begin by surveying related work. We then examine the issues involved in on-line learning for interactive agents and propose a set of requirements for such a learning method. We then present our technique for rapid adaptation of interactive agents through action prediction. Finally, we conclude with our experience from three case studies. Our first two case studies are based on a computer game/simulation of a sport like rugby or American football. We chose this interaction setting since athletics is interesting for studying cooperative and competitive behavior, and is known to be a difficult problem for machine learning [Stone 2000]. Our first case study is a simplified problem, whereas the second is significantly more complex. Our third case study is of Capture The Flag using the Gambots [Kaminka et al. 2002] test bed.

4.2 Related Work

Our case-based action prediction technique overlaps with many topics currently of interest in AI and machine learning, such as plan recognition, learning in multi-agent systems, and agent/user modeling. While our method is novel, it was inspired by previous work which we now review. Note that our method falls within the bounds of *fictitious play* theory [Stone and Veloso 1997].

The use of AI techniques in animating virtual characters has long been popular [Reynolds 1987]. By making a virtual character an autonomous agent, it can be self-animating. This topic includes all autonomous agents in interactive virtual environments, such as characters in training simulators and computer games. On-line learning for interactive virtual characters has only begun to be addressed [Evans 2002; Tomlinson and Blumberg 2002; Blumberg et al. 2002], and is currently limited to master-slave relationships and learning

high-level behavioral guidelines through immediate and direct instruction and feedback from the human user. Thus these techniques do not solve the problem we are addressing.

A paper that addresses many of the same concerns we do is [Isbell et al. 2001], where an agent in a text-based distributed computer game — a "MUD" — learns to pro-actively interact with the human users in a desirable manner. Users provide feedback on the agent's behavior, guiding its learning. The agent learns through a modified form of Q-learning, constructing value tables through linear function approximation. This previous work focuses on human-agent interaction in a social context, whereas we are interested in virtual environments where the human and agent interact in a physically oriented manner.

A well-known technique for learning in multi-agent systems is *minimax-Q* [Littman 1994]. This technique is a modification of Q-learning, designed for two agent zero-sum Markov games. While effective and formally sound, this technique does not address the problem of interest in this paper because minimax-Q requires too much time and too many experiences to learn (in a simple soccer environment, one million steps).

An interesting work performed in *action prediction* is [Laird 2001]. In this technique, the agent attempts to predict the human's future behavior by assuming the human will behave exactly like the agent would in the same situation. This interesting and simple technique has proven effective in practice for a complex virtual environment. However, the agent does not learn about the human's behavior, so it has no ability to adapt. Note that action prediction and *plan recognition* are often considered to be the same problem.

Agent modeling has been a popular approach to learning in multi-agent systems for some time [Bui et al. 1996; Vidal and Durfee 1997; Weiss 1999]. One such technique is *M** [Carmel and Markovitch 1996a,b], a generalization of minimax. Rather than assuming the opponent will always choose the optimal action, the opponent's observed behavior is modeled using a neural net. Thus, given a state as an input, the network produces a deterministic prediction of the opponent's next action. While interesting, this approach is not useful for our needs, since the neural net may require too much time and too many examples to learn. Also, the neural net will produce deterministic predictions (i.e., will average conflicting *state* $\rightarrow$ *action* examples), and thus is likely to make notable mistakes with regards to human behavior which is often highly non-deterministic.

Another modeling-based technique is *Maximum Expected Utility* (MEU) [Sen and Arora 1997], also a modification of minimax. The opponent is modeled such that, for any given state, each candidate action has a probability of being selected. Thus, the expected utility of an action performed by the learning agent can be computed as the weighted sum of the rewards with respect to all the actions the opponent can select (where the weights are the probabilities). Thus, unlike M^*, this technique does not ignore worst-case scenarios and does not produce deterministic predictions. However, while this is an interesting approach, it can require massive amounts of storage, is infeasible for continuous state and/or action spaces, and the probabilities can take a long time to learn. Also, if we wish to determine long-term utility (predict far into the future), our search will likely be intractable.

In [Tran and Cohen 2002], an agent models the reputation of sellers in an electronic marketplace to more optimally engage in trade. This technique, while interesting and useful, does not address the problem of interest in this paper since it only models a high-level aspect of the other agents, rather than their actual behaviors.

One of the most well known agent modeling techniques is the *Recursive Modeling Method* (RMM) [Gmytrasiewicz and Durfee 2000, 2001]. In RMM, the fact that an agent knows that the other agent is learning about it is modeled explicitly. Specifically, an agent models the other agent, but then must also model how the other agent will change due to the fact that the first agent has learned. This process can continue for some time, while each agent continues to learn models of the current behavior of the other. Thus each agent contains a recursive set of models, up to a pre-specified depth. In [Gmytrasiewicz and Durfee 2000, 2001], RMM is implemented using a tree of payoff matrices. Because of this, storage and computational requirements increase exponentially, and therefore this technique may not be useful for many practical settings. Also, RMM is limited to discrete state and action spaces. Some initial work has been performed in flattening RMM into a single level learning technique [Hu and Wellman 1998], but this problem is still unresolved.

Recent work [Rovatsos et al. 2003] in agent modeling has examined *open systems*: multi-agent systems where agents enter/leave quite often, and specific agents often never return to the system. To cope with these issues, agents are classified and then each class is

modeled. This is a general enough concept that it could be used with most agent modeling techniques.

Recently, *Case-Based Plan Recognition* (CBPR) [Kerkez and Cox 2003] was introduced. Unlike most previous plan recognition methods, CBPR does not require that a plan library be constructed *a priori*. State-action cases of an agent's observed behavior are recorded and then used to predict future behavior. The state-actions are expressed in first-order logic and are generalized by finding the single case whose associated state most closely matches the query state. However, while CBPR has proven to be interesting, it has some weaknesses that make it inappropriate as a solution for the problem of interest in this paper. Most notably, CBPR is only about 10% accurate in practice for toy problems, and can only predict one time step into the future (prediction chaining is ineffectual due to low accuracy). Further, CBPR makes deterministic predictions, and it cannot handle continuous state and action spaces (which many interactive environments, such as training simulators, have).

Closely related to agent modeling is *user modeling* [Zhu et al. 2003]. A human user is modeled so that software or an agent can more optimally serve him or her. This is closely related to the problem of interest in this paper. However, most existing techniques in this domain are designed for user interface adaptation, and thus model the user in a very specific/limited manner. In this paper, we are interested in autonomous agents that proactively interact with a user (according to their own goals, cooperative or competitive), and also engage in activities entirely independent from the user.

Our method is also somewhat related to agent *programming by demonstration* [Mataric 2000; Nicolescu 2003], where an agent learns how to behave from human demonstrations. In these techniques, a model is created of the human's behavior for the purpose of reproducing it to control the agent. Similarly, our method is somewhat related to *imitation learning* [Price 2002], where an agent learns by observing and imitating other agents (synthetic and/or biological). However, these techniques do not address the problem of interest in this paper.

There is need for a technique that allows an interactive agent to rapidly learn on-line to better interact with a unique human user. As discussed in Section 4.3, this adaptation

should be robust, require no explicit feedback from the human user, and learn fast enough to not tax human patience. Our contribution is a novel machine learning method that fulfills these needs.

4.3 The Interactive Agent Learning Problem

As mentioned previously, on-line learning for agents that interact in real-time with a human is a difficult problem. We have identified several requirements for an effective interactive agent learning technique, which are detailed below.

Requirements for an Effective Interactive Agent Learning Technique:

1. *Fast learning.* Human time and patience are scarce resources. Further, slow learning will not make an agent a more difficult opponent, or effective teammate, etc. Therefore, learning must occur quickly based on few experiences.

2. *Learn in the presence of non-deterministic and non-stable behavior.* In other words, the learning technique must be effective when the agent is interacting with a human.

3. *No explicit human feedback.* Requiring the human to provide feedback may be unnatural for many types of interactions, and may interrupt the flow of the interaction. Therefore, no explicit human feedback should be required.

4. *Must perform at interactive rates on a PC.* For the learning technique to be as widely useful as possible, it must be practical for interactive use on current PC CPUs (e.g., 2 GHz). Further, use of storage must be reasonable.

5. *Support both cooperative and competitive relationships.* While this is not truly a requirement, it is a desirable property so that the learning technique will be as broadly useful as possible.

4.4 Technique Description

We fulfill the requirements listed in Section 4.3 by using a combination of observation- and case-based approaches to construct a case-based model of the human's behavior. This

model is then used to predict the human's future actions, allowing the agent to predict the utility of its own candidate actions.

Our technique is summarized in Figure 4.2a. As the interaction proceeds, the agent records state-action pairs of the human's behavior. Each state-action is treated as a single case. This case library models the human's decision making, representing a non-deterministic mapping of states to actions. We generalize these state-action cases using either continuous k-nearest neighbor or a bounded minimax search. These alternate approaches to generalization allow us to adjust a confidence/caution tradeoff as we see fit.

As a formal overview, the use and execution of our adaptation technique involves the following steps:

1. Formulate a compact internal representation of the state space, S, and a logically structured internal representation of the action space, A.

2. Define a state-action case library: $L = \{(\mathbf{s},\mathbf{a})_i\}$. Partition the library into regions for fast case lookup: $P_S = \{r_i\}$ such that $r_i \cap r_j = \emptyset \leftrightarrow i \neq j, \cup_i r_i = S$. Initialize the library with state-action cases of generic behavior. Set time, $t := 0$.

3. As the interaction proceeds, observe and record state-action pairs. First, at time t, get $(\mathbf{s}_t,\mathbf{a}_t)$. Next, determine r_j such that $(\mathbf{s}_t,\mathbf{a}_t) \in r_j$. Finally, replace $\arg\min_{(\mathbf{s},\mathbf{a})_i}(M((\mathbf{s},\mathbf{a})_i))$ with $(\mathbf{s}_t,\mathbf{a}_t)$, where M is a case usefulness metric with positive scalar weights α and β: $M((\mathbf{s},\mathbf{a})_i) = -\alpha \cdot \text{age} + \beta \cdot d((\mathbf{s},\mathbf{a})_i,(\mathbf{s}_t,\mathbf{a}_t))$. d is an appropriate distance metric. "age" is the number of time steps for which case i has existed in L.

4. Predict the human's (or other agent's) future behavior by generalizing state-action cases: $\sim \mathbf{a}^H_{t+1} = f(\mathbf{s}_t, L)$. After predicting, the agent uses this information in selecting its own action to perform: $\mathbf{a}_{t+1} = g(\mathbf{s}_t, \sim \mathbf{a}^H_{t+1})$. Increment current time, $t := t+1$.

5. Repeat steps 3–5 for the duration of the interaction.

Note that steps 1 and 2 are performed by the designer/programmer in preparation for execution.

It is important that we predict the human's actions several steps into the future, so that our agent can maximize its utility based on non-local information. As shown in Figure 4.2b,

we do this iteratively by predicting using the previously predicted state. At each iteration, we either compute, predict, or assume the actions of all agents in the environment. The simulated actions of the predicting agent are either assumed or computed without the aid of prediction to avoid branching or recursion (although we discuss in Section 4.4.4 how branching can be used if desired to increase prediction accuracy). In our case studies, we have found that predicting between 5 to 15 steps into the future works well, is accurate enough to be useful, and is tractable. Finally, once the desired length of prediction has been computed, the agent uses the prediction in selecting its own actions, whether it intends to cooperate with or compete against the human.

Our goals for our learning method are somewhat different than in traditional machine learning. Where most traditional techniques are designed to learn *well* from scratch, our technique is designed to learn *fast* to adapt an agent to better interact with a unique human user. As a result, we rely somewhat on prior knowledge. Specifically, we assume that the programmer provides a reasonably compact state definition, and (if using minimax case generalization) a gradient fitness function.

4.4.1 State and Action Representations

Most interactive environments involving both software agents and human users are *virtual environments*: synthetic digital worlds that define the roles and possible actions of the human users and agents. Some of these virtual environments are presented to the user visually through computer graphics (e.g., training simulators, computer games, etc). In our case studies in this paper, we focus on this sort of environment. Thus the current state is a snapshot of the configuration of the virtual world. One benefit of this sort of environment is that sensing the current state is easy. However, our learning technique is not limited to this subset of interactive environments. As long as the current state and the behavior of the other agent (which we are modeling) can be perceived, our learning technique is applicable.

For any given agent and environment, the state space may have to be continuous. This is because, in a stimulating environment where the agent and human are competing or cooperating intimately, it is possible that a small difference in state can determine a large difference in behavior. A continuous state space can also help in achieving a realistic virtual

environment. For example, a discrete state space seems very unnatural for a car driving training simulator. Therefore, our technique uses a continuous internal representation of states (and actions). Since this is a very general representation, all continuous and most discrete state spaces are supported.

We represent the state space S as a real-valued, n-dimensional feature vector. Thus $S \subset \mathbb{R}^n$, and a state $\mathbf{s} \in \mathbb{R}^n$ is a point within this feature space. These features can be specified manually, or automatically discovered through principal component analysis. We assume that the designer has by some means provided a good, compact state representation for the given agent and environment. In other words, we assume that the state space dimensionality n is small. This is important because a compact state space will help the agent adapt quickly and better generalize what it has learned. However, our technique still operates successfully with a non-compact state representation, though learning may not be as fast as desired. As an example of a compact and effective state representation, in our 1-on-1 simplified rugby case study, the state is simply the translation-invariant separation of the agent and the human's avatar.

The state definition should be organized in such a way that states that are similar (according to the Euclidean metric) are usually associated with similar or identical actions. This makes generalization possible.

Like states, actions are internally represented by real-valued vectors, $\mathbf{a} \in \mathbb{R}^m$, so that both discrete and continuous action spaces are supported. If possible, the action space should be defined in such a way that actions can be combined into some sort of "intermediate" action (e.g., a 'left' vector $[-1,0]$ and a 'forward' vector $[0,1]$ become a 'diagonal' vector $[-1/\sqrt{2}, 1/\sqrt{2}]$). More formally, it is useful if the actions are organized such that they can be blended into valid intermediate actions using valid binary operators, for example, $\mathbf{a}' = \mathbf{a}_1 \triangle \mathbf{a}_2$ (for some operator $\triangle$). As we detail shortly, blending is required for case generalization through continuous k-nearest neighbor. However, blending is not performed with generalization through minimax, and thus is not strictly necessary.

4.4.2 Learning State-Action Cases

The observed state-action pairs of the human's behavior are recorded on-line. That is, at each time step, the current state and the action selected by the human are saved. For simplicity, we use a small constant time step for sampling the human's behavior. For example, in our complex rugby case study, the time step matches the frame rate of the computer graphics animation (15 Hz). These cases represent a discrete sampling of a Markovian model of the human's behavior. While many cognitive scientists postulate that human behavior is non-Markovian [Nadel 2003], Markovian models have often proven effective in the agent/user modeling literature (for more information on this topic see [Carberry 2001; Kerkez and Cox 2003]).

For our method to be entirely accurate, it would be necessary to record any salient features of the human's internal state as part of the current state in each state-action pair. However, the human's internal state is likely inaccessible. Thus our compact state space may be incomplete. Nevertheless, an approximate state space is acceptable; as we empirically show in the experimental results section, the accuracy of our technique is sufficient to produce effective agent behavior.

Each recorded state-action pair is treated as a case in a case-based reasoning engine. A library of useful cases is maintained. Since the state space is continuous, the library is organized as a (possibly hierarchical) partitioning of the state space to facilitate fast lookup of cases. Automatic partitioning techniques (e.g., a kd-tree) can be used to great effect, or partitioning can be performed by the programmer so that human insight may be applied. The library and its partitioning are defined more formally in Part 2 of the overview given in Section 4.4.

The case library is originally populated with state-action examples of "generic" human behavior. These are gradually replaced with user-specific examples, as they are observed by the character. In particular, a limited number of cases are allowed for each region of the state space, and (nearly) duplicate cases are not allowed. Cases are selected for replacement based on their age and unimportance. In other words, if a case was recorded long ago, and/or is very similar to the new case we are adding (in both the state and action), it is likely to be removed. Thus the character has the ability to "forget", which is

very important in learning something as non-stationary as human behavior. Also, since any given region of the state space will always contain the same number of cases, there is no need to repartition on-line. This can limit the agent's ability to learn detailed information in unanticipated regions of the state space, but allows for simple and fast maintenance of the case library. The case replacement procedure is defined more formally in Part 3 of the overview given in Section 4.4. In our implementation, we use the Euclidean metric for computing the distance between two cases. Specifically, the similarity metric is: $d((\mathbf{s},\mathbf{a})_i,(\mathbf{s},\mathbf{a})_j) = \|\mathbf{s}_i - \mathbf{s}_j\| + \|\mathbf{a}_i - \mathbf{a}_j\|$.

4.4.3 Generalization of Cases

To predict the human's future behavior, the library of cases must be generalized so that, for any given query state, an associated action can be computed. Note that an important question about how to generalize is whether to focus on confidence or caution. For example, if we are confident in our acquired knowledge, we can attempt to strongly exploit our knowledge by assuming that the human will behave as he usually has done when in/near the current state. However, this will ignore unusual behaviors, even worst-case scenarios. Alternatively, we can be cautious and assume that the human will choose the worst-case action (from the agent's perspective) from those actions he has previously chosen in/near the current state. Both approaches have their merits, and we have developed generalization techniques for both.

To focus on exploitation (i.e., exercise confidence in our acquired knowledge), we use the *continuous* k-*nearest neighbor* algorithm, as shown in Figure 4.3a. That is, the k cases closest to the query state (according to the Euclidean metric) are found and a distance-weighted normalized sum of the associated actions is computed:

$$\sim \mathbf{a} = \frac{\sum_{i=1}^{k}(w_i \cdot \mathbf{a}_i)}{\sum_{i=1}^{k} w_i}, \quad \text{where } w_i = \frac{1}{d_i^2}.$$

In our case studies we have found $1 \leq k \leq 3$ is effective. $k = 1$ is good for exactness, as no blending of actions occurs. However, $k = 3$ is good if there is no closely matching case, and/or for attaining a more general impression of the human's behavior. Therefore, it is useful to vary k depending on the nearness of the closest case, and/or the required

specificity of a prediction. Also, it is helpful to normalize the axes of the state space, so that they will contribute equivalently in computing distance.

To focus on caution, we use a bounded minimax search, as shown in Figure 4.4. The k cases closest to the query state (according to the Euclidean metric) are found. Then, the utility of the k actions associated with the retrieved cases is computed using the agent's own fitness function. Finally, the action that results in the minimum fitness for the agent is assumed to be the one the human will select. More formally:

$$\sim \mathbf{a} = \arg \min_{\mathbf{a}_i \in k \text{ neighbors}} (\text{fit}(\mathbf{s}, \mathbf{a}_i)).$$

What is unique about our modified minimax search is that we do not consider all possible actions. Rather, we consider only those actions we have previously seen the human perform in that region of the state space. This not only makes minimax tractable for large or continuous state and action spaces, but also still allows us to exploit our knowledge of the human at the same time as we focus on caution. In addition, note that the information that must be learned for our minimax approach can be learned very rapidly, since we only need to observe some representative cases of the human's preferred behavior.

When $k = 1$, the minimax approach degenerates to 1-nearest neighbor. That is, it predicts that the human will do exactly what was done previously in the closest matching case. However, as we increase k, this technique becomes more cautious. In fact, in the limit as $k \rightarrow \infty$, our bounded minimax becomes a standard minimax (assuming that we have cases to cover all possible actions). In our case studies, we have found that small values of k (e.g., ≤ 3) are useful when we want to be more cautious than in k-nearest neighbor, yet still wish to strongly exploit our knowledge. Alternatively, we have found that larger values of k (e.g., $5 \leq k \leq 16$) are useful for being extremely cautious.

Only our minimax approach to generalization requires a fitness function; k-nearest neighbor does not. Further, it must be a *gradient* fitness function. In other words, informative feedback should be given for all states, with fitness leading toward goal states. However, gradient fitness functions are well known, and have been used to great effect in reinforcement learning for speeding up convergence [Sutton and Barto 1998]. An interesting point we discovered in our experiments is that, sometimes, it is better to use a fitness

function that directly measures the human's fitness, rather than the agent's. Using such a fitness function, minimax would seek to maximize rather than minimize fitness. We have found this approach to be preferable if the human's and agent's goals are not necessarily exact opposites.

While generalization through k-NN blends actions (for $k > 1$), minimax does not. As a result, our action prediction method is still applicable for agents with non-blendable actions, though generalization must be performed through minimax (or k-NN with $k = 1$).

Generalization of cases is introduced formally in Part 4 of the overview given in Section 4.4. We use k-NN or bounded minimax for the prediction function f discussed in the overview.

4.4.4 Using Case-Based Action Prediction in Practice

Our learning technique can be used by agents that cooperate with or compete against the human user. This broad applicability is possible because our technique provides an agent with very general information (predictions of the human's behavior). How to use this information is up to the agent. Even our minimax approach to generalization can be used for cooperation, by seeking to maximize the human's fitness function, or assuming the human will seek to maximize the agent's own fitness function. In other words, the agent and human are trying to maximize a shared fitness function.

The accuracy of the learning in our technique has proven to be very promising (see the figures and tables in Section 4.5). It also has a small memory footprint (usually ≤ 2 MB per agent). Also, the performance of our technique is well within interactive speeds (see Table 4.3). If desired, there are ways to further speed up our technique at the cost of accuracy. These include the following:

- Predict actions for only a subsample of time steps into the future, holding constant in between (e.g., predict a new action once every 4 time steps).
- Rather than predicting, assume constant actions for some agents (e.g., null or last observed action).

- Ignore agents that will likely have no effect on the current decision making of this agent (e.g., only predict actions for those other agents that are spatially close to this agent in the environment).

Note that the function of our action-prediction technique is to supply an agent with supplementary information. Therefore, the way this information is used can be unique for any given agent. For example, given an n-step prediction of the human's actions into the future, the agent could perform an informed tree search to plan its own actions through deliberation. Alternatively, this n-step prediction could be used as extra inputs into a reactive action-selection method, even a black box implementation.

An alterative way to use action prediction (rather than predicting an entire chain of actions given an initial state, as in Figures 4.3b and 4.4) is for the agent to request individual predictions for certain states. This can be especially useful for agents that perform decision making through deliberation with a tree search, as the agent can request information specific to any state it encounters while deliberating. However, this requires more CPU than a single linear prediction, and it has not proven to be significantly more accurate in our case studies.

Our action prediction technique is not limited to small environments with only one human. Indeed, it may be appropriate for very complex environments of many agents (more than one human user, etc). However, for adaptation to perform well, the state space definition must always be reasonably compact. This circumvents the curse of dimensionality, thereby allowing our adaptation technique to be used for interesting, difficult problems.

To further counteract the curse of dimensionality, we have found it useful to modularize the action prediction where possible. For example, consider an agent who can perform two independent actions simultaneously (e.g., walk in a given direction while looking at something else). We can split this into two separate problems, with the adaptation for each performed separately. This can help simplify both the state and action spaces. It may also be useful to split up the state space into uncorrelated or independent regions.

Recall that the state-action model of the human's behavior is initialized to "generic" human behavior. This generic information is then gradually forgotten as new user-specific

cases are gathered. Specifically, we replace all generic cases before replacing any user-specific cases. Also, for generalization through k-nearest neighbor, we more heavily weight user-specific cases over any remaining generic cases.

It is possible to share all acquired knowledge between all adapting agents in a given environment. In other words, we can use a single repository for acquired knowledge, which all adapting agents share. This is useful for reducing storage requirements, as well as allowing every agent to behave optimally according to what has been learned. However, this will not be plausible for agents in all categories of multi-agent systems, especially physical agents with limited communication abilities.

We have presented two separate generalization techniques, one focused on confidence and the other on caution. This tradeoff can be further adjusted by varying k, as shown in the results section. Because this tradeoff represents the "personality" of the agent, it is not possible to formally dictate which approach is best. Indeed, as we show in the results section of this paper, both techniques have unique benefits from a fitness perspective. Therefore, the choice of how to generalize is left up to the agent designer. Indeed, this could even be altered dynamically for an agent, depending on the most important goal of the agent at the moment and/or what strategy is proving most effective. Moreover, each technique has unique requirements: if action blending is not possible, minimax must be used; if a gradient fitness function is not available, k-nn must be used.

4.5 Experimental Results

We now present our case studies in detail. In all of our case studies, we used the L_2-norm for our distance metric d. Specifically, $d((\mathbf{s},\mathbf{a})_i,(\mathbf{s},\mathbf{a})_j) = \|\mathbf{s}_i - \mathbf{s}_j\| + \|\mathbf{a}_i - \mathbf{a}_j\|$. Also, we tuned the parameters of the case usefulness metric M (α and β) such that difference between cases was more important that the amount of time a given case had existed in the library. For each adapting agent, the action selection function g was composed of an A* [Russell and Norvig 2003] implementation and our action prediction technique. Specifically, at each time step, A* was used to compute a new plan with respect to the human's or opponent's predicted actions. The agent then performed only the first action in that plan.

The reason we continually reformulated plans was to avoid poor agent behavior due to incorrect action predictions.

4.5.1 Simplified Rugby Case Study

Our first case study involves two software agents engaging in a sport such as rugby or American football (see Figure 4.5). In this game, the "ball runner" agent attempts to score by running with the ball to a designated end of the playing field. The "tackler" agent attempts to stop the ball runner by tackling him. The environment is discrete, divided into squares. Each player can be in only one square at a time. If the two agents move to the same square, a tackle (end of game) occurs.

The game field is 9 squares wide (movement outside of this boundary is not allowed), with a scoring zone at the top of the field. The possible actions available to the software agents are moving to an adjacent square (including diagonally), or staying in place. Thus there are nine total possible actions. As the game proceeds according to a fixed discrete time step, the agents act simultaneously once per time step. There is no explicit communication between the agents. The current state is fully perceivable to the agents within the bounds of *sensory honesty*: even though the agents' sensors are virtual, they are forced to perform realistically like physical sensors — see [Isla et al. 2001] for a detailed discussion. As a brief example, an agent is not allowed to see through the back of its head, or through solid objects in the environment.

For the learning technique, the compact state definition is simply the translation-invariant separation of the two agents. Therefore, it is a two-dimensional vector of integer values. An action is stored as a two-dimensional integer velocity vector. Thus k-nearest neighbor can blend several actions into a valid action. After predicting an action with k-NN, its components are in floating point; we round to get integer components.

At the start of each game, the agents are placed in the center of the field, one square apart, as shown in Figure 4.5. The ball runner has the ball, and attempts to score by running past the tackler to the far end of the field. The tackler agent performs its decision making through A* [Russell and Norvig 2003], whereas for the purpose of experimentation the behavior of the ball runner is exhaustively enumerated. Moreover, only the tackler adapts.

	$k=1$	$k=2$	$k=6$	$k=12$
k-*NN*	69.16 : 1	25.61 : 1	23.69 : 1	29.825 : 1
Minimax	69.16 : 1	239.5 : 1	963.8 : 1	1729 : 1

Table 4.1: Average ratio of tackles to scores for all behaviors of length 7. With no learning, the ratio was only 5.54 : 1.

	$k=1$	$k=2$	$k=6$	$k=12$
k-*NN*	-0.0553	-0.0625	-0.0629	-0.0444
Minimax	-0.0553	0.00957	0.0314	0.0325

Table 4.2: Average forward progress made by ball runner before end of game for all behaviors of length 7.

If prediction information is available, the tackler uses it in its deliberation. However, if prediction information is not available, it assumes the ball runner will continue to perform the last observed action. The tackler's fitness function simply directs it to get as close to the ball runner as possible, without letting the ball runner pass it by (and thereby be able to score). This fitness function is given in pseudo-code in Figure 4.6.

We performed several experiments in this case study, varying k and the generalization technique used. Note that in all these experiments, we forced the ball runner to exhaustively try all behaviors of length seven (i.e., a game lasting seven time steps). Each behavior was presented to the tackler three times; the first two times to learn, and then on the third iteration we measured the performance of the tackler. We gathered statistics on the effectiveness of the tackler's learning, with regards to ratio of tackles vs. scores, and to average forward progress made. These results are presented in Tables 4.1 and 4.2.

With no learning for the agent (i.e., no observed state-action cases added to the agent's action prediction library, just the default cases), the ratio of tackles to scores is only 5.54:1. This is significantly lower than the ratios achieved by using our learning technique.

Note that for $k = 1$, the experimental results are equal for both generalization techniques, because the two techniques are equivalent when only using a single case. Also note that k-NN effectively holds the ball runner to negative forward progress, whereas our bounded minimax allows positive forward progress for $k > 1$. This is because, while k-NN

	Simplified Rugby	Virtual Rugby
Action selection freq.	1 Hz	15 Hz
Action prediction time	21 μ sec	30 μ sec
Total avg. CPU usage	0.12%	6.82%
Memory usage	$\ll$ 1 MB	$\leq$ 2 MB

Table 4.3: Typical performance results of action prediction in our two rugby case studies (for one adapting agent). Case learning time is negligible, and therefore is not listed. We used a 1.7 GHz PC with 512 MB RAM.

strongly exploits (especially for smaller values of k), minimax is more conservative. However, minimax allows the human to score less often than k-NN, since it more effectively covers worst cases. As can be seen in the tables, minimax becomes more conservative and cautious as k increases. k-NN can be less effective for larger k since it produces a deterministic prediction (i.e., averages conflicting actions). However, k-NN with $k > 1$ can be useful for producing stable predictions over time, since it averages local cases.

Performance numbers for our technique (for both case studies) are given in Table 4.3. *Simplified Rugby* denotes this case study and *Virtual Rugby* the case study described next (Section 4.5.2). We also performed experiments on the effect of clearing vs. not clearing the case library between behaviors — this proved to have little effect.

4.5.2 Complex Virtual Rugby Case Study

Our second case study is similar to the first, but is significantly more complex and involves a human user. In the simplest configuration, there are two players. The human has the role of ball runner, while the agent is the tackler. However, in our experiments we varied the roles and number of members of each team, as we describe shortly.

The virtual environment is continuous, both in terms of the state and action space. The agent and the human's avatar can be located at any real-valued position on the playing field, and can be traveling at any velocity in the range [0.0, *MaxVelocity*] along a real-valued two-dimensional vector. The agent can change each component of its velocity by values in the range [$-$*MaxAcceleration*, *MaxAcceleration*] once per time step. The human controls

his avatar through a joystick, and has no software enforced acceleration limit. Therefore, the human has a distinct advantage in terms of maneuverability.

For this environment, the compact state definition used in learning is composed of: (1) the translation-invariant separation of the agent and human, and (2) the velocity vectors of the agent and human. Thus the compact state has six real-valued components. The action is stored as a two-dimensional real-valued acceleration vector.

As in the simplified rugby case study, there is no explicit communication between the agent and human, and the agent can fully perceive the current state of the virtual environment within the bounds of sensory honesty. The current state is presented to the human user in real time through computer graphics, as shown in Figure 4.1. A fixed time step of 15 Hz is used in this case study, which is fast enough to make learning challenging and forces our learning technique to operate quickly.

We performed several experiments in this case study, varying the number of agents on each team, the initial state, and the human user's behavior. We gathered statistics on the accuracy of the agent's learning, the increase in its success rate with respect to the human user (i.e., ratio of tackles to scores), and the runtime performance of our adaptation system. These results are presented in Figures 4.7 and 4.8, and Table 4.3. Note that in the graphs, the experiments started with the agent having very incorrect information about the human user. We purposely did this to demonstrate how quickly our technique learns, especially when its current knowledge is invalid.

As seen in Figure 4.7, our learning technique reaches high accuracy quickly, and then continues to learn such that it remains accurate for non-stationary human behavior. Also, as seen in Figure 4.8, the agent eventually beats the human more than 50% of the time, even though the human's acceleration is not bounded but the agent's is. An interesting result from our experiments is that our learning technique works well for multi-agent environments. For example, we performed an experiment where several agents (all on one team) were supposed to work together to tackle the human user. This experiment was successful, as the agents could predict what their teammates would do and thereby were able to cooperate effectively (see Figure 4.9).

A frame-by-frame example of our learning technique in action is given in Figure 4.10.

Further details and examples are given in a supplementary digital video, available from http://rivit.cs.byu.edu/a3dg/publications.php.

4.5.3 Capture the Flag

This case study is based on a well-known multi-agent research test bed called *Gamebots* [Kaminka et al. 2002]. This test bed modifies the popular computer game *Unreal Tournament 2003*, allowing a programmer to replace the built-in agent AI. In Unreal Tournament, the virtual world is a complex 3D environment of rooms, hallways, stairs, etc. It is populated with two or more players (virtual humans) organized into two teams. Each player is controlled by a human user or software agent. The players are armed with "tag guns"; once a player is tagged, he is out for a period of time. The objective is to reach the other team's flag.

We have modified the Gamebots test bed so that, rather than overriding the entirety of an agent's standard AI, we only override the choice of combat movements (e.g., dodging and/or moving into an advantageous position to tag an opponent). Goal selection, environment navigation, etc, are performed in the standard manner by the agent. In our experiments, there was one human user and multiple software agents (approximately half of the agents on the human's team, the others on the opposing team).

Each agent uses action prediction to predict the human's 2D movement to better cooperate with or compete against him. Actions such as aiming the tag gun, firing, and jumping were not predicted.

Because of the complexity of this environment, it is not plausible to use a complete state space for action prediction. Thus we use an approximate state space. The state space is composed of: (1) the translation-invariant separation between the human user and the opponent closest to him; (2) the 2D velocities of the human and his closest opponent; and (3) an angle θ representing the average direction toward nearby obstacles. Thus the compact state definition is: $(\triangle x, \triangle y, \mathbf{V}^H, \mathbf{V}^A, \theta)$, where $\{\triangle x, \triangle y\}$ is the translation-invariant separation between the human and his closest opponent, and $\{\mathbf{V}^H, \mathbf{V}^A\}$ are the 2D velocities of the human and closest opponent respectively. All nearby obstacles are represented by a single mean angle, θ (oriented around the "up" direction), representing

the average direction toward the obstacles according to the human's frame of reference. Assuming the user's avatar will never be in a very narrow hallway or room, this angle will be valid since the avatar will not be surrounded by obstacles in the environment.

The results of this case study are presented in Figure 4.11. Action prediction is of a lower accuracy than in our rugby case studies, largely because we use such a crude approximation of the current state. Nevertheless, our results are still promising, suggesting that our adaptation technique scales sufficiently to be useful for complex environments and agents. A slide show of this case study is given in Figure 4.12.

4.6 Summary and Discussion

We have presented a novel method that enables autonomous cooperate/competitive agents to quickly adapt on-line to better interact with a unique human user (or another synthetic agent). Our technique is very general, simply providing extra information to an agent, and thus can be used with many existing types of agents. Our system supports both reactive and deliberative decision making, in discrete or continuous state/action spaces. Our contribution in this paper is important because we present a solution for a previously unsolved problem: fast on-line learning for agents interacting with humans. Learning is an important problem for agents in many interactive software applications, such as training simulators, computer games, etc. For further examples than those given in this paper, see the supplementary digital video at http://rivit.cs.byu.edu/a3dg/publications.php.

The reason why our technique is sufficient for on-line agent learning to better interact with a unique human user is because the agent learns all the non-stationary knowledge it needs to intelligently select actions. For example, in our rugby case studies, the agent already has adequate perception, motor control, and decision-making skills. To optimally choose what actions to perform, it simply needs to know which of its candidate actions is most valuable for any given situation. By accurately predicting the behavior of the human, the agent can predict the results of its actions and thereby can rationally select what to do. Thus our knowledge-gathering approach to interactive on-line learning can be seen as hitting a "sweet spot" between nature vs. nurture.

It is because our technique forgets old cases that it is able to learn non-stationary behavior. Forgetting also helps keep memory requirements stable. However, forgetting can cause an agent to make a mistake if the human waits sufficiently long before repeating a behavior.

An interesting benefit of our technique is that, since an agent can adapt on-line, it can fill "gaps" in its decision-making skills. In other words, a programmer does not have to carefully construct the agent such that it will immediately handle every possible situation properly. This can also make an agent more robust. Further, in environments where there is no pareto-optimal Nash equilibrium (i.e., no universally best strategy), adaptation may be necessary to achieve and maintain good behavior.

However, while our technique has been shown empirically to work well, there are some weaknesses that are important to recognize. First, while knowledge gathering is very fast, using that knowledge does require a nontrivial amount of CPU time. As a result, it may not be plausible to have many adapting agents in the same interactive environment. Second, when generalizing cases, we make assumptions: most notably that the human will only perform an action he has previously performed in/near the current state. Thus the agent may ignore worst-case scenarios.

Although our technique has proven effective in our case studies, there is no guarantee that it will be effective for every imaginable agent and environment. However, as long as our assumptions in Section 4.4 are met, we believe that our technique will work well for many categories of agents and environments. In summary, we assume that a compact state representation and (if minimax is used for generalization) an effective gradient fitness function have been provided. Recall that organizing the action space such that actions can be blended is not strictly necessary because case generalization through minimax performs no blending. An example of an agent for which our technique will perform poorly is a chess player, because the state space is too large and complex.

Of course, our learning technique is not limited to learning about a human; it can be used for one synthetic agent to learn about another synthetic agent. The reason we have focused on agent-human interactions in this paper is because this type of learning is generally more difficult than in agent-agent interactions. Another possible use of our technique

is the creation of entirely new agent decision-making modules in an on-line fashion. We can use the state-action model of the human's behavior to perform decision making for an agent. This is fundamentally similar to *behavior capture* [van Lent and Laird 2001] and *behavior cloning* [Kasper et al. 2001]. One drawback to this proposed use of our method is that the decision making is somewhat shallow, since it only predicts actions (not task or goal selection).

For future work, we are interested in developing a more complete on-line agent adaptation system. Ever since the pioneering work of Brooks [Brooks 1986], agent AI has often been implemented in a layered fashion. Our action prediction technique is most pertinent for low-level decision making (e.g., action selection). However, higher-level decision making (e.g., task and goal selection) is also important. We want to develop new learning techniques for these higher layers. We envision these new learning techniques, along with our action prediction technique, composing a complete and effective on-line agent adaptation system. Another interesting avenue for future work may be to replace the case-based reasoning in our action prediction method with a more powerful and modern approach such as [Wilson and Martinez 2000]. Yet another direction for future work may be to apply our method for use in adaptive user interfaces [Zhu et al. 2003]. A further direction is to explicitly address partial observability of the state when recording state-action pairs of a human's behavior.

Figure 4.1: A virtual character (skeleton marked with a star) tries to catch the human's avatar (other skeleton). By adapting to the human's tactics, the character can become a more challenging, personalized opponent.

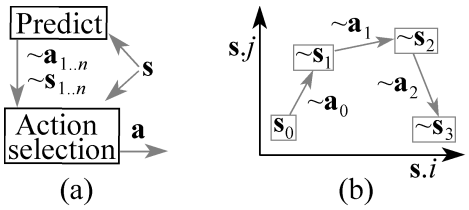

Figure 4.2: (a) Structure of our learning method. An n-step sequence of states and actions into the future is predicted, then the agent uses this information to make more optimal choices. (b) To predict the human's actions several steps into the future, we predict/compute the actions of all agents in the environment, determining a new state, and then repeat.

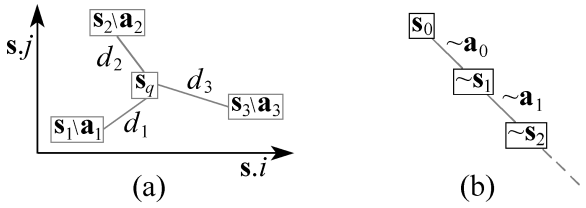

Figure 4.3: (a) To focus on exploitation (i.e., exercise confidence in our acquired knowledge), we use continuous k-nearest neighbor to generalize cases. (b) Since only one action is predicted for a given state, we predict linearly into the future.

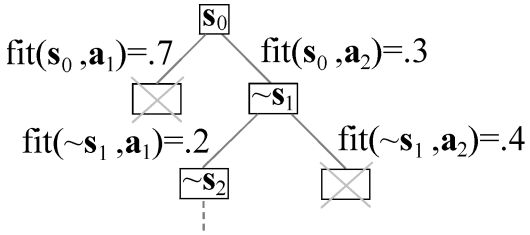

Figure 4.4: To focus more on caution than confidence, we use a bounded minimax search. The k nearest cases are found, and the associated action that minimizes the agent's fitness function is assumed to be the one the human will select. Since we only expand the minimum node, this technique predicts linearly like when we use k-NN.

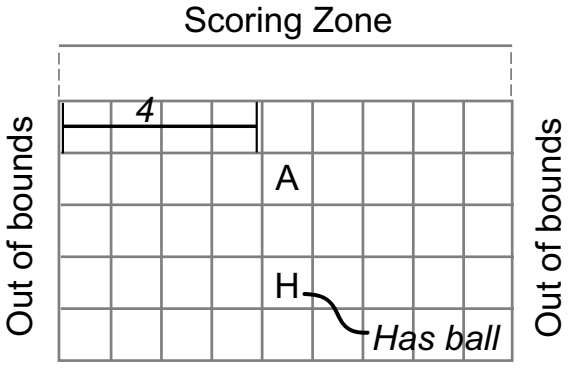

Figure 4.5: Environment used in the simplified rugby case study.

```
fitness_tackler(P^C, P^T)
{
    if (P^C.y > P^T.y)
    {
        /* Ball carrier has passed tackler, so he can score easily. */
        return (−100 − ||P^C − P^T||);
    }
    else if (P^C == P^T)
    {
        /* Close enough to tackle. */
        return (500);
    }
    else
    {
        /* Tackler is in front of ball carrier (where it should be). */
        return (100 − ||P^C − P^T||);
    }
}
```

Figure 4.6: Pseudo-code of the ball-carrier fitness function in the simplified rugby case study. $\mathbf{P}^C$ is the position of the ball carrier and $\mathbf{P}^T$ is the position of the tackler agent.

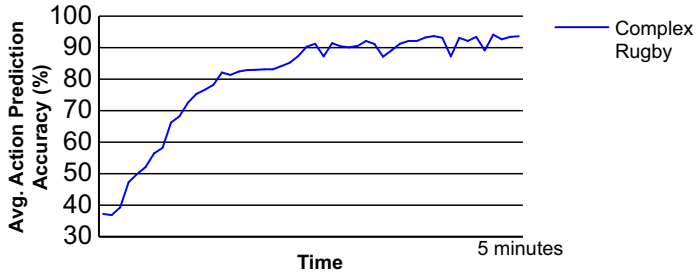

Figure 4.7: Accuracy of predicting the human's actions (L_2-norm) in the complex rugby case study. This experiment started with the agent having very incorrect information about the human user.

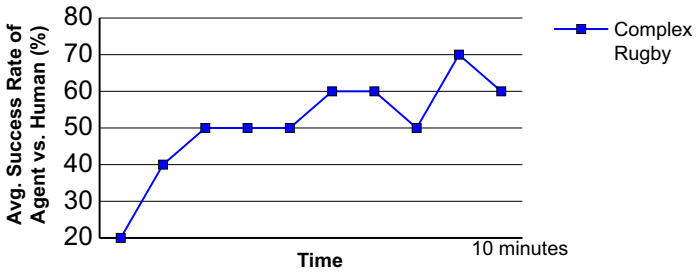

Figure 4.8: After a few minutes, the agent is able to outperform a human user in a complex continuous rugby environment, even though the human has more maneuverability. This experiment started with the agent having very incorrect information about the human user.

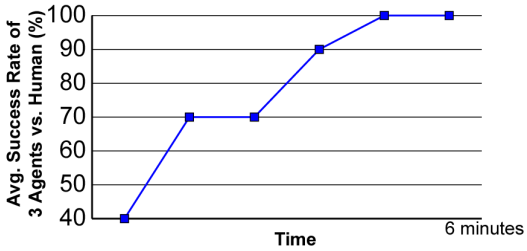

Figure 4.9: Effectiveness of three agents against the human user in the complex rugby environment. The agents learn to predict the actions of each other as well as a human, and thereby learn to cooperate effectively in the rugby environment.

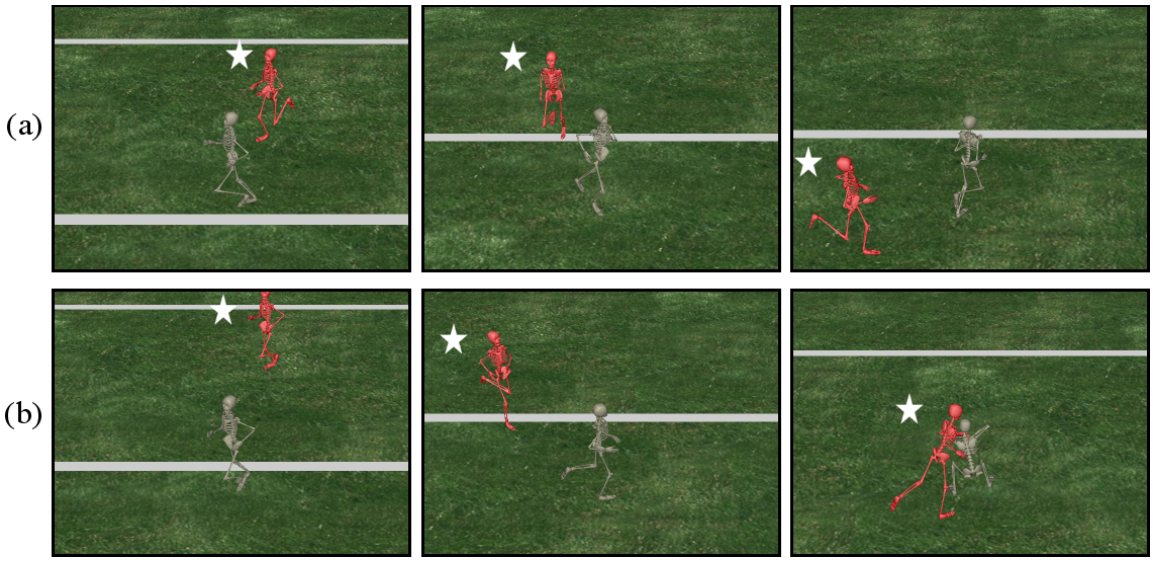

Figure 4.10: (a) The human user performs a loop, which succeeds in getting past the tackler agent (skeleton marked with a star). As a result, the human can score. (b) Now that the agent has adapted, the next time the human attempts a loop it predicts the human's actions and tackles him. Further examples of our technique in action are given in a supplementary digital video, available from http://rivit.cs.byu.edu/a3dg/publications.php.

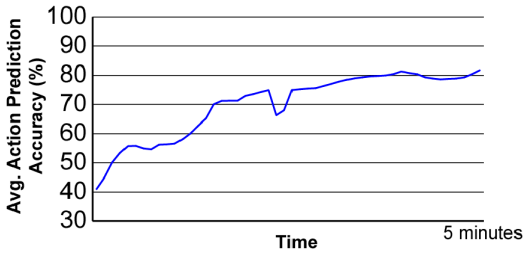

Figure 4.11: Accuracy of predicting the human's actions (L_2-norm) in the Capture The Flag case study. This experiment started with the agent having very incorrect information about the human user.

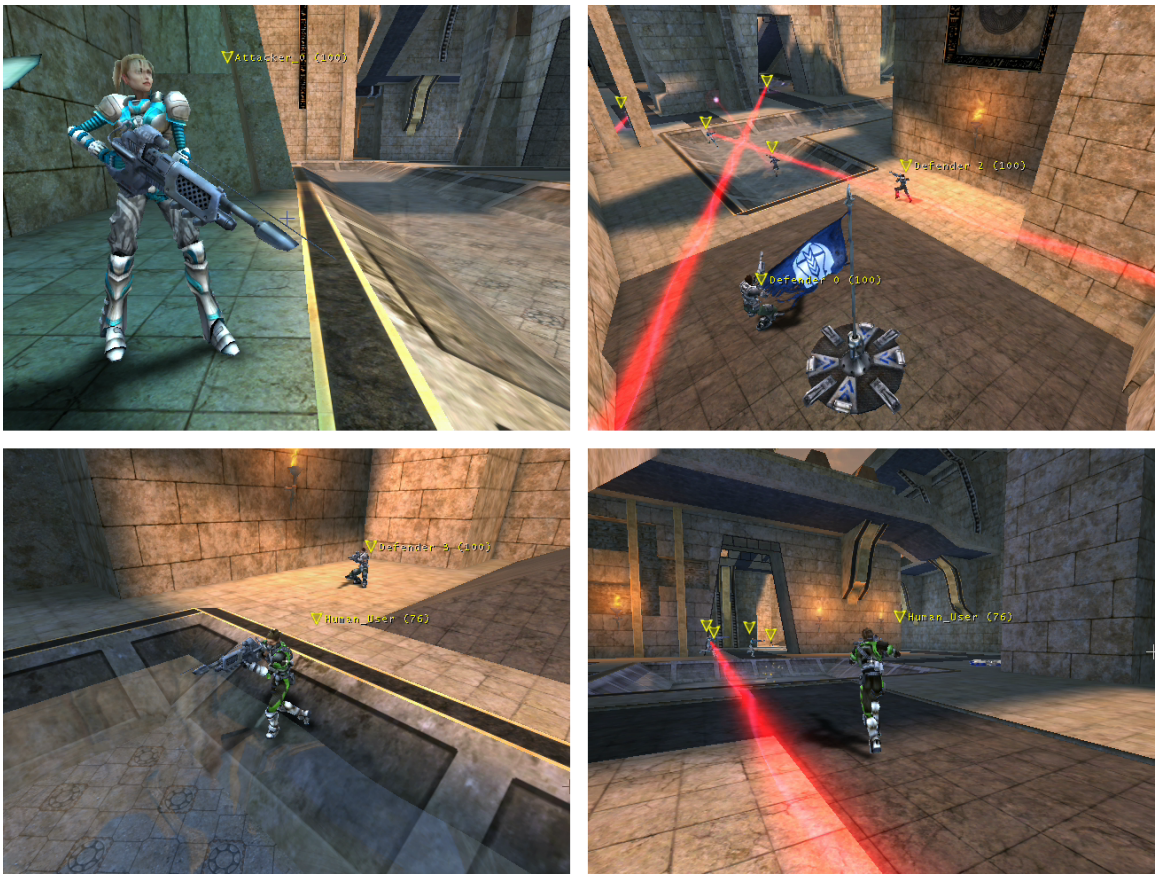

Figure 4.12: Slide show of our Capture The Flag case study.

Chapter 5

Fast Multi-Level Adaptation for Interactive Autonomous Characters

To appear in ACM Transactions on Graphics 2005.

Abstract: Adaptation (on-line learning) by autonomous virtual characters, due to interaction with a human user in a virtual environment, is a difficult and important problem in computer animation. In this paper we present a novel multi-level technique for fast character adaptation. We specifically target environments where there is a cooperative or competitive relationship between the character and the human that interacts with that character.

In our technique, a distinct learning method is applied to each layer of the character's behavioral or cognitive model. This allows us to efficiently leverage the character's observations and experiences in each layer. This also provides a convenient temporal distinction between what observations and experiences provide pertinent lessons for each layer. Thus the character can quickly and robustly learn how to better interact with any given unique human user, relying only on observations and natural performance feedback from the environment (no explicit feedback from the human). Our technique is designed to be general, and can be easily integrated into most existing behavioral animation systems. It is also fast and memory efficient.

Keywords: Computer animation, character animation, behavioral modeling, AI-based animation, machine learning.

Figure 5.1: A virtual character (red skeleton) tries to catch the human's avatar (brown skeleton) in a rugby simulation. By adapting to the human's tactics, the character can become a more challenging, personalized opponent.

5.1 Introduction

Behavioral animation has become popular for creating autonomous self-animating characters. However, an important limitation of traditional behavioral animation systems (with respect to interactive environments) is that they are largely static. In other words, the character's behavior does not adapt on-line due to interaction with a human user and/or its environment. This may not only lead to a lack of variety and non-stimulating animation, but also makes it easy for a human user to predict a character's actions.

Intelligent, rapid on-line adaptation of a character's behavior would be extremely useful for many computer graphics applications. For example, consider a training simulator where a virtual character is an opponent to a human user (see Figure 5.1). By learning on-line through interaction with the human, the character could adjust its tactics according to those of the human it is competing against and thereby become a more difficult, customized opponent.

In this paper, we present a novel multi-level technique for fast character adaptation. Our technique is grounded in traditional machine learning and is further inspired by insights into how humans learn. We focus our discussion on environments where the character

has a cooperative or competitive relationship with the human user. Note that our learning technique is applicable to both *behavioral* (reactive) and *cognitive* (deliberative) models of decision making for autonomous virtual characters. Also, our technique is designed to be general, and can be easily integrated into most existing behavioral animation systems.

Our adaptation technique contains a small set of distinct learning methods. A distinct learning method is applied to each layer of the character's behavioral (and/or cognitive) model. This allows us to efficiently leverage the character's observations and experiences in each layer. Thus the character can quickly and robustly learn how to better interact with any given unique human user, relying only on observations and natural performance feedback from the environment (no explicit feedback from the human).

Each learning method contained in our technique is specially designed with regards to the temporal constraints and degree of abstraction of the decision-making layer to which it is applied. For adaptation in low-level decision making (*action selection*), we take an observation-based approach that operates through case-based reasoning. Specifically, state-action pairs of the human's observed behavior are recorded. These cases are then used to predict the human's future behavior. Through these predictions, a character can extrapolate the long-term utility of any activity it may engage in, and thereby can intelligently select its actions.

For adaptation in mid-level decision making (*task selection*), we take a combination experience- and planning-based approach. First, the character approximately learns in which regions of the state space each of its possible tasks are likely to help achieve its goals. Then, to select a task for the character's current state, the most promising tasks are found and further scrutinized by running internal simulations to more accurately measure their utility. To ensure that the simulations will be adequately accurate, we use the model of the human user's decision making constructed during low-level adaptation as described above.

Finally, for high-level decision making (*goal selection*), we monitor changes in the character's emotional state to adapt its personality. Specifically, the character learns in which situations selecting a given goal leads to increased happiness. This can be combined

with existing interactive character training techniques so that a human user can explicitly adjust the character's personality.

In addition to these learning methods, we also provide the character with the ability to mimic novel, valuable behaviors that it observes the human user's avatar perform. Mimicking is executed in two steps. First, when the character determines that the human has achieved her goal, the human's behavior is tested to see if it is novel to the character; if it is novel, the corresponding state-action sequence is recorded. Second, the character can explicitly mimic this behavior when appropriate and desired.

Our multi-level adaptation technique allows an intelligent, complex character to adapt in the following ways:

- Learn to accurately predict the human's actions.
- Learn the best behavior for any given situation.
- Mimic the effective behaviors of the human.
- Learn through emotional feedback to maximize happiness.

We begin by surveying related work. We then give a brief introduction to the theoretical background of our technique, both in terms of behavioral animation and machine learning. Next we turn to a discussion of the key challenges in successfully performing fast and robust adaptation, providing a roadmap of what our technique must accomplish. We then present our technique for rapid adaptation of virtual characters. Finally, we conclude with our experience from three case studies. We use our first case study (a virtual sport like rugby or American football) for examples throughout this paper. Our rugby case study is a challenging environment for character adaptation, because it is very fast-paced, involves synthetic humans, and the state/action spaces are continuous.

5.2 Related Work

Our multi-level approach to adaptation is inspired by [Stone 2000], where agents (characters) in a multi-agent environment learn off-line in a layered fashion to perform their

tasks and thereby learn better than with a direct, non-layered approach. What is new in our work is that we apply this layered-learning concept to achieve *faster* learning so that our characters can adapt on-line within the tolerances of human time and patience. Also, the per-layer learning methods used in [Stone 2000] are standard machine learning techniques, whereas we have developed custom learning methods to fulfill our unique needs.

In some aspects, our custom per-layer learning methods resemble existing machine learning approaches. Our low-level learning method most closely resembles *agent modeling* or *user modeling* [Gmytrasiewicz and Durfee 2000; Kerkez and Cox 2003; Zhu et al. 2003]. The notable differences from these existing techniques include that our method operates in continuous state/action spaces, can predict the human user's behavior several steps into the future, and can exercise caution or confidence when predicting actions. Our mid-level adaptation method is most closely related to deliberation-based approaches to learning [Yoon 2003], but leverages *reinforcement learning* [Kaelbling et al. 1996; Sutton and Barto 1998] concepts to expedite the deliberation process. Our high-level adaptation method is very similar to previous work in computer animation for allowing a user to interactively train a virtual character [Blumberg et al. 2002; Evans 2002], as well as methods for learning from change in emotional state [Tomlinson and Blumberg 2002; Gadanho 2003]. Our mimicking technique most resembles previous work in *learning from observation* or *programming by demonstration* [Price 2002; Kasper et al. 2001; van Lent and Laird 2001], but our approach is either significantly faster, more general, or more automatic than previous techniques.

As pointed out above, while our per-layer learning methods bear similarities to existing techniques in machine learning, several aspects of our learning methods are novel. We more thoroughly discuss these novel aspects later in this paper after presenting each of our learning methods.

Our approach is also inspired by a study of how humans learn [Minsky 1985; Meltzoff and Moore 1992; Schyns et al. 1998]. Humans learn through a variety of stimuli, both observed and experienced, and apply lessons learned in many distinct ways. Therefore, it is sensible that a robust character adaptation technique must also be multi-faceted, leveraging

pertinent experience in several distinct ways. The optimal use of one experience to learn several lessons also makes learning faster.

A number of noteworthy architectures for behavioral animation of autonomous characters have been proposed [Reynolds 1987; Tu and Terzopoulos 1994; Blumberg and Galyean 1995; Perlin and Goldberg 1996; Funge et al. 1999; Faloutsos et al. 2001; Isla et al. 2001; Monzani et al. 2001]. While producing impressive results, most of these systems have not incorporated any form of learning and therefore cannot adapt to more skillfully interact with any given human user.

Off-line behavioral learning has recently seen some attention, e.g., in our own work [Dinerstein et al. 2004b; Dinerstein and Egbert 2004]. However, off-line learning does not address the problem of interest in this paper: interaction-based adaptation.

The multi-level adaptation approach we propose in this paper has strong underpinnings in the Belief-Desire-Intention (BDI) agent model [Rao and Georgeff 1995]. In BDI, the agent's internal state is composed of *beliefs* (i.e., knowledge), *desires* (i.e., goals), and *intentions* (i.e., plans). The agent uses this internal state to select actions. Our approach is similar in that we collect knowledge through interaction, which the character leverages to more effectively make decisions and thereby fulfill its goals. However, our approach varies from traditional BDI because we acquire and utilize knowledge in a layered fashion.

A notable work performed in virtual character adaptation through prediction is [Laird 2001]. In this technique, the character memorizes facts about the layout of the environment and then uses this information to weakly predict the human's future behavior (assuming the human will behave exactly like the character would in the same situation). This prediction is then used to improve the virtual character's decision making. This interesting technique is one of the first working examples of useful adaptation in practice. However, the character only learns about the layout of the environment, not the human's behavior, so the character's ability to adapt is distinctly limited.

On-line, interaction-based behavioral learning has only begun to be explored [Evans 2002; Tomlinson and Blumberg 2002]. A notable example is [Blumberg et al. 2002], where a virtual dog can be interactively taught by the human to exhibit desired behavior. This technique is based on reinforcement learning with immediate explicit feedback from the

human user, and has been shown to work extremely well. However, it has no support for long-term reasoning to accomplish complex tasks (i.e., it is reactionary). Also, the approach is based on continual and explicit feedback from the character's "master". Therefore these techniques, while interesting and useful, are best suited for a "master-slave" relationship between a character and a human user, not a competitive or cooperative relationship. It may be possible to alter [Blumberg et al. 2002] such that the underlying learning algorithm utilizes feedback from a synthetic trainer rather than a human, but it is not clear at this time how this could be done.

There is need for a technique that allows autonomous virtual characters with complex behavioral and/or cognitive models (in a competitive or cooperative relationship with the human) to rapidly adapt on-line to better fulfill their goals in an interactive environment. This adaptation should be effective, believable, fast enough to not tax human patience, and appear similar to the way a real human peer or competitor would learn through interaction. Our contribution is a technique that fulfills these needs.

5.3 Background

5.3.1 A Common Behavioral Animation Framework

For our on-line character adaptation technique to be broadly useful, it must be applicable in most, if not all existing behavioral animation systems. As discussed in [Millar et al. 1999], behavioral animation systems can be cast into a common framework. While the details of these systems can vary significantly, they are nevertheless quite similar from a framework-level perspective. We develop our technique within this common framework so that it can be usable in current and future behavioral animation systems.

A simplified common behavioral animation framework (as seen from a high level) is shown in Figure 5.2. Behavioral animation systems in general have three primary modules: perception, behavior, and motor control. We are most interested in the behavior module, as this is where decision making takes place and therefore is key to adaptation. This module is often called the *behavioral model*. Further details of this module are shown in Figure

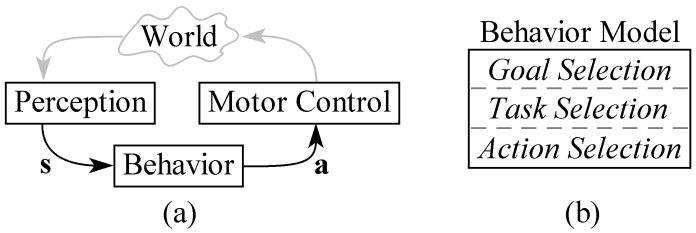

Figure 5.2: (a) A simplified common behavioral animation framework. **s** is the current perceived state, and **a** is the action selected to be performed. (b) Most behavioral models are layered to break up the decision-making process into tractable pieces.

5.2b. The decision-making process is usually split into layers, decomposing it into manageable pieces. This layered approach to decision-making has been standard for control of autonomous agents since the pioneering work of Brooks [Brooks 1986]. The upper layers perform higher-level, coarse-grain decision making, while the lower layers perform lower-level, fine-grain decision making. This provides a natural and powerful breakup of the decision-making process.

It is most common, both in behavioral animation and overlapping fields, for there to be three layers in the behavioral model. However, there are no universal names for these layers. Throughout this paper, we refer to these as *action selection*, *task selection*, and *goal selection* (in order of fine to coarse temporal granularity).

Our character adaptation technique integrates with a behavioral animation system through the layers of the behavioral model. A distinct learning method is applied to each layer. This allows us to efficiently leverage the character's observations and experiences for each layer, based on that layer's unique degree of abstraction and temporal constraints. However, note that while our technique is specially designed for three-layer behavioral models, it can be used effectively with models of any number of layers.

There are two types of decision-making models that are popular for use in behavioral animation. The traditional *behavioral model* [Reynolds 1987] performs reactive decision making. *Cognitive modeling* [Funge et al. 1999] was introduced to provide virtual characters with deliberative decision making. Our adaptation technique works well for either approach. However, for simplicity, and without loss of generality, we will use the term "behavioral models" to represent both behavioral and cognitive models.

5.3.2 Machine Learning

Our multi-level character adaptation technique is composed of four custom learning methods. These methods are grounded in traditional machine learning approaches, but are novel in several aspects. We now briefly review pertinent issues in machine learning, and give some motivation for our need to develop custom learning methods to achieve practical on-line character adaptation.

Formally, machine learning is the process of a computer program learning an unknown mapping, often formulated as $f : \mathbb{R}^n \rightarrow \mathbb{R}$. There are numerous machine learning techniques presented in the literature [Mitchell 1997]. Each approach has its own strengths and weaknesses, and each is a good fit for some category of applications. Nevertheless, there is no machine learning technique that works well for all applications (thus the existence of so many alternative approaches).

In general, there are four main categories of machine learning techniques:

1. *Example-based learning*
2. *Experience-based learning*
3. *Planning-based learning*
4. *Observation-based learning*

Most existing machine learning techniques are example-based. They learn using a large set of explicit input-output examples provided by the user. These techniques include artificial neural networks, decision trees, etc [Mitchell 1997]. Many of these techniques learn very slowly and/or require a large number of input-output examples, and therefore are not appropriate for interactive, on-line learning. Another challenge is that it may be inappropriate or prohibitively difficult for an end user to provide the necessary input-output examples. Thus these techniques (in their traditional form) are not applicable for our problem of interest.

Reinforcement learning [Kaelbling et al. 1996; Sutton and Barto 1998] is an effective approach to experience-based learning. In reinforcement learning, the world in which the

agent lives is assumed to be in one of a set of perceivable states. The objective is to learn the long-term value (i.e., *fitness*) of each state-action pair. The main approach taken is to probabilistically explore states, actions, and their outcomes to learn how to act in any given situation. Unfortunately, while this approach learns *well*, it does not learn *fast*. For example, the system TD-Gammon [Tesauro 1995] taught itself to play Backgammon at a master's level. However, to learn this, it had to play two million games against itself! While it is practical for a software program to play against itself this many times off-line, it is impractical to expect a human to spend so much time interacting with it. Indeed, human patience is the most critically scarce resource in interaction-based adaptation. There are further challenges in using traditional reinforcement learning for interactive adaptation, such as requiring discrete state and action spaces, etc.

Thus far we have discussed example- and experience-based machine learning, the most common approaches. There are other machine learning approaches, e.g., *planning*- and *observation-based*, which could be considered for use in character adaptation. For example, planning through a tree search [Sutton and Barto 1998] to compute the long-term value of each state-action directly, rather than waiting for the value to backup using local updates. This places the burden of learning more on processing power than continually repeating experiences, so learning can occur far more quickly with respect to human interaction. However, planning requires a complete model of the character's environment, including all agents therein—but we do not have a model of the human user's behavior.

Observation-based learning appears especially applicable to our needs in interactive character adaptation. These techniques include agent/user modeling [Gmytrasiewicz and Durfee 2000; Kerkez and Cox 2003] and imitation [Price 2002]. In agent/user modeling, the agent learns to predict the actions of someone else and can thereby predict the utility of any behavior it may engage in. In imitation, the agent memorizes the behavior of another agent or user and mimics it. These observation-based approaches to machine learning are interesting because they are natural for on-line use. However, existing techniques are limited to discrete state/action spaces, or do not learn quickly from few observations, etc.

Therefore, while current approaches in machine learning provide us with a solid theoretical foundation to build on, existing techniques do not solve our problem of interactive adaptation for virtual characters.

5.4 Making Adaptation Practical

5.4.1 Requirements

While machine learning provides a theoretically sound basis for building systems that learn, there are a number of issues that make existing techniques problematic in the context of interactive adaptation for autonomous animated characters. We have identified the following characteristics that are necessary for an adaptation algorithm (for cooperative or competitive autonomous characters) to be practical and useful, listed in order of importance:

1. Fast learning. Human time and patience is the resource we will certainly have the least of. Further, slow learning will not make a character keep up with a fast-learning human. Therefore, adaptation must occur quickly based on few experiences.

2. No explicit human feedback. To achieve adaptation for truly autonomous cooperative or competitive relationships, we cannot ask the human to supply detailed feedback on the character's every action (this denotes a master-slave relationship, and could interrupt the flow of an animation). Therefore, the adaptation must occur without any explicit human feedback, just natural feedback from the environment.

3. Believable adaptation. To be convincing, adaptation must appear intelligent. That is, when the character learns, it should usually (or always) become better at its task.

4. Must perform at interactive rates on a PC. For adaptation to be as widely useful as possible, it must be practical for interactive use on current PC CPUs.

5.4.2 Assumptions

In order to achieve the above listed goals, we make the following assumptions. These assumptions, however, are valid within our theoretical foundation and do not impose any important limitations:

1. Knowledge acquisition is sufficient for adaptation. We assume that our character already possesses adequate motor control, perception, and decision-making skills. Therefore, to optimally interact with a unique human user, all that it needs to do is gather some key knowledge with which to guide its decision making.

2. Natural-looking learning is sufficient. Our adaptation technique does not need to be as powerful as traditional reinforcement learning. We know from ethology that Nature places a premium on learning adequate solutions quickly.

3. Insight is used to accelerate adaptation. It is well known that insight by the programmer into what the character must learn can greatly simplify the learning process. We assume that the programmer provides such domain knowledge in the form of a good compact state definition, and a *gradient fitness function* [Kaelbling et al. 1996] (i.e., reward is given for approaching goal states).

4. The current state and the user's action are observable. All of our learning methods utilize the current state of the environment, and some utilize the human user's actions. Thus the current state and user's actions must be observable. This assumption is reasonable in our problem domain because the low-level actions of the human user are explicitly input through a device such as a joystick. Also, the virtual environment exists in its entirety in software and therefore is perfectly observable (with the exception of the user's internal state, which we infer as discussed later in the paper).

5.5 System Description

We now give a detailed description of our multi-level technique for fast character adaptation. First, note that we start with a non-user-specific behavioral model that is constructed off-line. This model provides the baseline behavior for the character. It is this

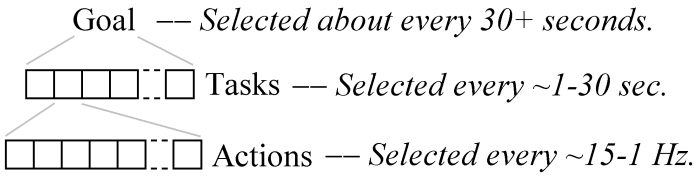

Figure 5.3: Relationship between *goals*, *tasks*, and *actions*. These are the names we apply to these temporal levels of decision making.

model that we adapt on-line so that the character will better interact with a unique human user.

In our technique, we fulfill the requirements listed in Section 5.4.1 by using a combination observation-, experience-, and planning-based approach to machine learning. Moreover, each layer of the behavioral model is treated differently; the specific approach to learning used in each layer is unique to the needs and temporal constraints of that layer. In fact, it is the temporal granularity of decision making in each layer that primarily dictates what approach to learning will be most effective. We will discuss this in more detail later.

There are no universal names for the decision-making layers of behavioral models. For clarity, we use the following names (see Figure 5.2): *action selection*, *task selection*, and *goal selection* (in order of fine to coarse temporal granularity). This relationship is demonstrated in Figure 5.3. An *action* is a very low-level decision, which can be directly translated into motor control. A *task* is somewhat more high level, requiring several actions to perform. A *goal* is the highest level, representing the character's strongest current desire, and can be broken up into several tasks. The current goal loosely determines the tasks to perform, etc.

An overview of our adaptation system is given in Figure 5.4. As mentioned previously, it is composed of a set of discrete learning methods which are applied to individual layers in an existing behavioral (reactive) or cognitive (deliberative) model. Note that if a given model has fewer than three layers, an appropriate subset of our learning methods can be used.

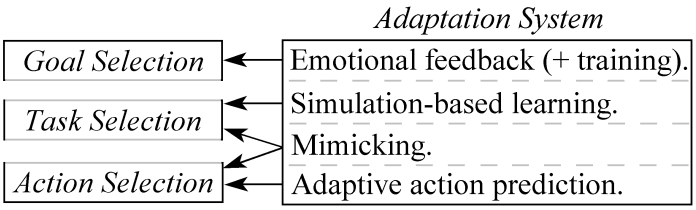

Figure 5.4: Overview of our adaptation system. Individual learning methods are applied to the layers of a behavioral model.

5.5.1 State and Action Representations

For any given character and virtual environment, our state space may have to be continuous. This is because it is possible that a small difference in state can determine a large difference in behavior. A continuous state space can also help in achieving smooth and aesthetically pleasing character animation. Therefore, our technique uses a continuous internal representation of states and actions. Since this is more general than a discrete space, both representations are naturally supported.

We represent the state space as a real-valued, n-dimensional feature vector. Thus a state $\mathbf{s} \in \mathbb{R}^n$ is a point within this feature space.

A *feature* is some salient aspect that characterizes the state of the world. Usually, even complex worlds can be effectively represented with a compact set of features. In fact, there are known techniques to automatically discover salient features, e.g., principal component analysis [Mitchell 1997]. Alternatively, human intuition can be applied to this problem. As was stated in Section 5.4.2, we assume that the programmer has provided a good, compact state representation for the given character/world. This is important because a compact state space will help the character adapt quickly and better generalize what it has learned. For more information on selecting a compact set of features, see [Reynolds 1987] or [Grzeszczuk et al. 1998]. As an example, in our 1-on-1 rugby case study, the state is composed of the translation-invariant separation of the two characters, and their velocities.

Like states, actions are real-valued vectors, $\mathbf{a} \in \mathbb{R}^m$, so that both discrete and continuous action spaces are supported. Actions should be organized in such a way that they can be combined into some sort of "intermediate" action (e.g., 'left' and 'forward' become 'diagonal').

It is important, for the sake of generalization, that our real-vector-valued states and actions be organized such that similar states usually map to similar actions. More formally:

$$(\|\mathbf{s}_1, \mathbf{s}_2\| < \varepsilon_a) \Rightarrow (\|\mathbf{a}_1, \mathbf{a}_2\| < \varepsilon_b),$$

where $\| \cdot \|$ is the L_2-norm, and ε_a and ε_b are small scalar thresholds. Certainly, this constraint need not always hold, but the smoother the relationship the simpler it will be to learn.

If our adaptation technique is to be integrated into an existing behavioral animation system, it may be necessary to transform states and actions into our internal real-vector-valued representation. This can be performed through a simple, custom transformation:

$$T_{\mathbf{s}} : \texttt{state} \rightarrow \mathbf{s} \in \mathbb{R}^n, \quad T_{\mathbf{a}} : \texttt{action} \rightarrow \mathbf{a} \in \mathbb{R}^m, \quad T_{\mathbf{a}}^{-1} : \mathbf{a} \rightarrow \texttt{action}$$

where teletype signifies the external format of states and actions.

While we assume in this paper that the programmer has provided an effective and compact state space, there are several techniques available for automatic state space discovery (e.g., [Blumberg et al. 2002; Guyon and Elisseeff 2003]). Our motivation for using explicitly designed state spaces is that they have often proven superior in machine learning experiments reported in the literature. Nevertheless, better results may be achieved through automatic techniques when the programmer is inexperienced with designing compact state spaces.

5.5.2 Low-Level Learning (for Action Selection)

The primary challenge faced in performing low-level adaptation (i.e., learning in the action selection layer) is that, with the decision making being so temporally fine-grain, it is impossible to quickly learn long-term values of state-action pairs. However, due to this fine temporal granularity, it is easy to observe the human's behavior in the form of state-action pairs. This is possible because the human's actions are explicitely input using a device such as a joystick, and the state is entirely observable because the virtual world exists in software (with the exception of the human's internal state which we infer as discussed below).

Therefore, we take an observation-based approach to adaptation for action selection. Specifically, through observation, we can construct a Markovian model of the human's behavior. We can then use this model to predict the human's future behavior, and thereby our character can more wisely select actions to perform (see Figure 5.5).

In our action selection adaptation method, the behavior of the human user is recorded on-line in the form of state-action pairs. That is, at each time step, the current state and the action selected by the human are saved. For simplicity, we use a small constant time step for sampling the human's state-action pairs. For example, in our rugby case study, the time step matches the frame rate of the animation (15 Hz).

The model of the human's behavior is constructed through case-based learning. Each recorded state-action pair is treated as a case in a case-based reasoning engine. A library is maintained of useful cases. Since the state space is continuous, the library is organized as a hierarchal partitioning of the state space. Partitioning is important so that fast lookup of cases can be performed. Automatic partitioning techniques (e.g., a kd-tree) can be used to great effect. Alternatively, partitioning can be performed by the programmer so that human insight may be applied.

To predict the human's future behavior, the library of cases must be generalized so that, for any given query state, an associated action is computed. To fully exploit our knowledge of the human, we generalize through the *continuous* k-*nearest neighbor* algorithm. That is, the k cases closest to the query state (according to the Euclidean metric) are found and a distance-weighted normalized sum of the associated actions is computed:

$$\tilde{\mathbf{a}} = \frac{\sum_{i=1}^{k} (w_i \cdot \mathbf{a}_i)}{\sum_{i=1}^{k} w_i}, \qquad \text{where } w_i = \frac{1}{{d_i}^2}.$$

The tilde notation signifies that the answer is approximate. We have found $1 \leq k \leq 3$ is effective. $k = 1$ is good for exactness, as no blending of actions occurs. However, $k = 3$ is good if there is no closely matching case, and/or for attaining a more general impression of the human's behavior. Note that it is helpful to normalize the axes of the state space, so that they will contribute equivalently in computing distance.

Alternatively, to focus on caution rather than exploitation, we generalize using a custom modified minimax search. The k cases closest to the query state are found. Then,

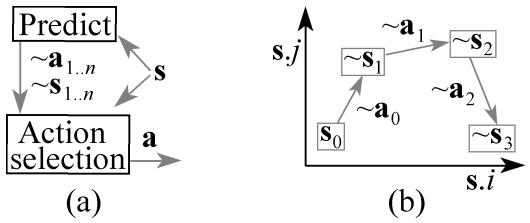

Figure 5.5: (a) Structure of our low-level adaptation method. An n-step sequence of states and actions into the future is predicted, then action selection uses this information to make wiser choices. (b) To predict the human's actions n steps into the future, we predict/compute the actions of all agents in the environment, determining a new state, and then repeat. The tilde notation signifies that the predictions are approximate.

the k actions associated with the retrieved cases are tested with the character's own fitness function. Finally, the action that results in the minimum fitness is assumed to be the one the human will select. More formally:

$$\tilde{\mathbf{a}} = \arg \min_{\mathbf{a}_i \in k \text{ cases}} (\text{fitness}(\mathbf{s}, \mathbf{a}_i)).$$

For this cautious generalization, we prefer $3 \leq k \leq 16$. The greater the value of k, the more cautious the generalization will be.

It is important that we predict the human's actions several steps into the future, so that our character can make wise decisions. To do this, we first predict the human's action for the current time step, then we either compute, predict, or assume the actions of all other agents in the virtual world. This allows us to predict the future state, which in turn allows us to predict the next action taken by the human, and so forth. We have found that predicting between 5 to 15 steps into the future works well, is accurate enough to be useful, and usually requires little CPU time.

The case library is originally populated with state-action examples of "generic" human behavior. These are gradually replaced with user-specific examples, as they are observed by the character. In particular, a limited number of cases $(\mathbf{s}, \mathbf{a})_i$ are allowed for each region r_j of the state space. Cases are selected for replacement based on their age and unimportance. In other words, if a case was recorded long ago, and/or is very similar to the new case to be added, it is likely to be removed. Thus the character has the ability to "forget", which is very important in learning something as non-stationary as human behavior.

We formally define the case replacement as:

$$\text{replace} \quad \arg\min_{(\mathbf{s},\mathbf{a})_i \in r_j} (M(\mathbf{s},\mathbf{a})_i) \quad \text{with the currently observed case} \quad (\mathbf{s}_t,\mathbf{a}_t),$$

using a metric M:

$$M((\mathbf{s},\mathbf{a})_i) = -\alpha \cdot \mathsf{age} + \beta \cdot \|(\mathbf{s},\mathbf{a})_i,(\mathbf{s}_t,\mathbf{a}_t)\|.$$

Human decision-making and behavior is non-deterministic. Therefore, it is critical that our action prediction technique properly handle non-determinism. Because we explicitly store discrete state-action cases, non-determinism can be represented in our case library. Both of our approaches to case generalization properly handle non-determinism, but in different ways. k-nearest neighbor is less tolerant of non-determinism than minimax, since conflicting cases can average out to be a null action. However, in situations of greater case homogeneity, k-nearest neighbor produces accurate predictions of average behavior. In contrast, our custom minimax always properly handles non-determinism, since it merely searches for the action that will most damage the character's fitness.

The reason our observation-based approach to low-level adaptation is sufficient is because the character learns all the non-stationary knowledge it needs to wisely select actions. By accurately predicting the human's behavior, the character can predict the results (i.e., utility) of its actions and can thereby wisely select what to do.

An alternative way to use action prediction (rather than predicting an entire sequence of actions given an initial state, as in Figure 5.5) is for the behavioral/cognitive model to request individual predictions for specific states. This can be especially useful for a cognitive model, as it can request information specific to any state it encounters while deliberating. However, this requires more CPU than a single linear prediction.

Of course, the human user's decision making will vary based on her current goal. As a result, it can be useful to employ multiple state-action case libraries, one for each goal. As discussed in works such as [Blumberg et al. 2002; Evans 2002], the human's goal can be easily inferred because the virtual environment constrains what she needs to accomplish.

Relationship To Previous Work

Our action prediction method is related to other agent/user modeling techniques, such as *Case-Based Plan Recognition* (CBPR) [Kerkez and Cox 2003]. CBPR fundamentally operates like our method, using state-action cases. However, CBPR is limited to discrete state/action spaces, does not fully generalize cases, and can only predict an agent's behavior one action into the future. Moreover, CBPR always assumes the closest case in the state space is correct, whereas our technique can exercise caution or confidence when predicting actions. In fact, our technique can even predict the Nash equilibrium strategy if the proper information is available in the local cases. Another related technique is *Maximum Expected Utility* (MEU) [Sen and Arora 1997], which is a modification of minimax. However, MEU is limited to discrete state/action spaces, and can require significant CPU (exponentially increasing) to predict behavior several steps into the future.

In [Gmytrasiewicz and Durfee 2000], a hierarchical approach to agent modeling is used to produce coordination between software agents. This technique assumes that all agents desire to cooperate and that payoff matrices are sufficient to model behavior. Thus this technique is limited to producing only cooperative behavior, and is limited to discrete state/action spaces. Moreover, this technique is likely not plausible for interactive use, because it creates a tree-like nesting of models, which can require significant storage and processing power. Our action prediction method contrasts with hierarchical techniques like [Gmytrasiewicz and Durfee 2000] because we use a single "flat" model, which is updated to represent recently observed behavior and forget older behavior. Our method has theoretical underpinnings in *Fictitious Play* theory [Stone and Veloso 1997].

Another related technique is [Zhu et al. 2003], an example of user modeling. In this technique, the system learns to predict the activities of a user interacting with a web browser. Like this technique, most other user modeling techniques are focused on interaction through GUI interfaces. The fundamental assumption is that the agent-user interaction takes place through a constrained interface. Thus they are not appropriate for our problem domain of graphical characters interacting directly with a human user in a virtual world.

We believe that our action prediction method may be very applicable to problems outside of interactive virtual character adaptation, since it has unique strengths as compared

to existing techniques. However, an examination of these uses is outside the scope of this paper and therefore left to future work.

5.5.3 Mid-Level Learning (for Task Selection)

The challenges faced in performing mid-level adaptation are different than those in low-level adaptation. This is primarily due to the fact that the temporal granularity of decision making is now more coarse. Specifically, action selection is performed quite often (about once every 15 to 1 Hz), while task selection is more seldom (about once every 1 to 30 seconds).

The primary challenge faced in mid-level learning is that we cannot with certainty observe the human's behavior in the form of state-task pairs. This is because task selection is at a conceptually high enough level that it is likely to involve a large amount of hidden state information inside the human's brain (i.e., it is non-Markovian). As a result, it is impossible to determine exactly what motivated the human to select tasks as she did. Therefore, we cannot take a direct observation-based approach here like we did for action selection (low-level learning). Moreover, we also cannot easily take an experience-based approach for task selection, as this can take a long time to learn.

Our approach to mid-level adaptation is primarily planning-based, but also involves experience- and observation-based reinforcement learning. The key is to be able to run simulations (i.e., plan) to determine which candidate task will most likely perform best for the current state. As mentioned in our review of machine learning, we can simulate the outcome of any decision if and only if we have a complete model of the environment, including a model of the human's behavior. Luckily, through our approach to adaptation we already have a model of the human's low-level behavior, constructed during low-level adaptation. We reuse this model here, to run simulations.

Our mid-level adaptation technique involves the following steps, as shown in Figure 5.6:

1. **Estimate value.** Compile a small set of candidate tasks that have the highest estimated values (utilities) for the current state.

2. **Simulate.** Run internal simulations for each of the candidate tasks, to more accurately measure their utility.

3. **Select.** Rank candidate tasks according to their utility. Once this is done, the behavioral model then selects one.

The reason we use step #1 rather than proceeding straight to step #2 is because simulating all possible tasks may be implausible. Therefore, to limit the set of candidate tasks to a practical number, we need rough approximations of each task's value so that poor candidates can be eliminated early. To do this, we use case-based libraries of state-task values, one library for each task. These libraries perform a state $\rightarrow$ value mapping, i.e., determining for any given state the utility of selecting a given task. The libraries are structured and evaluated similar to those in low-level adaptation, using k-nearest neighbor. However, the case population of the state-task libraries is much more sparse (maximizing learning speed) because we only need rough approximations; we will run simulations to provide more accuracy as needed. The state space can be uniquely defined and/or partitioned for each state-task library, if desired. Also, the libraries are originally initialized with regards to generic human behavior.

Credit assignment is performed to update the state-task values both after running simulations (according to their predicted utility) and after selecting a task (according to feedback from the environment). Updating after a simulation is simple, as we know exactly what state-task value should be updated. However, updating due to feedback is more difficult, as feedback is usually delayed. We could use a traditional local update rule from reinforcement learning, but this is too slow. Instead, as shown in Figure 5.7, we maintain a chain of the previous n state-tasks visited by the character. The value of every state-task in the chain is updated every time new feedback is received. The longer the chain, the more accurate to long-term utility the state-task values will be. The actual update of the case-based library of values is performed by updating the existing cases closest to the actual case we wish to update; no new cases are added nor are any removed. More formally:

$$\text{value}(\mathbf{s}_j, task)' = \text{value}(\mathbf{s}_j, task) + \alpha \cdot (\tilde{}value - \text{value}(\mathbf{s}_j, task)), \ \forall (\mathbf{s}_j, task) \in k \text{ neighbors},$$

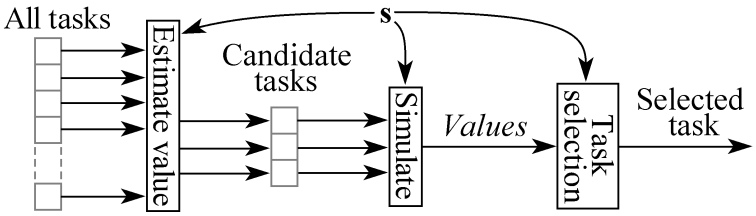

Figure 5.6: Structure of our mid-level learning method. The values of all tasks are estimated for the current state, then promising tasks are evaluated more accurately through simulation.

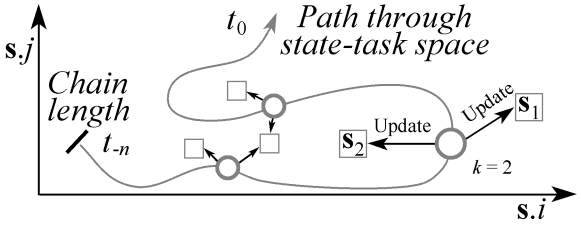

Figure 5.7: To know which state-task values to update, we store the path traversed by the character. Then, after receiving feedback, we update the k cases closest to each state-task visited.

where $\alpha \approx 0.5$ is the scalar learning rate, and $\tilde{}value$ is the apparent state-task utility. The maximum aggregate change of a case is bounded for each time it is visited (e.g., 40% of its possible range).

The internal simulations are run for a reasonable amount of time into the future. We have found that a few seconds in the character's time frame usually works well. During the simulations, the human's actions are predicted using the case-based model constructed during low-level learning. If there are other computer-controlled characters besides the one currently performing task selection, their actions can either be computed, predicted, or assumed. Surprisingly, we have found that it usually is sufficient to assume constant action (e.g., the character's last performed action). It can also work well to compute only one new action for every several time steps. Such assumptions can help speed up the simulation for more computationally complex behavioral or cognitive models.

The apparent utility of the simulated and/or executed task is computed as the average of all feedback received during the simulation or execution:

$$\tilde{}value = \sum_{t=t_0+1}^{t_0+n} \text{fitness}(\tilde{}\mathbf{s}_t),$$

where t_0 is the current time step, and n is the number of steps to simulate into the future. No emphasis is given to early or latter time steps. As mentioned in Section 5.4.2, a *gradient fitness function* is used, meaning that feedback should be received for most/all states visited. While strong feedback is given for reaching a goal or terminating state, weak feedback is given for intermediate states. This gradient fitness function, if designed properly by the programmer, can help lead even short simulations to optimal long-term utility.

Our mid-level learning technique can take time to compute, especially if the character's behavioral/cognitive model is computationally complex. When necessary to maintain interactive performance, it is possible to simply select a task using the case-based state-task values. Note that task selection does not need to be performed between animation frames—the simulation process can be spread out over several frames, as long as the current state does not vary too greatly.

Relationship To Previous Work

Our case-based approach to learning state-task values has a solid foundation, as local function approximation is currently considered to be the best approach for learning value functions [Sutton and Barto 1998]. The aspect of our task-selection adaptation method that is most novel is the combination of experience- and simulation-based machine learning concepts. In fact, we are not aware of any existing techniques that use a similar method. By taking a combination approach, we are able to achieve sufficiently accurate results with reasonable CPU use and storage requirements (see the results section). In comparison, traditional reinforcement learning forces an agent to experience all state-task pairs multiple times to learn their values.

Another interesting aspect of our approach is that it is a practical realization of using deliberation to generalize sparse learning, a relatively unexplored direction in machine learning that has recently seen a great deal of interest [Yoon 2003]. In fact, it is currently hypothesized by many cognitive scientists and engineers that deliberation is a necessary ingredient to achieve human-like learning (see the proceedings of DARPA 2003 Cognitive Systems Conference for more information on this topic).

5.5.4 Mimicking (for Action and Task Selection)

In our system, *mimicking* (i.e., imitation) is another learning method for low- and mid-level decision making. It compliments the learning methods already presented by gathering knowledge in a different fashion. While less general than our other learning methods, it is very fast and can easily encapsulate very complex behaviors and decision-making.

Mimicking provides the character with the ability to quickly learn novel behaviors in a clever and natural, yet indirect way. Moreover, creating truly novel behaviors is difficult, whereas mimicking is far simpler to perform and is highly likely to improve the performance of a character. Another interesting aspect of mimicking with regards to interactive virtual environments is that it leverages human intelligence in a non-intrusive, intuitive way. While the human user interacts with a challenging, adapting environment, she will be forced to adjust her behavior and tactics. Thus she will attempt possibly novel behaviors. The character can observe these new behaviors and, based on their apparent success, decide to remember them for later imitation when faced with the same situation as the human was.

Our mimicking technique is summarized in Figures 5.8 and 5.9. In our system, our mimicking method affects both the task and action selection layers. This is because the observed novel behavior, which the character wishes to mimic in the future, is stored as a new task. If this new task is selected, it overrides action selection by forcing the recorded chain of actions to be performed, as they were observed. Each recorded observed behavior is stored as its own task, and has individual candidacy to be selected for execution by the behavioral model.

Since the novel observed behaviors are treated as tasks, we must associate state-task values with them to determine when they should be simulated during task selection (see Section 5.5.3). We do this by creating a new case library for each recorded behavior, with state-task value cases for the states visited by the human while executing this behavior, plus a sparse set of additional cases that are regularly positioned throughout the remainder of the state space. Those states that were visited during the human's execution of the behavior are initialized with "good" values, while all other regions of the state space are given "poor" values. These state-task values are thereafter updated just like those of standard tasks. If the

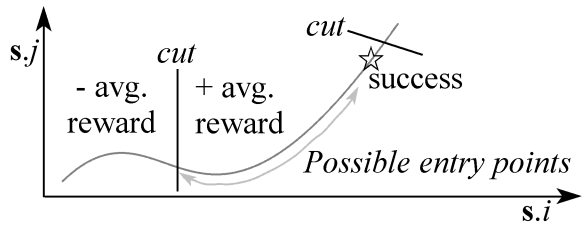

Figure 5.8: Overview of our mimicking technique. The character is interested in sequences of actions where the human's situation improved overall (i.e., positive average reward), culminating in achieving her goal. This action sequence is recorded and later mimicked.

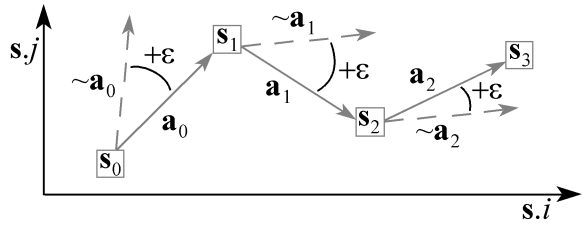

Figure 5.9: To determine if an observed action sequence is novel to the character, we compute the average difference, ε, between the recorded actions and those that the character would normally select.

behavior fails too often (e.g., 3 times consecutively), it is assumed to be entirely ineffective and is deleted.

An important question is how to determine the beginning and ending of the novel observed behavior, since there is a continuous stream of observed state-actions. As shown in Figure 5.8, we do this by keeping all state-actions where an *overall* improvement toward achieving the human's goal was observed (as measured using the character's own fitness function). In other words, we keep the subsequence of state-action pairs starting with the global fitness minimum in the entire sequence, and ending with the pair where the human achieved her goal. Thus our retained subsequence has the property:

$$\text{fitness}(\mathbf{s}_0) \leq \text{fitness}(\mathbf{s}_i), \quad \forall i,$$

where $\mathbf{s}_0$ is the global minimum (i.e., first state in the retained state-action subsequence). Later, when performing simulations during task selection, we allow the entry point into the recorded sequence to vary depending on the current state of the virtual world $\mathbf{s}_c$:

$$\text{entry point} = \arg \min_{\mathbf{s}_i \in \text{sequence}} \|\mathbf{s}_c - \mathbf{s}_i\|.$$

To determine whether an observed behavior is novel to the character, and therefore of interest for mimicking, the recorded state-action sequence must be compared to the decision making in the action selection layer of the character's behavioral model. As shown in Figure 5.9, we do this by performing action selection for a random subset of the states in the recorded state-action sequence. We then compute the average component-wise difference, ε, between the recorded actions and those selected by the behavioral model:

$$\boldsymbol{\varepsilon} = (\sum_{j=0}^{n-1}(\mathbf{a}_j - \tilde{}\mathbf{a}_j))/n.$$

$$\text{if } \|\boldsymbol{\varepsilon}\| \geq \xi, \text{ then } \texttt{Novel}, \text{ else } \texttt{Known}.$$

We have found that $\xi \leq 20\%$ of the max possible error is effective.

Note that for mimicking to work, the observing character must be able to infer the human's goal. In interactive virtual environments, this is usually trivial, since the goal is largely determined by the high-level situation of the virtual world. For example, in our rugby case study, the general goal can be determined by which team has the ball. Obviously, the team with the ball will attempt to score, while the other team will try to stop them. This general, high level understanding of the human's goal is sufficient. For example, if the human's avatar succeeds in running around the opposing team's characters and scoring, then the human's behavior in that situation is a candidate for mimicking by the character when it has the ball.

The character can generalize observed behavior if similar enactments are observed and recorded. This generalization can be done either by performing a weighted blending of two or more recorded action sequences, or by unexpectedly switching from one sequence to another. Similarity between recorded behaviors is determined by attempting to overlap some portion of the recorded state-action sequences by computing the mean component-wise difference between the actions.

Since we assume that behaviors begin with fitness minima, our mimicking method cannot learn all interesting behaviors. For example, there are effective strategies where the human could purposefully do something of poor fitness to misdirect the character. In such situations, it is likely the character will only learn the portion of the behavior after the misdirection. Nevertheless, in our experience most valuable novel behaviors do not

directly violate the character's fitness function, and thus can be learned effectively by our mimicking method.

Relationship To Previous Work

There has long been interest in teaching synthetic agents through observation or demonstration. This is because learning through demonstration is a natural form of instruction used in the real world by humans [Meltzoff and Moore 1992; Yoon 2003]. It is widely believed that demonstration is one of the most natural and effective human-computer interfaces possible, especially for a non-technical user. However, developing an effective and general demonstration interface has proven elusive.

[Blumberg et al. 2002] allows a human user to interactively train a virtual character (e.g., a graphical dog). This technique allows the user to lure the character into a certain position, indirectly demonstrating desirable poses. However, while effective at teaching some aspects of motor control, this technique does not allow teaching of decision-making to achieve tasks and goals. Therefore, this technique does not fulfill our needs. [van Lent and Laird 2001] presents a programmer-oriented technique for learning through demonstration. A programmer must define and implement all *operators* (i.e., actions or tasks) which a character can perform. The post- and pre-conditions of these operators are then learned through demonstration (with explicit annotations provided by the programmer after the demonstration). This technique has proven effective but is not automatic, and therefore cannot be used for on-line adaptation. [Kasper et al. 2001] presents a technique for teaching a robot simple navigational behaviors. However, this is too limited for our needs. In [Price 2002], one agent guesses state-action values by observing the behavior of another agent. While this allows for fine-grain learning, it is not significantly faster than traditional reinforcement learning and is limited to discrete state/action spaces.

In contrast to these previous techniques, our mimicking method is less general but is fully automatic, learns quickly, and can easily encapsulate very complex behaviors. The most novel aspects of our mimicking method include automatic detection of novel behaviors, and the use of a fitness function to automatically determine which novel behaviors may be valuable to imitate.

5.5.5 High-Level Learning (for Goal Selection)

It is at the level of goal selection that some previous master-slave interactive behavioral learning techniques have been applied. This is because the decision making is temporally coarse grain and thus it is plausible for the human to provide timely feedback. Unfortunately, in many cooperative and competitive situations, it is unnatural for the human to provide explicit feedback to the character. We leverage a different type of feedback: *emotion*.

In nature, emotional feedback plays a heavy role in the formation of personality [Matthews 1997]. For example, it is well known that humans will more often engage in activities that make them happy. In algorithmic terms, this means that the desirability (value) of each goal is updated based on the happiness of the character as a result of performing it.

Emotion is often implemented in a behavioral model (as in [Tu and Terzopoulos 1994; Tomlinson and Blumberg 2002; Egges et al. 2004]) as a small set of variables, e.g.: `Happiness`, `Fear`, etc. The character's current emotional state is the combination of all these variables. In our method for high-level adaptation for goal selection, we are only concerned with the change in happiness. There are many possible events that can affect a character's happiness. One of the most well known is success or failure. Note that emotion-changing triggers are part of the behavioral model, and have been examined in previous works, so we do not dwell on them further here. The fact that the character's emotional state changes is sufficient for our needs.

In our emotion-driven adaptation method, we use a highly abstract representation of the character's state: e.g., "`Hungry` + `FoodNearby` → `Eat`". We determine the value (i.e., predicted resulting happiness, ˜`Happiness`) of a goal by maintaining a set of weights which define how important each high-level feature of the abstract state is for a certain goal. The value of the goal for the current state is then computed as the weighted sum of the state features using a linear perceptron:

$$\tilde{}\texttt{Happiness} = \sum_i (\mathbf{s}.i \cdot w_i).$$

In other words, if features `Hungry` and `FoodNearby` are high, and the associated weights are high, then the goal `Eat` will have a high value. The weights are updated once, either

when a new goal is about to be selected or after the current goal has been active for a sufficient amount of time (e.g., 30 seconds). We update the weights using gradient descent, based on the state under which the goal was selected, to more accurately be able to predict the resulting happiness in future iterations:

$$w_i' = w_i + (\gamma \cdot \mathbf{s}.i \cdot (\texttt{Happiness} - \tilde{}\texttt{Happiness})), \quad \forall\, i,$$

where $\gamma \approx 0.5$ is the scalar learning rate, $\tilde{}$Happiness was the predicted happiness, and Happiness is the actual resulting happiness. This allows the character to not only adjust under what circumstances a goal is desirable, but also adjust the magnitude of the desire.

An interesting result of our approach is that similar characters can develop widely varied personalities, depending on their experiences. In fact, a character's personality may diverge from the "optimal" personality with respect to its assigned role in a virtual world. This is because an unlucky character may repeatedly fail at a goal which usually would be accomplished and thereby learn to avoid that goal. This type of learning has an interesting parallel to phobias in real humans. If desired, such divergence can be avoided by either not using our goal-level adaptation method or placing range limits on the weights in the linear perceptron.

Relationship To Previous Work

Our goal-level adaptation method is very similar to previous work in computer animation for character training [Blumberg et al. 2002; Evans 2002], as well as methods for learning from change in emotional state [Tomlinson and Blumberg 2002; Gadanho 2003]. The novel aspect of our approach is the use of emotion to update personality within a multi-level framework that also provides learning for non-reactive behavior. We are not aware of such a combination in the literature.

Our use of a linear perceptron to predict the value of a goal is similar to [Evans 2002]. This approach has proven effective, both in our own experiments and in Evans's work. However, since the function approximation is linear, there can only be one contiguous region of the state space where a certain goal is likely to be selected. However, since we use a highly abstract representation of the state space, this should usually not be a problem.

Note that our adaptation technique for cooperative/competitive relationships can coexist nicely with previous techniques for master-slave relationships. This is because a character may have a cooperative/competitive relationship with some humans and/or characters, and a master-slave relationship with others. For example, to integrate our work and [Evans 2002], both emotional and human-user feedback could be used to train the perceptron.

5.5.6 Using Adaptation in Practice

The accuracy of the learning in our system has proven to be very promising (see Figures 5.11 and 5.12 in the results section). Also, the performance is well within interactive speeds (see Table 5.1), and it has a small memory footprint (usually $\leq$ 2 MB).

Recall that the function of our adaptation system is to supply a behavioral/cognitive model with supplementary information. Therefore, the way this information is used can be unique for any given behavioral/cognitive model. For example, given an n-step prediction of the human's actions into the future, the character could perform an informed tree search to plan its own actions through deliberation. Alternatively, this n-step prediction could be used as extra inputs into a reactive model, even a black box implementation. The information can also be used to cooperate with or compete against the human, as the character wishes.

Our adaptation technique is not limited to small environments with only one human. Indeed, it can operate in very complex environments of many agents (more than one human user, etc). However, for adaptation to perform well, the state space definition must always be reasonably compact. This is because this circumvents the curse of dimensionality, thereby allowing our adaptation technique to be used for interesting, difficult problems. If necessary, it can even be useful to aggressively approximate the current state—even though this limits the accuracy of state information, and thereby limits the potential accuracy of the character's learning, it makes the dimensionality tractable. By keeping the state space small, we have successfully applied our adaptation technique to very complex characters/environments.

To further counteract the curse of dimensionality, we have found it useful to modularize the adaptation when possible. For example, consider a character who can perform

two independent actions simultaneously (e.g., walk in a given direction while looking at something else). We can split this into two separate problems, with the adaptation for each performed separately. This can help simplify both the state and action spaces. Modularization is especially useful for our action prediction method, as it is the most sensitive to the curse of dimensionality of all our learning methods.

In some circumstances, we have found it useful to share acquired knowledge between all adapting characters in a given animation. In other words, we only use a single repository for acquired knowledge, which the adapting characters share. This is useful for reducing storage requirements as well as allowing every character to behave optimally according to what has been learned.

It is important to point out that, while our adaptation technique does change the behavior of a character, it only does so within bounds set by the behavioral/cognitive model. That is, since our adaptation technique only supplies supplementary information, the behavioral/cognitive model is still in full control of decision making. This feature of our technique is important for stability, and maintaining animator goals.

Each of our learning methods provides a unique degree of adaptation for a character. We have determined the individual usefulness of each learning method by applying only one learning method at a time in our rugby case study. Action prediction (low-level learning) is actually the most powerful of all our methods. This is because low-level decision making is most critical in achieving overall life-like and effective behavior since actions are the only decisions directly performed by the character. The second most powerful of our learning methods is mimicking, as it allows a character to rapidly learn novel, complex behaviors which affect both the action and task levels. The third most powerful is mid-level learning for task selection and the least powerful is high-level learning for goal selection.

In some circumstances, it may be desirable to only use a subset of the learning methods presented in this paper (i.e., not apply learning to all layers of a behavioral/cognitive model). This is because, as detailed in the previous paragraph, each learning method provides a different degree of benefit. Also, each method can require a notable amount of CPU to execute (except goal-level adaptation). Moreover, each learning method must be integrated separately into a behavioral/cognitive model, which can be a time-consuming

undertaking. If a subset of methods is to be used, we recommend choosing methods according to the usefulness order given the previous paragraph.

Recall that we utilize a gradient fitness function in our system to guide the character's behavior. It is beyond the scope of this paper to detail methods for creating fitness functions — but this problem has been thoroughly studied, and thus we point the interested reader to the literature. Techniques to (semi)automatically create gradient fitness functions include *potential fields* [Reif and Wang 1999], *value iteration* [Sutton and Barto 1998], and interpolation of discrete state fitness labels. As reported in the literature, even complex problems can often be adequately represented with gradient fitness functions if they are properly abstracted (e.g., in motion synthesis for character animation [Sims 1994; Arikan et al. 2003]). In our case studies reported in this paper (see Section 5.6), we have used explicitly programmed gradient fitness functions, one for each task.

As discussed previously, our learning methods for both action selection and task selection use case-based reasoning, where the case libraries are initialized with regards to "generic" human behavior. This initialization is worth discussing in additional detail here because early character behavior is crucially dependent on this data. These initial cases are created by training the character "off-line" through interaction with one or more human users. In other words, the character gains its initial knowledge through learning to interact with a small set of users that is considered to be representative of the set of all possible users. Thereafter, the character need only adjust its knowledge to more effectively interact with a specific human user.

5.6 Experimental Results

5.6.1 Virtual Rugby

Our first case study is of synthetic human players engaging in a sport such as rugby or American football (see Figure 5.13). This application of our adaptation system to virtual athletics is an interesting challenge; sports in general is known to be a very difficult environment for autonomous synthetic characters [Stone 2000].

In our case study, there is no explicit communication between the characters nor with the human user. Like the human user, the characters must rely on "visual" perceptions to ascertain the current state of the virtual world. Specifically, a character senses physical features such as its location, the distance from it to other characters, velocities, etc. Perception is performed by each character individually, and semi-realistic sensory honesty is enforced (i.e., a character can't see through the back of its head, etc).

The characters' and user avatar's motor control is performed through skeletal animation, based on a library of motion capture data. Specifically, there is a motion capture clip associated with each action a character may perform. These clips are blended together when necessary (using quaternion interpolation) to avoid jittering or discontinuities in the motion.

Action selection is performed at 15 Hz (once per frame), task selection once every 1 to 8 seconds, and goal selection once after every time a tackle or score occurs. The action space is composed of a continuous range of acceleration vectors, which represent the change in running velocity of a player. The human user controls his avatar through a joystick. The characters have several candidate tasks, such as to charge the user, cautiously wait for the user to approach, etc. The only two goals are to score (if the character's team has the ball) or stop the user from scoring.

As detailed in Section 5.4.2, our adaptation technique assumes that the programmer provides compact state space definitions as necessary for each learning method. We now present the state space used for action prediction in this case study. For one-on-one rugby (one character against one human user), the compact state is defined as the relative (i.e., translation invariant) positions of the character and user, and their velocities. Thus the compact action prediction state space is six-dimensional:

$$\mathbf{s}_t = (\Delta x,\ \Delta y,\ V_x^U,\ V_y^U,\ V_x^C,\ V_y^C),$$

where $U =$ "user" and $C =$ "character". We ignore non-critical features, such as closeness to going out-of-bounds, to keep the space dimensionality low. This one-on-one rugby state space can be extended to support many-on-one rugby by including salient information about more than one character. We have found that fully representing the closest character,

partially representing the second-closest character, and ignoring all more-distant characters works well.

We use one state space for all mid-level adaptation and mimicking. The state space is analogous to the one used for action prediction but is smaller, and is from the adapting character's frame of reference. It is composed of the translation-invariant separation of the adapting character and user, and the user's velocity. Thus the task-level compact state space is four-dimensional:

$$\mathbf{s}_t = (\Delta x,\ \Delta y,\ V_x^U,\ V_y^U).$$

We purposefully made this space small so that approximate state-task value learning would occur quickly. When performing task selection, the simulations we run on the most promising tasks/behaviors provide us with sufficient accuracy. Since the simulations involve all characters in the virtual world, both one-on-one and many-on-one rugby work well with the same task-level state space.

The characters are controlled with a cognitive model, which performs decision making through a tree search (using A*). To make searching tractable, a discrete version of the action space is used for deliberation. We support varying goals and tasks by allowing a character's fitness function to vary. One of our fitness functions (for a character that is attempting to tackle the user) is given in pseudo-code in Figure 5.10. This is a gradient fitness function because a fitness is produced for every state, with fitness increasing toward the goal state (tackling the user). Rather than implementing gradient fitness functions algorithmically, a popular approach in the literature is to specify fitnesses for a subset of states and then interpolate. This alternative approach is especially pertinent for use with our adaptation technique, since our state spaces are real-vector-valued and organized so that Euclidean-similar state vectors represent similar physical states.

We performed several experiments in this case study, varying the number of characters on each team. We tested both cooperative relationships between teammates, and competitive relationships between non-teammates. We also performed experiments in which we varied the initial state and the human user's behavior. We gathered statistics on the accuracy of the character's learning, the increase in its success rate with respect to the human user, and the runtime performance of our adaptation system. These results are

```
fitness_tackler(P^U , P^C)
{
    const float PLAYER_SIZE = 0.4;
    float dist = ||P^U − P^C||;
    if (P^U_y > P^C_y + PLAYER_SIZE) {
      /* User has passed character, so she can score easily. */
      return (-100 - dist);
    }
    else if (dist < PLAYER_SIZE) {
      /* Close enough to tackle. */
      return (500);
    }
    else {
      /* Character is in user's way (where it should be). */
      return (100 - dist);
    }
}
```

Figure 5.10: Pseudo-code of our gradient fitness function for a rugby character that will rush and attempt to tackle the human user. $P =$ "position", $U =$ "user", and $C =$ "character". This fitness function specifies that the character should get as close to the user as possible without falling behind him (and thereby allowing the user to score). The character's cognitive model automatically determines how to behave to maximize long-term fitness. This function can also be used to detect valuable behaviors to mimic by measuring the fitness of the human user's own tackling behaviors.

presented in Figures 5.11 and 5.12, and Table 5.1. Demonstrations are given in the supplementary video accompanying this paper (available from `http://rivit.cs.byu.edu/a3dg/publications.php`).

Additional Action Prediction Experiments

We also ran some focused experiments on action prediction, to determine when k-nearest neighbor or minimax should be used to generalize cases. Recall that action prediction provides more benefit than any of our other per-layer learning methods (see Section 5.5.6)—therefore we thought it worthwhile to delve deeper into this specific learning method. In these experiments, we used a simplified, discrete version of our rugby environment. There was no human user, just two characters. The tackler adapted, whereas the ball-runner exhaustively tested all possible behaviors of 7 actions in length. The results of

	Rugby	*CTF*	*Camera*
Action prediction time	30 μ sec	33 μ sec	24 μ sec
Simulation time	28 ms	42 ms	N/A
Total avg. CPU usage	7%	12%	35%

Table 5.1: Typical performance results of our adaptation system in our three case studies (for one given adapting character). We used a 1.7 GHz PC with 512 MB RAM.

	$k = 1$	$k = 2$	$k = 6$	$k = 12$
k-*NN*	69.16 : 1	25.61 : 1	23.69 : 1	29.825 : 1
Minimax	69.16 : 1	239.5 : 1	963.8 : 1	1729 : 1

Table 5.2: Average ratio of tackles to scores for the ball-runner performing all behaviors of length 7 in a simplified, discrete rugby environment. Note that this is a different environment than the continuous world used in the rest of our rugby case study (as shown in Figures 5.1 and 5.13). With no learning for the tackler character, the ratio was only 5.54 : 1.

these experiments are presented in Tables 5.2 and 5.3. While using k-nearest neighbor lets the tackler keep the ball-runner to negative forward progress on average, we found that critical mistakes were sometimes made. Alternatively, using minimax allowed the ball-runner to achieve positive forward progress on average, but very few critical mistakes were made.

5.6.2 Capture The Flag (CTF)

This case study is based on a well-known research test bed called *Gamebots* [Kaminka et al. 2002]. This test bed modifies the popular computer game *Unreal Tournament 2003*, allowing a programmer to replace the built-in behavioral model. In Unreal Tournament, the virtual world is a complex 3D environment of rooms, hallways, stairs, etc. It is populated with two or more players (virtual humans) organized into two teams. The players are armed with "tag guns"; once a player is tagged, he is out for a period of time. The objective is to reach the other team's flag. A slide show of this case study is given in Figure 5.14.

We have modified the Gamebots test bed so that, rather than overriding the characters' standard behavioral model, we can simply provide the characters with auxiliary information and suggestions. It is important to note that what we have done in this case study is added our adaptation system to an existing, professional behavioral model. Integration, while not trivial, proved to be straightforward in most aspects. This provides some additional

	$k=1$	$k=2$	$k=6$	$k=12$
k-*NN*	-0.0553	-0.0625	-0.0629	-0.0444
Minimax	-0.0553	0.00957	0.0314	0.0325

Table 5.3: Average forward progress made by ball runner before end of game for all behaviors of length 7 (in the same simplified, discrete rugby environment used in Table 5.2).

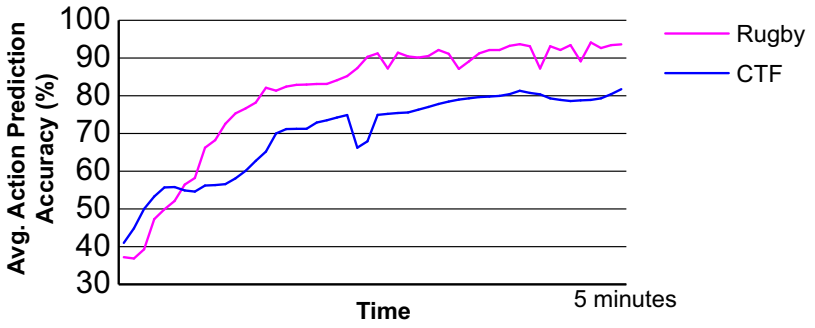

Figure 5.11: Accuracy of predicting the human's actions (L2-norm). This demonstrates the accuracy of both low- and mid-level learning in our system. This experiment started with the character having very incorrect information about the human user.

validation for our claim that our adaptation technique can be integrated into most existing behavioral animation systems.

The Unreal Tournament behavioral model is composed of three layers, named: *Team* (goal selection), *Squad* (task selection), and *Bot* (action selection). A team is composed of one or more squads, and a "bot" is an individual character. What is unique about this behavioral model is that all members of a team or squad share the same *Team* or *Squad* layer instance, respectively. Thus there is unified group decision making. We applied our adaptation technique to all three of these layers. However, the behavioral model is complex enough that many small "component" decisions are made, and we chose to not apply adaptation to a number of these because the possible utility was deemed to be too small in comparison to the workload of integration. If the behavioral model were implemented with adaptation in mind, integration would likely be easier and more complete.

The most difficult portion of integration was the task-level learning method. Specifically, it proved challenging to run internal simulations to determine the utility of candidate tasks. This was difficult because the Unreal Tournament behavioral model is tightly coupled with the rest of the software and could not easily be decoupled to allow "hidden"

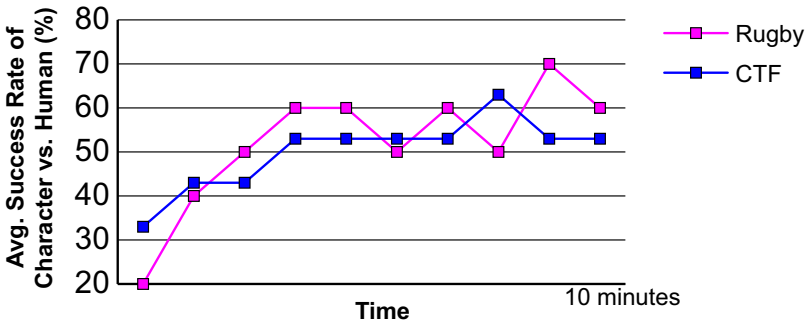

Figure 5.12: This graph demonstrates the effectiveness of our adaptation system as a whole. For example, in the rugby case study, how often the character scores compared to how often the human user scores.

executions of the behavioral model, which would not be reflected visually to the human user. Due to these complexities, we implemented a very simple version of the environment for use in running simulations. This proved sufficient in our experiments, although the characters occasionally made critical mistakes in their decision making.

The compact state spaces we use for adaptation in this case study are quite similar to those in our rugby case study. The notable differences are that only the nearest opponent and no teammates are represented in the current state, and we supply approximate information about nearby obstacles in the virtual environment. All nearby obstacles are represented by a single mean angle, θ (oriented around the "up" direction), representing the average direction toward the obstacles according to the character's or user's frame of reference. Assuming the character will never be in a very narrow hallway or room, this angle will be valid since all nearby obstacles will have surface normals pointing in the same general direction. Thus, for action prediction, the compact state space is defined as:

$$\mathbf{s}_t = (\Delta x,\ \Delta y,\ V_x^U,\ V_y^U,\ V_x^C,\ V_y^C,\ \theta).$$

The dimensionality of this state space is greater than the space used for action prediction in our rugby case study. As a result, adaptation is somewhat slower than in our rugby case study. Moreover, because we use such a crude approximation of the complex virtual environment, action prediction is of a lower accuracy than in rugby. Nevertheless, our results are still promising, suggesting that our adaptation technique scales sufficiently to be useful for complex environments and characters.

The results of this case study are presented in Figures 5.11 and 5.12, and Table 5.1. A slide show of one contiguous animation is given in Figure 5.14. Additional examples are given in the supplementary video accompanying this paper (available from `http://rivit.cs.byu.edu/a3dg/publications.php`).

5.6.3 Automated Cinematography and Attention Selection

Usually, the actions taken in behavioral animation are movement. However, our adaptation technique is not limited to applications in movement or navigation. In this case study, we examine a very different use for adaptation: automatic selection of where a virtual camera or character's attention should be directed. Autonomous camera control and attention selection have been topics of interest in computer graphics for years (e.g., [He et al. 1996; Gillies and Dodgson 2002]).

There is a notable challenge in automatic camera and attention control: the system must be able to accurately predict the future actions of all agents in the environment. For example, in cinematography, this is necessary to achieve natural and aesthetically pleasing camera cuts between views (e.g., cut *before* two participants begin to interact). For attention selection, this is helpful in achieving intelligent-looking eye movement and in ensuring that no critical sensory information is missed. Therefore it is important that user and/or character behavior be predicted accurately.

In this case study, we performed an experiment where the camera automatically placed itself within a dynamic scene of many characters who either milled about or stopped to talk to each other. We only used the action prediction portion of our adaptation technique, and achieved good results. Note that, as shown in Table 5.1, total CPU usage was higher in this case study than the others because there were many characters for which to predict actions.

5.7 Summary and Discussion

We have presented a novel technique that enables autonomous cooperative/competitive virtual characters to quickly adapt on-line due to interaction with a human user. Our system

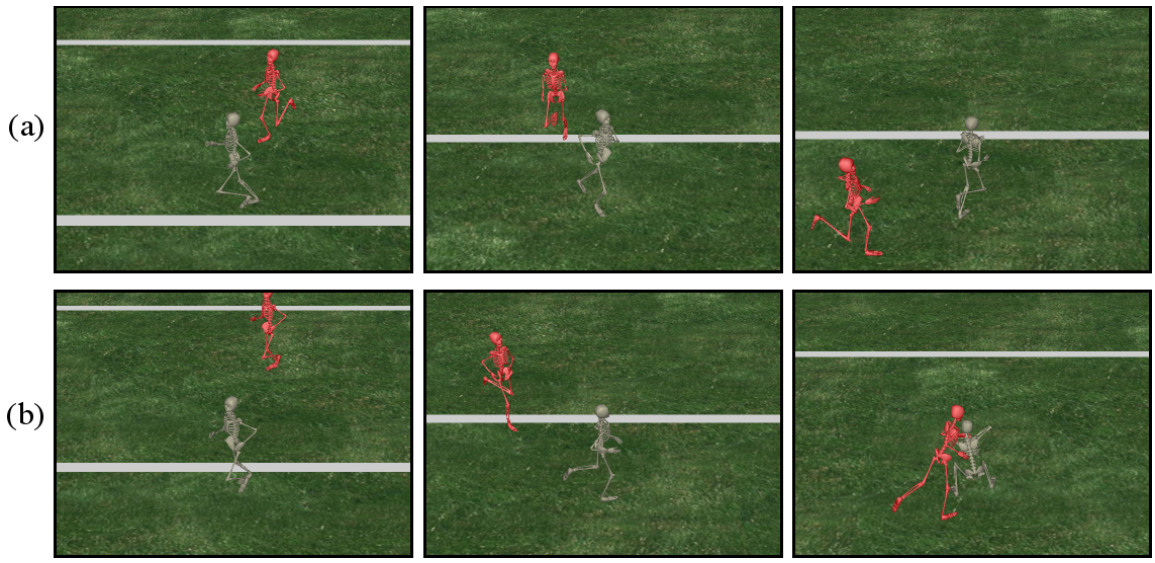

Figure 5.13: (a) The human user (brown skeleton) performs a loop, which succeeds in getting past the character (red skeleton). As a result, the human can score. (b) Now that the character has adapted, the next time the human attempts a loop it predicts the human's actions and tackles him.

is composed of a small set of independent learning methods, which are applied individually to the layers of a behavioral/cognitive model. Our system is designed around a common behavioral animation framework, and thus can be used with most existing behavioral animation systems. Our system fully supports both reactive (behavioral) and deliberative (cognitive) decision making, in discrete or continuous state/action spaces. Our contribution in this paper is important because we present a solution for a previously unsolved problem: fast adaptation for cooperative/competitive virtual characters. Adaptation is an important problem for many interactive graphical applications, such as training simulators, computer games, etc.

As discussed throughout this paper, the layered approach we have taken to interactive adaptation is logical with regards to current trends in diverse but related fields (e.g., behavioral animation, machine learning, multi-agent systems, etc). There is also interesting validation from psychology, where researchers postulate that the best approach to model human cognition is in computational layers [Newell 1990].

Our knowledge-gathering approach to adaptation can be seen as hitting a "sweet spot" between nature vs. nurture. This is because the character begins with a fundamental skill

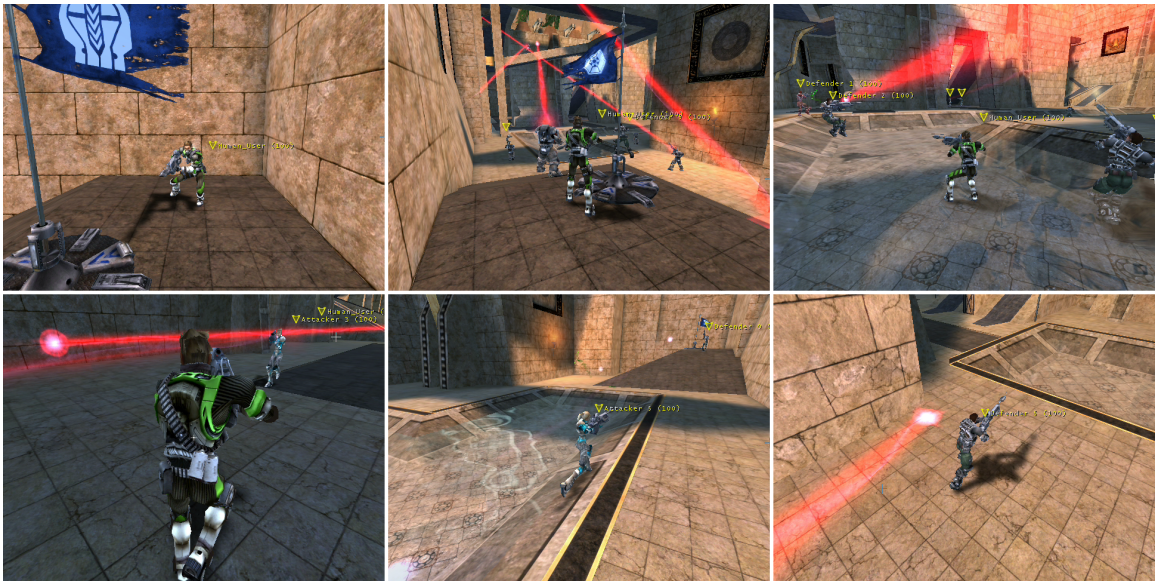

Figure 5.14: Snapshots of a single Capture The Flag (CTF) animation. The human user (in green and black) is on the defending team. They are defeated by the attacking team (characters in blue and gray), who have adapted to the human's tactics and learned to work well together as a team. The last remaining defender, who has adapted to become a coward, runs away.

set (motor control, perception, and decision making), and then gathers knowledge in an on-line fashion. The character then uses this knowledge to more optimally interact with the human user.

An interesting benefit of our technique is that, since a character can adapt on-line, it can fill "gaps" in its behavioral model. In other words, a programmer does not have to carefully construct the behavioral model such that it will immediately handle every possible situation properly. This can also make a behavioral model more robust. Further, in environments where there is no pareto-optimal Nash equilibrium (i.e., no single best strategy), adaptation may be necessary to achieve and maintain good behavior.

However, while our multi-level adaptation technique has proven to work well, there are some weaknesses that are important to recognize. First, while knowledge gathering is very fast, using that knowledge does require a notable amount of CPU. As a result, it may not be plausible to have many adapting characters in the same interactive animation. Second, integrating our adaptation technique into an existing behavioral/cognitive model is

not trivial but appears to usually be straightforward. Of course, integration will be easier with new behavioral/cognitive models that are implemented with adaptation in mind. Third, the success of our adaptation technique for a given environment/character depends on the programmer supplying an adequate compact state space representation and fitness function. Therefore, to effectively use our adaptation technique, the programmer may have to develop some new skills. Finally, since we apply discrete learning methods to different layers, there is no smooth "inbetweening" for additional intermediate layers. However, note that any layer of a behavioral/cognitive model can use one of our learning methods, based on the temporal granularity of its decision making (see Figures 5.3 and 5.4). Therefore, our adaptation technique can be applied to most existing behavioral animation systems and autonomous characters.

Although our technique has proven effective in our case studies, there is no guarantee that it will be effective for every imaginable character and environment. However, as long as our assumptions in Section 5.4.2 are met, we believe that our technique will work well for nearly all characters and environments. This is because the requirements of the underlying learning methods (and the system as a whole) will be met.

Another possible application of our adaptation technique is for one character to learn to adapt to another virtual character. This is interesting because it can result in very natural-looking behavioral animation, as realistic learning is reflected as the animation proceeds. Note that this use of adaptation is also applicable to off-line animation, as human interaction is not required. Another possible use of our technique is the creation of entirely new behavioral models in an on-line fashion by leveraging our work in this paper to perform learning through demonstration. We can use the state-action model of the human's action selection to determine the decision making of a character. One drawback to this approach is that the decision making is somewhat shallow.

Part IV

Creating Behavior Through Demonstration

As discussed in the introduction of this dissertation, the design and programming of behavioral/cognitive models is very difficult and time-consuming. In Part II we presented techniques for simplifying the construction of behavioral/cognitive models by generating state-action pairs through planning. However, while this approach is useful in certain circumstances, it lacks scalability, it is difficult to achieve specific styles of behavior, and the required fitness function can only be specified by a programmer.

Part IV presents two chapters on simplified construction of behavioral models through programming by demonstration (i.e., learning by observation). This addresses problem Problem #3 listed in Chapter 1.

Chapter 6 introduces a technique for learning policies from human example. This technique is related to existing methods in the robotics and agents literature but is specifically applied to behavioral animation and includes a novel conflict elimination algorithm. This paper is currently under review by *Journal of Graphics Tools*.

> Jonathan Dinerstein, Trent Crow, and Parris K. Egbert. "Intelligence capture — Automatic behavioral animation from human example". To be submitted, 2005.

Chapter 7 discusses how autonomous virtual character behavior can be specified and synthesized in a data-driven manner. Sequences of actions are automatically captured from human demonstration. This data is then used to synthesize novel behaviors by "cutting and pasting" disjoint segments of the demonstrated action sequences. This data-driven approach is interesting because it has been empirically shown to be very scalable, intuitive,

and powerful. This paper has been submitted to IEEE Transactions on Visualization and Computer Graphics and is currently under review.

> Jonathan Dinerstein, Parris K. Egbert, Dan Ventura, and Michael Goodrich. "Data-driven programming and control for autonomous virtual characters". Submitted to *IEEE Transactions on Visualization and Computer Graphics*, April 2005.

Chapter 6

Intelligence Capture — Automatic Behavioral Animation from Human Example

To be submitted, 2005.

Abstract: This paper presents *intelligence capture*, a novel technique for programming behavioral animation by demonstration. This technique operates in two modes: *training* and *autonomous behavior*. In training mode, the human user has direct control over the virtual character through an input device such as a joystick. The human user demonstrates the desired behavior for the character by dictating its actions during an interactive animation. This demonstration is recorded as a set of state-action pairs, each pair representing the specific action the human chose for the character when in the associated state. This data is then processed to ensure that any conflicts in the pairs are eliminated. Later, when the character is to behave autonomously, the recorded state-action pairs are generalized to form a continuous state-to-action mapping. This mapping dictates the action the character is to perform for any given state. Thus a behavioral model can be automatically constructed by an animator-programmer team in an intuitive manner, eliminating a programming bottleneck and making this process simpler and quicker. Moreover, the animator has greater control over the creation of the behavioral model, and stylized behavior can easily be realized. While not a panacea, intelligence capture can effectively produce many types of behavioral animation.

6.1 Introduction

In *behavioral animation* [Millar et al. 1999], virtual characters are designed to be autonomous agents. If given sufficient intelligence, the characters can animate themselves by choosing actions to perform (where each action is a motion for the character). This is often done by using a library of motion capture data, each motion clip being associated with an action the character may select. Behavioral animation has become popular for interactive virtual worlds where a character's exact behavior cannot be dictated *a priori* (e.g., computer games, training simulators, etc). Behavioral animation has also been used in film production, in particular for large groups or crowds of characters.

Despite the success of behavioral animation in certain domains, some important arguments have been brought against current techniques [Isla and Blumberg 2002; Devillers et al. 2002]:

1. Behavioral models can often be very difficult and time-consuming to design and program.

2. Human decision making can be difficult to quantify realistically into a model.

In this paper we present *intelligence capture*, a general and reusable technique for automatic machine learning of a behavioral model by mimicking demonstrated human behavior. This technique not only provides a natural and simple method for the construction of behavioral models, but it also allows an animator to be intimately involved in the construction of the behavioral model. While not a panacea, intelligence capture can effectively produce many categories of stylized intelligent behavior very quickly in an intuitive manner.

First, the human user demonstrates a behavior for the character by controlling the character during an interactive animation. This demonstration is recorded as a set of state-action pairs, each of which represents how the character should react to a given configuration of the virtual world. This data is then processed to ensure that any conflicts in the examples are eliminated. Later, when the character behaves autonomously, the recorded state-action pairs are generalized to form a continuous state-to-action mapping. This mapping dictates the behavior of the character (i.e., how it should respond to any given state

it may encounter). This mapping can be created through any continuous form of interpolation. We use machine learning techniques (either k-nearest neighbor or support vector machines (SVM) [Mitchell 1997]) that are well-known and robust.

6.1.1 Previous Work

Several notable techniques for behavioral animation have been developed, e.g., [Reynolds 1987; Funge et al. 1999; Monzani et al. 2001; Blumberg et al. 2002; Gillies and Dodgson 2002; Egges et al. 2004]. Unfortunately, all of these techniques require that a behavioral model be explicitly programmed. As discussed in the introduction, this is a difficult task.

There has recently been interest in reducing these problems through the use of machine learning. In [Blumberg et al. 2002; Evans 2002; Dinerstein and Egbert 2005], techniques are presented for allowing a character to adapt its behavior online. However, these techniques still require that a full behavioral model be explicitly programmed. Another technique, presented in [Dinerstein et al. 2004b], allows a character to learn a behavioral model offline given only a fitness function. However, this technique has trouble learning highly complex behavior, and it is difficult to achieve stylized behavior. A technique developed in the computer game community is [Alexander 2002], where high-level features of the desired behavior are learned through demonstration. Unfortunately, since this technique only learns at a high level, it cannot produce complete behavioral models (merely parameters for the behavior).

Some research has been performed in using machine learning for function approximation to aid in computer animation. For example [Grzeszczuk et al. 1998], where physically-based animation is accelerated through approximation of the state transitions. In [Dinerstein and Egbert 2004], deliberative behavioral models are sped up through approximation of the decision making. Another example is [Faloutsos et al. 2001], where the proper situations in which to use different skeletal motor controllers are learned. In contrast to these techniques, we use machine learning to approximate explicitly demonstrated human decision-making. Specifically, state-action pairs extracted from observation of human behavior are used to infer a policy.

The agents and robotics communities have long recognized the need for simplified

programming of agent AI. There has been some interesting work performed in *programming by demonstration* (e.g., [van Lent and Laird 2001; Kasper et al. 2001; Price 2002; Nicolescu 2003]), where an agent is instructed on what to do through demonstrations by a user. Our intelligence capture technique fits in this category but is specifically designed for agents that are virtual characters. The existing technique that is most related to our work is [Kasper et al. 2001], where a robot (using a neural net) learns a *state* $\rightarrow$ *action* mapping from demonstration. However, our technique is unique in many aspects, in particular because it performs conflict elimination and is applied within the unique constraints of computer animation. We will discuss in more detail later why these contributions are significant.

Contribution We present *intelligence capture*, a novel technique for programming behavioral animation through demonstration. To our knowledge, this is a previously unexplored approach to behavioral animation. With regards to previous work, we make the following notable contributions in this paper:

- The application of agent programming-by-demonstration concepts to behavioral animation.
- A novel conflict elimination method (previous techniques in agents/robotics have merely ignored conflicts).

6.2 Intelligence Capture

6.2.1 Overview and Formulation

Our intelligence capture technique helps overcome the difficulties inherent in programming behavioral models by providing an intuitive programming-by-demonstration interface. Thus an animator-programmer team can more easily create a behavioral model as compared to a programmer using traditional methods. Intelligence capture is summarized in Figure 6.1. It includes the following steps:

1. *Train:*

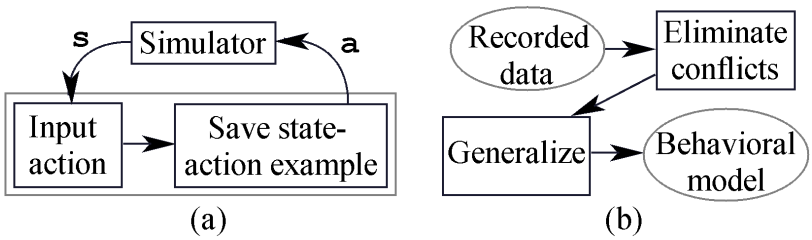

Figure 6.1: The workflow of intelligence capture. (a) The simulator, which drives the animation, provides a current state $\mathbf{s}$. This is visualized for the animator. The animator responds with an action $\mathbf{a}$, which is recorded and then used to update the simulation. (b) Once a sufficient set of state-action examples have been recorded, they are processed to eliminate conflicts and then generalized into a continuous policy function. This policy is used online as our behavioral model.

(a) Observe and record state-action pairs.

(b) Eliminate conflicts.

2. *Autonomous behavior:*

(a) Generalize recorded state-action pairs into a policy μ.

(b) Use μ to compute decisions for the character, once per fixed time step Δt.

Intelligence capture is a process of learning to approximate intelligent decision making from human example. We formulate decision making as a state-to-action mapping, or *policy*:

$$\mu : \mathbf{s} \rightarrow \mathbf{a}, \tag{6.1}$$

where $\mathbf{s} \in \mathbb{R}^n$ is a compact representation of the current state of the character and its world, and $\mathbf{a} \in \mathbb{R}^m$ is the action chosen to perform. Each component of $\mathbf{s}$ is a salient feature defining some important aspect of the current state. If the character has a finite repertoire of possible actions, then $\mathbf{a}$ is quantized.

A state-action pair is denoted $\langle \mathbf{s}, \mathbf{a} \rangle$. The human demonstrator's behavior is sampled at a fixed time step Δt, which is equal to the rate at which the character will choose actions when it is autonomous. Thus the observed behavior is a set of discrete cases, $B = \{\langle \mathbf{s}, \mathbf{a} \rangle_1, \ldots, \langle \mathbf{s}, \mathbf{a} \rangle_q\}$. Each pair represents one case of the target behavior. There is

no ordering of the pairs in the set. We construct μ by generalizing these cases. Specifically, to ensure that the character's behavior is smooth, we construct μ by interpolating the actions associated with these cases. Because $\mathbf{s} \in \mathbb{R}^n$ and $\mathbf{a} \in \mathbb{R}^m$, we can theoretically use any continuous real-vector-valued interpolation scheme. In our implementation, we either use continuous k-nearest neighbor or SVM, as detailed later in Section 6.2.3.

For generalization of cases to succeed, it is critical that the state and action spaces be organized by the programmer such that similar states usually map to similar actions. In other words, $\|\mathbf{s}_i - \mathbf{s}_j\| < \alpha \Rightarrow \|\mathbf{a}_i - \mathbf{a}_j\| < \beta$, where α and β are small scalar thresholds and $\|\cdot\|$ is the Euclidean metric. Certainly, this constraint need not always hold, but the smoother the mapping the more accurate the learned policy μ will be. Moreover, it is important for the programmer to design the state space such that the dimensionality n of $\mathbf{s} \in \mathbb{R}^n$ is as small as possible. This is due to the curse of dimensionality, a famous thesis in machine learning stating that the difficulty of learning a mapping increases exponentially with each additional input dimension [Mitchell 1997].

For more information on designing effective state and action spaces, see the experimental results section of this paper, or the literature cited in the previous work section (e.g., [Reynolds 1987; Grzeszczuk et al. 1998; Dinerstein et al. 2004b; Dinerstein and Egbert 2004]).

6.2.2 Training

Intelligence capture operates in two modes: *training* and *autonomous behavior*. These modes correspond to whether the character is currently learning how to behave or is acting autonomously based on what it has already learned. Usually, training will occur once, following by unlimited uses of the trained character.

In the training mode, both the animator and programmer need to be involved. First, the programmer integrates intelligence capture into an existing (but thus far "brainless") character. This integration involves designing the state and action spaces such that μ will be learnable. Once integration is complete, the animator is free to create behavioral models for the character at will. In fact, the creation of multiple behavioral models for one character may be interesting to achieve different stylized behaviors, etc.

Training by the animator proceeds as follows. The character and its virtual world are visualized in real-time, and the animator has interactive control over the actions of the character (e.g., through the keyboard, joystick, etc). Note that the continuous, real-time presentation of state information to the animator is critical to make the character training process as natural as possible, as this is analogous to how humans naturally perceive the real world. As the simulation-visualization of the character and its world proceeds in real-time, the animator supplies the character with the actions it is to perform. This information is saved in the form of state-action pairs. Once enough state-action examples have been collected, all conflicting examples are automatically eliminated, as described below. We now have a discrete but representative sampling of the entire policy function μ. Moreover, because the demonstrator has had control of the character, she has forced it into regions of the state space of interest — therefore the density of the sampling corresponds to the importance of each region of the state space.

Once demonstration is complete (i.e., enough state-action examples have been collected), all conflicting examples are automatically eliminated. Elimination of conflicts is extremely important. This is because human behavior is not always deterministic, and therefore some examples will likely conflict (i.e., for a given state, more than one action may be proposed). This is an important issue, because machine learning will "average" conflicting examples, creating a new action for that state. This can result in unrealistic or unintelligent-looking animation. For example, consider a car driver who sometimes turns either left or right to avoid an obstacle in the road. If these actions are averaged, this could result in driving straight into the obstacle. Therefore, to ensure that the learned policy is true to the human trainer's behavior, it is important to eliminate all conflicts in the examples.

Conflicting examples are formally defined as:

$$\text{if } \|\mathbf{s}_i - \mathbf{s}_j\| < \nu \text{ and } \|\mathbf{a}_i - \mathbf{a}_j\| > \upsilon, \text{ then } \texttt{conflicting}, \tag{6.2}$$

where ν and υ are scalar thresholds. In other words, if two cases have similar states but notably different actions, they are considered to be conflicting. To eliminate conflicts, we cannot simply arbitrarily delete cases involved in a conflict — this can lead to high

frequencies in the policy. Rather, we must ensure that each example is not an outlier with respect to its neighborhood in the state space. Our goal is to remove those examples that represent high frequencies. We define a neighborhood as the l cases closest to the current example in the state space. We define an outlier as an example whose action is significantly different from the median action of the examples in its neighborhood.

Pseudo-code for our conflict elimination technique is given in Figure 6.2. To compliment this pseudo-code, we now describe our technique. In brief, each state-action pair is tested in turn to determine whether it is an outlier. The current pair is denoted $\langle \mathbf{s}, \mathbf{a} \rangle'$. First, the l neighbors of the current example are found and their median action $\mathbf{a}_{vm}$ is computed, using the following vector-median method [Koschan and Abidi 2001]:

$$\begin{aligned} &\mathbf{a}_{vm} \in \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_l\}, \\ &\text{where } \sum_{i=1}^{l} \|\mathbf{a}_{vm} - \mathbf{a}_i\| \leq \sum_{i=1}^{l} \|\mathbf{a}_j - \mathbf{a}_i\|, \quad j = 1, 2, \ldots, l. \end{aligned} \tag{6.3}$$

In other words, $\mathbf{a}_{vm}$ is equal to the action of the neighbor which is the closest to all other actions of the neighbors according to the Euclidean metric. Finally, if $\|\mathbf{a}_{vm} - \mathbf{a}'\| > \eta$, then the case is an outlier and is marked for deletion. Marked cases are retained for testing the other cases, and then are all deleted as a batch at the conclusion of the conflict elimination algorithm. In our experiments, we have found that it works well to use $l = 5$. If there are not that many neighboring examples within a reasonably small region of the state space (e.g., 15% per axis), we assume that this portion of the state space is undersampled and do not consider deleting this example. To determine outliers, we use a threshold η of about 10% of the possible range of component values in $\mathbf{a}$.

Note that a behavioral model can be incrementally constructed by adding examples, testing it, then adding further examples to it, and so forth. Also, a model can be naturally edited in a simple fashion by adding new examples that conflict with existing examples; elimination of conflicts will then naturally delete examples that do not correspond with the most desired behavior. Even stronger editing can be achieved by explicitly deleting any local state-action pairs that disagree with newly observed and recorded pairs.

For each recorded pair $\langle \mathbf{s}, \mathbf{a} \rangle'$ ...
    Find l closest neighbors of $\langle \mathbf{s}, \mathbf{a} \rangle'$
    if (not l close neighbors)
        Skip $\langle \mathbf{s}, \mathbf{a} \rangle'$
    Compute median action $\mathbf{a}_{vm}$ of l neighbors using Equation 6.3
    if ($\|\mathbf{a}_{vm} - \mathbf{a}'\| > \eta$)
        Mark $\langle \mathbf{s}, \mathbf{a} \rangle'$
Delete all marked pairs

Figure 6.2: Conflict elimination algorithm.

6.2.3 Autonomous Behavior

Once an adequate set of state-action pairs has been collected, we must construct the continuous policy, $\mu : \mathbf{s} \rightarrow \mathbf{a}$. We do this through one of two popular machine learning techniques: k-nearest neighbor or SVM. We have chosen these two techniques because they are powerful and well-established, yet have contrasting strengths and weaknesses.

The continuous k-nearest neighbor (k-nn) algorithm is one of the most well-established and popular machine learning techniques [Mitchell 1997]. In this technique, examples of the target function are retained explicitly. Then, to compute new values of the function, the k examples nearest to the input vector $\mathbf{s}$ are distance-weighted and averaged:

$$\mathbf{a} = \frac{\sum_{i=1}^{k} (w_i \cdot \mathbf{a}_i)}{\sum_{i=1}^{k} w_i}, \quad \text{where } w_i = \frac{1}{\|\mathbf{s} - \mathbf{s}_i\|_p^2}. \tag{6.4}$$

$p \in \mathbb{R}^+$ represents the L_p-norm distance metric used (usually, $p = 2$). Thus k-nn is a local approximation technique. The primary strength of k-nn is that there are strong guarantees about its accuracy, as it merely interpolates local cases (e.g., it can robustly handle non-smooth mappings, and the outputs will be within the convex hull of the k local cases). Another notable strength is that, based on recent research [Aggarwal et al. 2001], we know that k-nn can remain robust for high-dimension problems when using small values of p (e.g., $p = 0.1$). Therefore we use k-nn whenever the demonstrated policy is non-smooth or of high dimensionality. K-nn has proven quite robust for our application. However, k-nn does have some weaknesses in our application, such as a large memory footprint ($\sim$1 MB per policy), and more state-action cases are required than with SVM due to weaker generalization.

The support vector machine (SVM) [Mitchell 1997] is an interesting alternative to k-nn due to disparate strengths and weaknesses. SVM is a learning technique that produces an artificial neural network. It is a compact and global technique, because the entire network contributes in computing an answer. As a result, it performs powerful generalization (with guarantees of no overfitting), but can struggle with highly non-smooth mappings. Moreover, SVM guarantees optimal neural net learning; i.e., it converges to the global minimum mean-squared-error with probability 1. We have found SVM to be useful when μ is smooth, especially when it is C^0 or C^1 continuous. However, such a smooth mapping usually requires that the programmer define excellent state and action spaces. This can prove challenging in practice, and therefore we have found k-nn to be a better choice in the majority of cases since it is more robust for non-smooth mappings.

Due to the inherent challenge in designing smooth mappings, a great deal of work has been performed in *feature construction* (the creation of effective input features from raw inputs) and *feature selection* (the selection of a minimal subset of candidate input features) [Guyon and Elisseeff 2003]. However, much of this work is still at a research (not yet practical) stage. Nevertheless, some techniques have matured to fruition. Some plausible approaches for feature selection for *state* $\rightarrow$ *action* mapping are discussed in [Dinerstein and Egbert 2004]. However, at this point in time, no algorithm has proven better than the human brain at designing effective mappings. As a result, in this paper we focus on the traditional manual approach.

The components of our system (state-action capture facility, conflict elimination, and machine learning tool), if implemented in a sufficiently general manner, are fully reusable. Thus the intelligence capture system is an inexpensive, portable tool. All that must be done to use this tool is for a programmer to develop the character and its virtual world (work that must be done anyway for behavioral computer animation), and then integrate the existing intelligence capture components. An animator then has free reign to create and test behavioral models at will.

6.3 Experimental Results

We implemented our intelligence capture technique and used it to perform a series of experiments. These experiments were designed to cover, in a general fashion, most of the major distinguishing aspects of popular uses of behavioral animation. Between our favorable experimental results, and the results from the agents/robotics literature on programming-by-demonstration (see the previous work section), we have some empirical evidence that intelligence capture is a viable tool for creating some popular types of behavioral models.

When using k-nn in our experiments, we used $k \in [2,7]$ and 32-bit single-precision floating point accuracy. We also assigned an importance-weighted scale to each axis of the state space, determined by that axis' importance for calculating distances (these scales can be computed automatically as discussed in [Mitchell 1997]). For SVM we used 64-bit accuracy, and all target function inputs and outputs were normalized to have zero means and unit variances. All of these experiments were performed on a 1.7 GHz processor with 512 MB RAM.

The results we achieved in our experiments are summarized in Table 6.1.

***k*-nn**	
Number of examples required for k-nn	6,000
Animator time spent capturing all examples	8 minutes
Disk space required for initial examples	$\leq$ 2 MB
Time to eliminate conflicts	16 seconds
RAM required for final examples	$\leq$ 1 MB
Time to compute new target function values	13 microseconds

SVM	
Time to train an artificial neural network	2.5 minutes
Time to compute new target function values	2 microseconds

Table 6.1: Summary of average usage and performance results (with a 1.7 GHz processor, 512 MB RAM, $k \in [2,7]$, kd-tree).

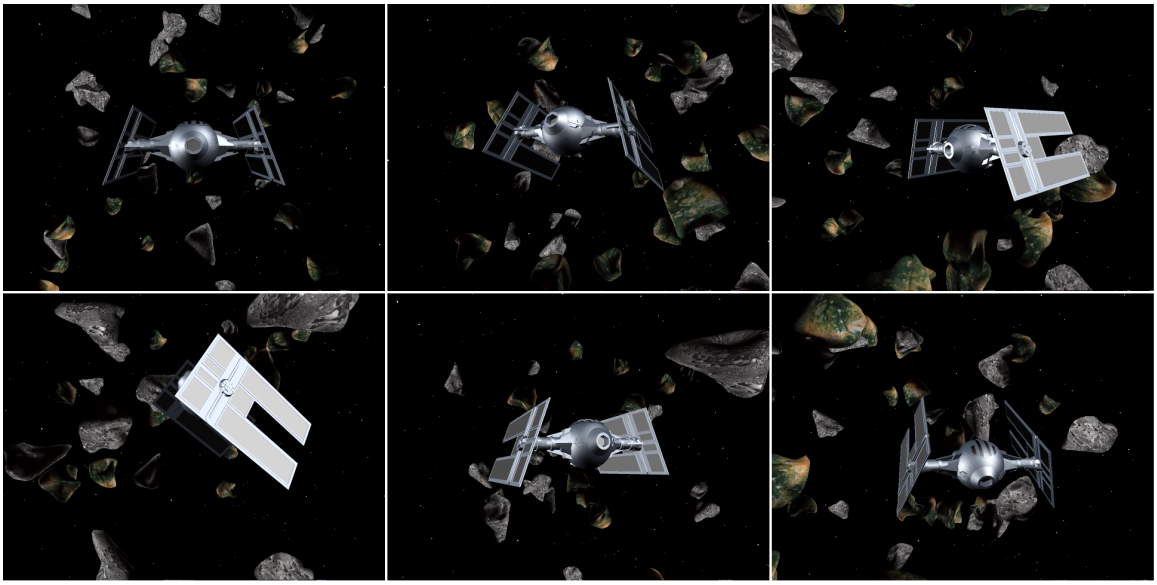

Figure 6.3: A spaceship pilot (autonomous virtual character) intelligently maneuvers within an asteroid field. Its animation is guided by a behavioral model constructed through intelligence capture. The pilot's goal is to cross the asteroid field (forward motion) as quickly as possible with no collisions.

6.3.1 Spaceship Pilot

In our first experiment, the virtual character was a spaceship pilot (see Figure 6.3). The pilot's task was to maneuver the spaceship through an unknown asteroid field, flying from one end of the field to the other as quickly as possible with no collisions. To ensure that this task would be difficult, we limited the maneuverability of the spaceship so that the pilot would have to plan his path through space well in advance. We also placed the asteroids close together. The animation ran at 15 frames per second, with an intelligent action computed for each frame. The virtual pilot had two controls over the spaceship: yaw (rotation around the Y-axis) and pitch (rotation around the X-axis). The state and action spaces were real-valued (i.e., continuous). Thus behavioral animation was performed at a fairly low level, with action selection being fine grain. Intelligence capture was performed by rendering the spaceship as a wireframe (so asteroids beyond it could be seen by the human animator), and controlling it through a joystick.

μ was formulated as follows. The target function inputs were the spaceship's current orientation (θ, ϕ), and the separation between the spaceship and the two nearest asteroids

($\mathbf{p}_s - \mathbf{p}_{a1}$ and $\mathbf{p}_s - \mathbf{p}_{a2}$). Thus there were 8 inputs total: $\mathbf{s} = (\theta, \phi, x_s - x_{a1}, y_s - y_{a1}, z_s - z_{a1}, x_s - x_{a2}, y_s - y_{a2}, z_s - z_{a2})$. There were two outputs, which determined the change in the spaceship's orientation: $\mathbf{a} = (\Delta\theta, \Delta\phi)$. All of these inputs and outputs were continuous spatially. Temporally, μ was computed discretely according to Δt (15 Hz).

The k-nearest neighbor algorithm used about 10,000 examples online, requiring about 0.4 MB of RAM. These examples took about 10 minutes to capture. Note that it is because of the low-level, temporally fine-grain operation of this behavioral model that so many examples were required (a behavioral model applied at a higher, coarser level requires far fewer examples, as discussed in our next experiment). We achieved good results in this experiment of intelligence capture, as shown in Figure 6.3 and in the accompanying video that can be found at http://rivit.cs.byu.edu/a3dg/publications.php. The behavioral model worked well for all random asteroid fields we tried, and the animation was human-like, intelligent, and aesthetically pleasing. SVM also performed well, producing even smoother animation while requiring fewer examples. However, it proved to be more sensitive to the choice of state space (it often failed to produce a good behavioral model when given a suboptimal state space).

To gain a point of reference, we also implemented an explicit behavioral model for our spaceship pilot. Programming this explicit model took over a week (and we are somewhat experienced at developing behavioral models). In contrast, constructing a behavioral model through intelligence capture took a matter of minutes once the intelligence capture class was integrated.

6.3.2 Crowd of Articulated Human Characters

In our next experiment we created behavioral models to control groups of articulated human characters (see Figure 6.4). The behavioral model operated at a high level in the animation hierarchy, controlling the decision making of the character (i.e., "turn left," "walk forward," etc), while the nuts-and-bolts animation was carried out by a traditional skeletal system. Thus the behavioral model only specified a small set of discrete actions (the real-valued policy output was quantized to be discrete). Cloth animation (for the characters'

clothing) in this experiment was performed using a standard spring system. We created several crowd animations. In each animation, all characters used the same behavioral model, showing the variety of behavior that can be achieved. Note that each behavioral model we constructed only required a few minutes to capture all necessary state-action examples, and only a few thousand examples were required for use online.

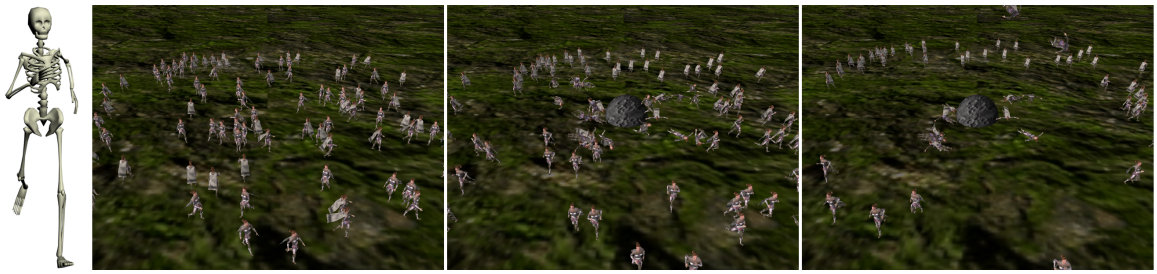

Figure 6.4: A crowd of articulated human characters who intelligently move about. When a boulder falls from the sky, the characters panic and run. Action selection is performed by the behavioral model, and motor control by a traditional skeletal animation system.

One of our primary goals in this experiment was to show that intelligence capture can be used for a crowd of interacting characters. To do this, we performed several brief intelligence capture iterations, randomly placing some static characters in a scene. The human trainer then guided the learning character through the scene. Thus the character learned how to behave in response to other agents around it in arbitrary positions. Ultimately, the character learned behavior such as to not walk into another character, without the need for explicit collision detection.

We created two policies. The first was used before the boulder impact, and the second afterwards. In the first policy (pre-boulder), the environment was discretized into a 2D grid, and the character could walk to the three adjacent squares ahead of it, or turn in preparation for walking to a another adjacent square. **s** was formulated as the current orientation of the character, and the separation between it and the next three closest characters. Thus **s** had seven components. **a** was formulated as a single component, over which the five possible actions were distributed in the following order: turn left, forward-left, forward,

forward-right, turn right. Decision making was computed at a rate Δt of about twice per second. After the boulder hit the ground, we switched to another policy (with continuous state/action spaces) that simply directed the characters to run straight away from the boulder.

6.4 Discussion

It is important to note that intelligence capture is not a simple "tape recording" of computer animation (like motion capture). Rather, since intelligence capture constructs a continuous policy function, an infinite number of unique animations can be generated using a single captured behavioral model.

Intelligence capture is pertinent for all currently popular uses of behavioral animation, including: film, computer games, and training simulators. Intelligence capture works well with existing animation techniques, because it merely provides a new interface for creation of behavioral models. However, intelligence capture cannot create all types of behavioral models, as we clarify shortly.

Conflict elimination is important because a demonstrator is unlikely to perform the same task twice in exactly the same manner. Moreover, conflicts may be impossible to avoid due to varying delays in response time. These conflicts, if left unresolved, are likely to result in either unintended behavior or temporal aliasing (dithering) in the animation. Our conflict elimination technique safely removes high frequencies in the state-action pairs, helping make μ a smooth and representative mapping.

To further help avoid temporal aliasing, we have found it useful to apply the learned behavioral model at the most temporally coarse-grain level possible. This naturally spreads out the action selection process, making visible dithering between actions less likely. The risk of temporal aliasing can be further reduced by constructing μ through SVM, because it performs powerful generalization. Nevertheless, regardless of these precautions, there is no formal guarantee that our technique will never suffer from temporal aliasing.

As has been shown empirically, intelligence capture is simple to implement, fast, and reusable. Its use is also natural and straightforward, and there is empirical evidence that

it is a viable tool for computer animators. We have found that integrating our intelligence capture technique (implemented as a C++ class) into existing animation software usually requires very little effort. Behavioral models can then be constructed by an animator in a matter of minutes and tested within the same framework. Also, note that our approach has a nearly fixed online execution time (unlike some other techniques for explicit behavioral models), and can be computed in a matter of microseconds, which is a useful feature for interactive computer animation.

However, intelligence capture does have some issues. First, the use of the k-nearest neighbor algorithm can require storing many examples, and thus the memory footprint is not negligible. Another issue is that there exists a "communication bottleneck" between the animator and the computer during training in intelligence capture; thus the current state must be presented concisely to the demonstrator and **a** must be of reasonable dimensionality. Next, note that an animator's choice of actions may not make the character traverse into all regions of the state space, leaving gaps in μ. This problem can be automatically solved by, during training, periodically forcing the character into these unvisited regions of the state space. Another issue is that, even with conflict elimination, significant delay in the animator's reactions can still lead to an inaccurate behavioral model; e.g., observing an inaccurate state-action pair because the current state does not actually correlate with the delayed action. This problem can be reduced through conscious effort by the animator, and by keeping Δt as large as possible. Another important issue is *perceptual aliasing*: the possibility that two dissimilar states may appear identical due to our compact state representation. This can result in a character making mistakes. However, perceptual aliasing can be minimized through effective design of the compact state space. The final issue with intelligence capture is that its scalability is limited due to the curse of dimensionality. The state space must be kept reasonably low-dimensional, or attempts to learn μ will fail.

There are certain types of behavior, involving a large amount of state information (such as chess), that could never be constructed through machine learning in the manner we propose. Also, intelligence capture can only learn deterministic policies. However, we (and researchers in agents/robotics) have empirically shown that some classes of behavior popular in behavioral animation can be learned. Our own case studies cover a wide range

of temporal decision-making granularities, with rigid and articulated characters, involving navigation and collision avoidance behavior. There are many other interesting case studies of programming agents by demonstration, such as [van Lent and Laird 2001] which examines programming-by-demonstration for aircraft piloting, etc.

Some behavioral models can operate best as part of a complete synthetic brain architecture [Isla et al. 2001], where the brain chooses goals, performs sensing, maintains the character's internal state, etc. This may be especially useful for intelligence capture, because the decision making learned by intelligence capture is shallow: simply reactions to perceived states.

An important topic we have not yet discussed is context-sensitive decision making. Most behavioral models are non-context sensitive, and it has been shown in the AI literature that the majority of intelligent tasks can be achieved through non-context sensitive logic. However, we have found that context can be very interesting for portraying emotion. For example, a human often behaves very differently whether she is happy, afraid, angry, etc. This same variety can be achieved easily in intelligence capture by constructing a set of behavioral models, each representing a different emotion. Then, at any given time, the behavioral model to use is selected based on the character's current emotional state. This is also valuable for achieving non-deterministic animation, as policies (being functions) are deterministic.

Finally, modularity in a behavioral model is well known to help produce better results for autonomous characters. Intelligence capture can be used in a modular fashion by capturing separate policies for unrelated portions of a complete behavioral model. This can help limit the number of inputs (dimensionality of **s**) for each policy, which can significantly reduce the number of state-action examples required and simplify the machine learning process. Also, independent models can be learned for distinct tasks and/or goals that a virtual character may need to perform.

An unanswered question is whether another behavior representation (rather than a policy) would be more effective for behavioral animation programming-by-demonstration. The two approaches championed in the field of robotics are learning policies [Kasper et al.

2001] (like our technique), and learning task structure or pre-/post-conditions for behavioral sub-modules [van Lent and Laird 2001; Nicolescu 2003]. We have opted for learning policies because it allows for maximum programming-by-demonstration versus explicit programming. An interesting area for future work is to examine the benefits to behavioral animation in learning pre-/post-conditions, and/or perhaps develop a novel representation.

Chapter 7

Data-Driven Programming and Behavior for Autonomous Virtual Characters

Submitted to IEEE Transactions on Visualization and Computer Graphics, April 2005.

Abstract: We present a novel technique for behavioral animation through data-driven behavior synthesis. This technique has two key features: it provides realistic and natural character behavior, and has a programming-by-demonstration interface. Thus we can quickly create compelling and realistic autonomous virtual characters that exhibit stylized behavior. First, the human user demonstrates behavior for the character by controlling it (e.g., with a joystick) during an interactive session. Each demonstration is recorded as a sequence of discrete actions. Later, when the character behaves autonomously, it performs data-driven behavior synthesis by concatenating segments of action sequences. The choice of action sequence segments is guided by simulations that predict fitness. We empirically show that our technique is robust, computationally feasible, general, and produces effective character behavior. Also, the interface is intuitive enough that realistic virtual character behavior can be effectively created by non-technical users.

Keywords: behavioral animation, programming by demonstration, machine learning, autonomous agents, AI-based animation.

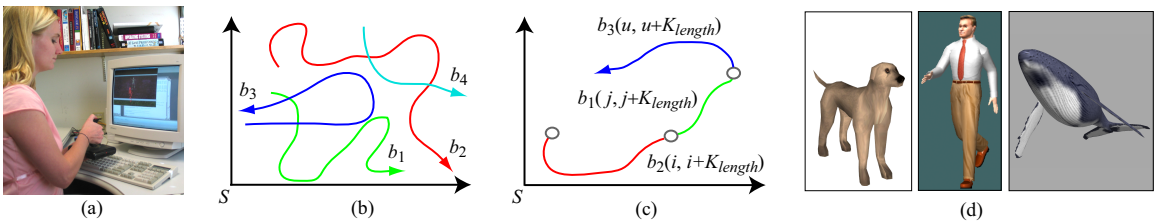

Figure 7.1: *Overview of our approach*. (a) A human interactively demonstrates the target behavior for an autonomous virtual character. (b) These demonstrations are recorded as a library of action sequences, or *behavior trajectories*. (c) At run-time, segments of the behavior trajectories are combined into novel behaviors for the character. Example characters are shown in (d).

7.1 Introduction

The use of autonomous virtual characters is becoming increasingly pervasive. This is because, in complex or interactive virtual environments, the explicit behavior of virtual characters may be too difficult or impossible to define *a priori*. Common applications of these self-animating autonomous characters include training simulators, film, computer games, virtual tutors, etc.

Despite the success of autonomous characters in certain types of virtual environments, some important arguments have been brought against current techniques (see [Funge et al. 1999; Isla and Blumberg 2002]), such as:

1. Behavioral models can often be very difficult and time-consuming to design and program.

2. Character decision making is entirely synthetic, thereby limiting its realism.

In this paper we present a novel technique for behavioral modeling. Our technique addresses the two problems listed above by a combination of programming-by-demonstration and data-driven behavior synthesis. In short, demonstrated human behavior is recorded in an unmodified form, and then is segmented and combined in novel sequences that are performed by the character. Thus the virtual character can engage in innovative behaviors but is constrained to sub-sequences of real human behavior.

Our technique operates in two modes: *training* and *autonomous behavior*. In the training mode, the human user instructs the character by controlling it (e.g., with a joystick) during an interactive session where the character and its virtual environment are visualized in real-time for the demonstrator. Each demonstrated behavior is stored as a *behavior trajectory*: an ordered sequence of actions. After being instructed, a character has a library of one or more behavior trajectories.

Later, in the autonomous behavior mode, the character innovates behavior to perform through data-driven synthesis. This is done by concatenating segments of behavior trajectories. The choice of behavior trajectory segments is guided by simulations that predict the utility of performing each given segment. These simulations are accurate and simple to perform because we leverage the existing virtual environment. To keep the number of simulations tractable, the behavior trajectory segments are clustered so that they can be hierarchically searched.

Our technique is useful because a character can engage in a nearly endless variety of behaviors but is constrained to perform action sub-sequences that have been demonstrated by a human. Thus the character's behavior appears natural and can be easily programmed through demonstration by a non-technical user. Because synthesis is based on simulations, it is unlikely that the character's behavior will suffer from critical generalization errors. We verify our technique with empirical findings and a video, demonstrating that it is robust, tractable, general, and easy to use.

We begin by surveying related work. We then present our technique in detail. Afterwards, we present experimental results from applying the technique to a number of test beds. We then discuss the technique and examine its strengths/weaknesses, usefulness, and our empirical findings.

7.2 Background and Related Work

Our technique was inspired by work in data-driven motion synthesis, e.g., [Arikan et al. 2003; Gleicher et al. 2003; Safonova et al. 2004], etc. In some of these methods, motion capture data is segmented and combined in novel sequences to create new motions (assuming motion is deterministic). These "cut-and-paste" techniques have proven extremely useful for creating motions that are highly realistic, because they leverage actual human motion. This led us to conjecture that data-driven synthesis could be performed at a conceptually higher level: decision making. Unfortunately, simple cut-and-paste of decision-making sequences is not plausible in a complex environment due to non-deterministic state transitions and errors from generalizing action sequences over the state space. To overcome these problems we use simulations to guide the character's choices in data-driven synthesis.

Another source of inspiration for our technique comes from findings in neuroscience and ethology, such as the notion that some animals use a small set of fundamental behaviors to create complex behaviors [Bizzi et al. 1995]. This is a biological parallel of the concept of behavior-based robotics [Arkin 1998]. Our technique can be classified as a behavior-based method. We also have validation for our approach from the perspective that the character *imitates* demonstrated behavior (with some innovation). Many cognitive scientists postulate that imitation is one of the primary learning methods of humans [Byrne and Russon 1998].

A number of noteworthy architectures for behavioral animation of autonomous characters have been proposed. These methods for independent characters include [Funge et al. 1999; Badler et al. 1999; Blumberg et al. 2002; Egges et al. 2004; Dinerstein and Egbert 2005]. Methods for crowds and flocks include [Reynolds 1987; Musse and Thalmann 2001; Metoyer and Hodgins 2003; Anderson et al. 2003; Sung et al. 2004]. In these techniques, characters behave autonomously by making decisions through a *behavioral model*: an executable model defining a character's thought process[1]. While these techniques have

[1] A behavioral model can either make decisions in a *reactive* or *cognitive* (i.e., deliberative) manner [Funge et al. 1999]. For simplicity, and without loss of generality, we simply refer to all these varieties as behavioral models.

produced impressive results, behavioral model programming has remained a difficult, technical endeavor. Moreover, it has proven difficult to create characters that exhibit natural or stylized decision making.

One approach taken to simplifying the programming of behavioral models (problem #1 in Section 7.1) is the creation of special-purpose languages. Some examples, which are specially designed for virtual agents, include [Funge et al. 1999; Devillers et al. 2002]. Special-purpose languages have enjoyed some success, but are often limited to certain types of characters and/or cannot be employed by non-programmers. Another approach has been offline learning, e.g., [Dinerstein et al. 2004b; Dinerstein and Egbert 2004]. In these techniques, the virtual agent is given a fitness function and then automatically constructs a behavioral model. However, because the user's control over the learned model is merely a fitness function, it is difficult to achieve stylized or specific behavior.

To provide a more intuitive agent programming interface, many robotics researchers have turned to *programming by demonstration* [Mataric 2000]. Two main approaches have been examined: learning policies [Kasper et al. 2001; Dinerstein et al. 2004a], and learning task structure or sub-behavior pre/post conditions [van Lent and Laird 2001; Nicolescu 2003]. While these techniques are interesting and powerful, they do not solve problem #2 because they construct models from the demonstrated behavior — these models produce entirely synthetic decision making. For example, the policy-construction techniques blend the demonstrated behavior into a deterministic $state \rightarrow action$ mapping. Thus features of human behavior such as non-determinism, non-Markovian action selection, etc, are lost. Also, these techniques require a large amount of explicit programming, lack scalability, and can only produce shallow decision making. The approach we take in this paper is unique and powerful with respect to the programming-by-demonstration literature.

Little research has been performed on designing virtual agents to make decisions in a human-like manner (problem #2 in Section 7.1). Indeed, most behavioral animation and agent research has focused on simply achieving effective decision making versus stylized decision making. However, for a virtual character to appear convincing, it must behave as its real-world equivalent would behave. In the case of a synthetic human, the behavior should be stylized to the point where it matches the unique personality and conduct of the

real-world human the character is meant to represent. One attempt to solve problem #2 was the use of personality parameters in MASSIVE for crowd animations in the Lord of the Rings films [Duncan 2002]. However, while these parameters allow each character to be unique, it is not entirely clear how to create specific personalities, and the characters' decision making is entirely synthetic.

Our technique is also related to previous work in pre-computation and approximation of expensive state transition and control functions, e.g., [Grzeszczuk et al. 1998; Reissell and Pai 2001; James and Fatahalian 2003]. Most of these methods interpolate discrete samples through some form of machine learning, but this can result in generalization errors. Alternatively, data-driven approaches such as tabulation can be taken. For example, [James and Fatahalian 2003] approximates physical systems using n distinct state space paths. Because the state transitions are deterministic, and the current system state can be forced into a known state, these paths are sufficient to represent n discrete system activities. However, because our problem domain of interest is complex virtual environments with one or more agents (human and synthetic), the environment's state transitions will be non-deterministic and the current state cannot be forcibly changed. Moreover, in a complex environment, it is unlikely that a demonstrator can provide enough information for the character to perform all necessary behaviors without some form of synthesis. Thus we perform data-driven behavior synthesis using simulations to ensure that we achieve sufficient behavior while avoiding generalization errors.

Contribution: In this paper we present a novel character decision making technique that performs data-driven behavior synthesis using data gathered through an intuitive programming-by-demonstration interface. Our technique produces stylized and natural virtual character behavior that can be effectively and quickly demonstrated by non-technical users. We provide empirical evidence that our approach is robust, scales well, and is less computationally expensive than many traditional behavioral animation techniques. Our method is applicable to all kinds of virtual characters whose target behavior can be demonstrated using a broad range of input devices.

With respect to previous work, some of our specific contributions include: (1) our

$\mathbf{a}$	An action. $\mathbf{a} \in A$, $A \subseteq \mathbb{R}^m$.
b	A behavior trajectory (ordered sequence of actions).
B	Set of demonstrated behavior trajectories.
Q	Set of behavior trajectory segments.
f	Fitness function.

Table 7.1: *Summary of notation*.

novel data-driven cut-and-paste approach to decision making; and (2) leveraging robotic programming-by-demonstration concepts for character behavioral modeling.

7.3 Overview and Formulation

Our data-driven technique contains two inter-related methods: programming by demonstration and cut-and-paste behavior synthesis. We begin by providing a general overview and an exposition on the concepts shared between these two methods. Our formal notation is summarized in Table 7.1.

The workflow of our technique is summarized in Figure 7.1. It involves the following stages:

1. *Training:*
    - A programmer integrates the behavior trajectory system into a new, "brainless" virtual character.
    - A human demonstrates behavior trajectories.
    - The demonstrator tests and edits the behavior trajectories.
2. *Autonomous Behavior:*
    - The behavior trajectories are segmented and the segments clustered.
    - The segment selection algorithm chooses the behavior trajectory segment to perform next.
    - The virtual character iteratively performs the actions in the behavior trajectory segments that are selected.

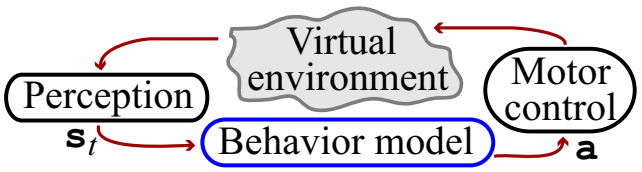

Figure 7.2: *Autonomous virtual character control loop.*

Our technique is designed to provide character decision making. Note that an autonomous virtual character is a synthetic agent and traditionally is composed of several modules as shown in Figure 7.2. We are interested in the behavioral model, where action selection takes place. We assume that motor control, perception, and other agent needs are carried out competently by existing techniques (e.g., [Isla et al. 2001]). For example, motor control for autonomous characters has traditionally been performed through motion capture data (where each action a character may perform maps to an explicit motion).

States, actions, and behavior trajectories: A *state*, $\mathbf{s}$, is a configuration of the virtual environment. An *action*, $\mathbf{a}$, is a primitive activity that a character may perform. The state space is denoted S and the character's action space A. The virtual environment can be formally represented as a non-deterministic relation:

$$\mathbf{s}_{t+1} = E(\mathbf{s}_t, \mathbf{a}_t). \tag{7.1}$$

In other words, at each discrete time step t, the character performs an action $\mathbf{a}_t$ that causes the environment to transition non-deterministically to state $\mathbf{s}_{t+1}$. The environment can include any number of agents: e.g., $E(\mathbf{s}_t, \mathbf{a}_t^1, ..., \mathbf{a}_t^p)$ for p characters. The specific format of $\mathbf{s}$ and $\mathbf{a}$ is not critical, but we use $\mathbf{a} \in \mathbb{R}^m$ so that we can use the Euclidean metric to compare actions. The human's demonstration action space is identical to the synthetic character's action space because the demonstrator controls the character during training.

We record each demonstrated behavior as a *behavior trajectory*, denoted b_i (see Figure 7.3). A behavior trajectory is an ordered sequence of actions that a virtual character may perform:

$$b = \langle \mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_l \rangle. \tag{7.2}$$

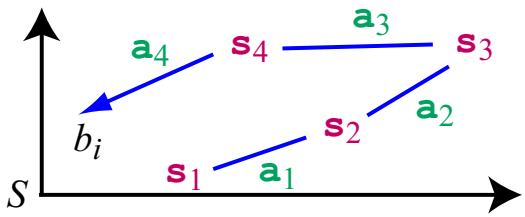

Figure 7.3: *Behavior trajectory*. A demonstrated sequence of actions. The observed state transitions do not generalize easily and are not guaranteed due to non-determinism, so they are ignored.

Data-driven programming and behavior: The demonstrated behavior is observed and recorded according to a fixed time step Δt (e.g., 15 Hz). At each discrete step t, the demonstrator's current action $\mathbf{a}_t$ is appended to the end of the behavior trajectory. Later, during autonomous behavior, the character performs the action sequences using the same Δt.

The set of behavior trajectories demonstrated for the character is denoted $B = \{b_i\}$. Due to practical constraints on the demonstrator, B is likely to be a small subset of all possible behavior trajectories. The character's current location in a behavior trajectory i (i.e., the next action to perform) is denoted $b_i(j)$, where $j \in \mathbb{N}$ and monotonically increases with each action performed.

The choice of how to cut-and-paste behavior trajectory segments to create an effective novel behavior can be framed as an optimization problem. To make this problem tractable, we segment the trajectories into equal-sized segments of length K_{length}. The set of segments is $Q = \{b_i(j, j+K_{length}) \mid \forall b_i \in B,\ 1 \leq j \leq |b_i| - K_{length}\}$. In other words, Q contains every contiguous sub-sequence of length K_{length} of the behavior trajectories in B. We also cluster these segments so that they can be searched hierarchically (see Section 7.5.1).

The behavior trajectory segment selection algorithm greedily determines which segment is to be performed next by the character. Formally:

$$b_i(j, j+K_{length}) = Select(\mathbf{s}_t), \tag{7.3}$$

where the input is the current state of the environment and the output is the behavior trajectory segment for the character to perform next. In alternative notation, $Select : S \rightarrow Q$.

The goal is to select segments such that the character's behavior b^c is optimal with

regards to its total finite-horizon fitness:

$$b^c = \arg \max_{b_i(j,j+K_{length}) \in Q} \left(\sum_{u=1}^{K_{length}} f(\tilde{\mathbf{s}}_u) \right), \tag{7.4}$$

where $b_i(j, j+K_{length})$ is the sequence of actions tested, $\tilde{\mathbf{s}}_1 ... \tilde{\mathbf{s}}_{K_{length}}$ are the states resulting from performing these actions, and $f : S \rightarrow \mathbb{R}$ is the fitness function.

7.4 Training

Under our technique, training must occur before a character can behave autonomously. As described in Section 7.3, the training mode is composed of three stages: (1) integration of the behavior trajectory system, (2) demonstration of behavior trajectories, and (3) testing and editing of the recorded behavior trajectories. The integration stage must be performed by a programmer. The latter two stages can be performed by any user. Proper training results in a stylized, compelling virtual character that chooses actions through our behavior trajectory technique.

7.4.1 Integration

Integration is a necessary precursor to using our behavior trajectory technique. This involves a programmer plugging our technique into a virtual character, thereby providing it with decision-making skills. Conceptually, this is not much different than developing autonomous characters under any other behavioral animation technique. Our technique is designed to be highly reusable and portable so that it can be easily applied to most characters. The programmer need only supply: (1) a fitness function f, and (2) a distance metric for the character's action space A. The fitness function implicitly defines the character's goal and therefore is unique to each given character and goal. We require a distance metric to compare actions so that behavior trajectory segments can be clustered. All other components of our technique are generally applicable and need not be modified. The existing virtual environment is leveraged for simulations when selecting behavior trajectory segments to perform.

The fitness function f is solely responsible for defining the target behavior of the character (see Equation 7.4). It is through f that our technique determines what behavior trajectory segment should be performed next. Thus f guides the character's data-driven behavior synthesis. f must represent for each given state an approximate usefulness of the agent's achieving that state ($\texttt{fitness} = f(\mathbf{s}_t)$). Fortunately, fitness functions are well-established tools in the literature [Russell and Norvig 2003]. *Interpolation of fitness labels* and *potential fields* are particularly interesting methods because a fitness function can be generated semi-automatically.

An appropriate distance metric for the character's action space is necessary for us to be able to cluster behavior trajectory segments. This clustering is useful for allowing us to perform rapid searches, thereby speeding up behavior synthesis. In all of our case studies, we utilize the Euclidean metric to compute action-action comparisons. This is straightforward due to our use of real-vector-valued action spaces (see Section 7.3). Thus:

$$d(\mathbf{a}_i, \mathbf{a}_j) = \|\mathbf{a}_i - \mathbf{a}_j\|_w, \tag{7.5}$$

where $\mathbf{a}_i, \mathbf{a}_j \in \mathbb{R}^m$, and w is an optional weighting matrix. The only requirement we must fulfill is that the action space A be defined such that similar actions are located near each other in A. Of course, an alternative action space formulation and/or distance metric can be used if desired.

7.4.2 Demonstration

Once our behavior trajectory technique has been integrated (as detailed in §7.4.1), the virtual character can be instructed by a human demonstrator. The programming-by-demonstration interface operates as follows (see Figure 7.1 a-b). The character and its world are visualized interactively for the demonstrator. Thus the demonstrator is continuously being updated with the current state of the environment. The human user demonstrates the target behavior for the character by interactively controlling it using one or more input devices. Thus the sequence of actions the demonstrator chose for the character is recorded in order. Each demonstration session is recorded as a separate behavior trajectory b_i, and stored in B. The number of behavior trajectories, $|B|$, is not important — we can

extract the same number of behavior trajectory segments from one long trajectory as several short ones. The only reason for not using a single long behavior trajectory is that the logical sequence of actions will be broken when a session change occurs.

Any human input device is pertinent for use in training virtual characters. We have successfully used joysticks, keyboards, and mice. Other possibilities include motion capture systems, etc. All that is necessary is for the input device to map onto the character's action space.

It is likely intractable (in both storage and the demonstrator's time) to require that the demonstrated behavior trajectories densely sample the entire space of possible behaviors. However, this is not necessary for our technique since behavior trajectory segments are combined to synthesize pertinent behavior.

Our programming-by-demonstration interface is intuitive, and can be used effectively by non-technical users to quickly create compelling character behavior (as we show in the experimental results section). One limitation is that the demonstrator must be able to control the character in real-time. Therefore, the dimensionality of the action space must be reasonably small.

7.4.3 Testing and Editing

Once demonstration of behavior trajectories is complete, the set of trajectories B and the fitness function f can be tested. This is done in two ways: first, by directly playing back the recorded behavior trajectories; secondly, by observing the character as it exercises autonomous behavior in the virtual environment. If it is clear that a portion of a behavior trajectory $b_i(j, j+\zeta)$ represents undesirable behavior, that portion is marked and deleted. The first part of the trajectory ($< j$) is retained, and the remainder of the trajectory ($> j+\zeta$) becomes a new trajectory. If the character's autonomous behavior is incorrect, it is likely that either f is incorrect or that B contains an insufficient amount of data. Corrections in f can be made by the programmer, and any user can add more behavior trajectories.

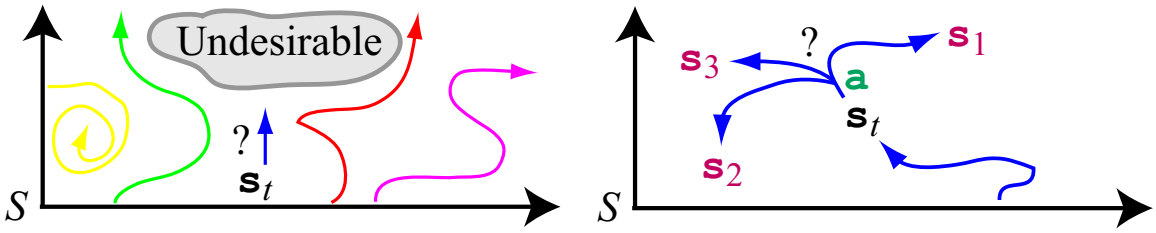

Figure 7.4: *(Left)* Generalization of behavior is not trivial, and can lead to error. *(Right)* The virtual environment is likely to be non-deterministic, especially if it includes multiple characters and/or a human user.

7.5 Autonomous Behavior

Our approach to data-driven autonomous behavior is motivated by several issues, some of which are shown in Figure 7.4. Given the black box origin of the behavior trajectories, the target behavior is generally complex and generalization of the behavior data is problematic. Fundamental complications include insufficient data, complex environments, and non-determinism in the environments and human users/secondary characters. As a result, the character could never reactively use this behavior data without risk of inappropriate behavior. To ensure intelligent and appropriate decision making we have designed our technique to simulate the outcome of its choices.

The virtual character does not merely play back "canned" behaviors. Rather, it performs innovative behaviors, unique to its current situation $\mathbf{s}_t$, which are created by concatenating segments of behavior trajectories. Behavior synthesis can be framed as an optimization problem. This optimization problem is difficult because we wish to achieve real-time performance (for interactive environments). Our behavior trajectory construct can be considered an optimized, stylized search space representation that is defined through programming-by-demonstration.

7.5.1 Behavior Synthesis

Synthesizing behavior by concatenating behavior trajectory segments is an optimization problem. Specifically, it is a Markov Decision Process (MDP). As shown in Equation 7.4, only the future is taken into account in determining which segment is optimal

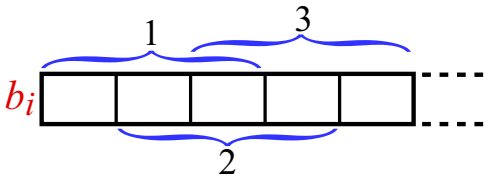

Figure 7.5: *Example of behavior trajectory segmentation*. Three segments, each of length three.

given the current state. Ideally, we should solve this optimization problem through dynamic programming to construct a sequence of behavior trajectory segments that maximizes long-term fitness. However, this is implausible because we perform an internal simulation to measure the fitness of each candidate segment. Thus we have opted for a greedy approach, where only the next behavior trajectory segment is selected. This reduces the problem to O(n), where n is the number of segments to consider.

Even with our greedy approach to optimization, our technique will likely not execute in real-time if there are prohibitively many behavior trajectory segments to test (i.e., $|Q|$ is large). A naïve solution to this challenge is to leverage state information to suggest the region of the state space where each segment will likely be useful, thereby limiting the number of segments to simulate. However, as discussed throughout this paper, the virtual environment is complex and non-deterministic. Thus it would be difficult or improbable to effectively generalize states and thereby gain context. Instead, we hierarchically cluster the behavior trajectory segments. Thus the segments can be searched in O(log n) time. While this does introduce the possibility of finding sub-optimal segments, we show in the empirical results section that our greedy-hierarchical algorithm produces compelling results and is very fast.

Segmentation and Clustering

We segment the trajectories into equal-sized segments of length K_{length} (see Fig. 7.5). Specifically, there is a segment for every point $b_i(j)$ that is followed by at least K_{length} actions in the given trajectory. Thus these segments mostly overlap. The purpose of this is

to make sure that every contiguous sub-sequence of length K_{length} in B is available for the character to perform. All segments are placed in a set denoted Q.

To allow the segments to be searched hierarchically, we create a hierarchical clustering. Specifically, a β-ary tree of clusters is created, where $K_{nclust} = \beta$ is the number of clusters created at each iteration of clustering. The tree is created recursively (as shown in pseudo-code in Figure 7.6). The root of the tree is a single cluster of the entire set of segments (Q). We begin by clustering the root cluster into K_{nclust} sub-clusters. Each segment in Q belongs to one and only one of these sub-clusters. These sub-clusters are the child nodes of the root in the tree. Next, any sub-cluster than contains more than K_{nclust} children is recursively clustered. At each recursive iteration, the new sub-clusters are set as the children of the parent cluster.

Hierarchical clustering is performed only once offline after training is complete. Thereafter, the tree of clusters can be used without change to select segments for the character to perform (as detailed in the next section).

We create clusters using the *k-means clustering algorithm* [Duda et al. 2000]. The mean of a cluster is the average of each action (1 through K_{length}) in the segments in the cluster: $\langle \bar{\mathbf{a}}_1, ..., \bar{\mathbf{a}}_{K_{length}} \rangle$, where bar denotes average. The difference between two segments is computed as:

$$\texttt{difference} = \sum_{i=1}^{K_{length}} \left(\|\mathbf{a}_{1,i} - \mathbf{a}_{2,i}\| \cdot 0.95^{i-1} \right), \tag{7.6}$$

where $\|\cdot\|$ is the Euclidean metric. The term "0.95^{i-1}" discounts each action such that those early on have the greatest weight. The constant 0.95 was set empirically.

Note that k-means is an iterative algorithm that is randomly seeded, and may converge to a local minimum. To ensure that we get a good clustering, we recompute the cluster tree multiple times and keep the tree that performs best in the virtual environment (i.e., has the highest average fitness).

Behavior Trajectory Segment Selection

Our segment selection algorithm operates as follows (pseudo-code is given in Figure 7.7). A beam search of the cluster tree is performed until the first leaf is found. When a node is traversed, it is expanded by testing all of the K_{nclust} clusters (nodes) that descend

```
HCluster(R) {   // R is a node in tree (cluster) to hierarchically cluster
   Create K_nclust clusters of the segments in R
   Set new clusters as children of R in tree
   For every new cluster C_i ...
     if (C_i contains ≥ K_nclust segments)
       Call HCluster(C_i)
}
Set root of tree as Q
HCluster(Q)
```

Figure 7.6: *Pseudo-code of the hierarchical clustering algorithm.*

from it. Specifically, each cluster's *prototype member* is simulated to measure its fitness given the current state $\mathbf{s}_t$. The prototype member of a cluster is the segment that is closest to the cluster mean according to Equation 7.6. The fitness of the segment is computed using Equation 7.4. The cluster whose prototype member has the highest fitness is the winner, and it is the node that is traversed next in the tree. Once a leaf node is reached, all segments in that cluster are simulated. These segments are then ranked by fitness and the character probabilistically selects one to perform, where $p = 1/(2^i)$ for $i = rank$.

The character performs the actions in the selected segment in order, from $\mathbf{a}_1$ to $\mathbf{a}_{K_{length}}$. We periodically (once every K_{verify} steps, where $1 < K_{verify} < K_{length}$) determine whether the currently selected behavior trajectory segment is still valid. This is important because, due to error in the simulations and/or non-determinism, the remainder of a chosen segment may prove detrimental for the character to perform. If the current segment is deemed invalid, our algorithm immediately selects a new segment. Thus the character can respond robustly to surprising events, but because $K_{verify} > 1$ segments are not abandoned prematurely so "thrashing" is avoided.

Two distinct factors assure that the demonstrator's style of behavior will be faithfully reproduced by the character. First, only actions demonstrated by the human trainer exist in B and Q, and therefore the character is limited to the subset of the action space demonstrated. Second, the character must perform contiguous sequences of actions demonstrated by the trainer. The larger the values of K_{length} and K_{verify}, the more constrained the character will be to the trainer's behavior.

```
Select(s_t, N) {   // s_t and N are the current state and node in tree.
    best = NULL
    For each child node C_i of N ...
      Simulate the prototype member of C_i   // Prototype is member closest to mean.
      if (C_i prototype more fit than best)   // Fitness measured by Eq. 7.4.
        best = C_i
    if (best is a leaf node)
      Simulate all segments in best
      Rank segments by fitness and probabilistically select one
      Return selected segment
    else return Select(s_t, best)
}
```

```
Character {
    For every discrete time step t ...
      if (a new segment needs to be selected)
        Perceive current state s_t
        Segment to perform = Select(s_t, root)
      Perform next action in current behavior trajectory segment
}
```

Figure 7.7: *Pseudo-code of the behavior trajectory segment selection algorithm and the character control loop that uses it.*

In our implementation, transitions between segments are performed by concatenating action sequences. In other words, no blending of actions is performed. Because our technique can form many different behaviors from the set of segments Q, and selection of segments is probabilistic, our technique produces rich and varied non-deterministic character behavior (as shown in Section 7.7 and the video available from `http://rivit.cs.byu.edu/a3dg/publications.php`).

7.5.2 Running Simulations

As discussed previously, our technique uses simulations to select segments (see Fig. 7.8). This is important because, as shown in Figure 7.4, generalization of behavior data is not trivial. Simulation allows us to choose a segment with high confidence in the expected outcome. We leverage the existing virtual environment to run internal simulations. These simulations are transparent to the user/observer. Thus, the environment is not visualized graphically, and the actual state of the environment $\mathbf{s}_t$ does not change.

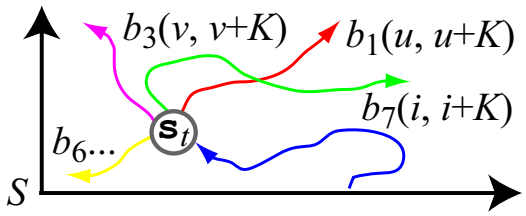

Figure 7.8: The fitness of a candidate segment is computed through simulation, starting at the current state. No consideration is given for following segments.

We run a single simulation for each segment to test. Although averaging the fitness of multiple simulations may reduce variance due to non-determinism in the environment, this has not proven necessary in practice.

Any secondary characters in the environment may be included in the simulations to determine their affect on the fitness of each candidate segment. To do this, we simply have these characters make decisions based on the simulated state. To achieve faster results, we can have them compute fast heuristic decisions or simply assume their actions. It can also be useful to only consider those secondary characters that will most likely have a profound influence on the fitness of a given segment. For a human user that is interacting with the character (e.g., through an avatar), her actions can be predicted through one of a number of existing agent modeling techniques, e.g., [Laird 2001; Dinerstein and Egbert 2005].

7.5.3 Parameters

We utilize three parameters in our technique: K_{length}, K_{verify}, and K_{nclust}. Each parameter is set to a positive integer value. So far, we have only discussed these parameters in tandem with the presentation of our technique. We now define these parameters:

- K_{length} : *Length of all behavior trajectory segments*. The longer the segments, the more the character is constrained to the demonstrator's behavior, and vice versa. The default setting is $K_{length} = (1.5 \text{ seconds})/\Delta t$, where Δt is the fixed time step between the character's actions.
- K_{verify} : *Behavior trajectory segment verification rate*. Once every K_{verify} discrete time steps (i.e., actions performed), the remainder of the current segment is simulated

to verify that it is still valid. Specifically, if the average fitness of the remaining actions is poor (e.g., less than zero), then a new segment is immediately selected. Thus unexpected events can be handled robustly. The default is $K_{verify} = K_{length} / 5$.

- K_{nclust} : *Number of clusters*. This parameter defines the number of child clusters created by our hierarchical clustering every time a cluster is divided. Thus this is the branching factor of our cluster tree. This parameter tunes a quality/CPU utilization trade-off. The default setting is $K_{nclust} = 25$.

These are all the parameters of our technique. This number of parameters (three) is quite reasonable for a technique of this complexity. We discovered the given default values through empirical evaluation. Our studies indicate that the character behavior is largely insensitive to the parameters, and the parameters usually do not need to be modified from their default values (see Section 7.7 for details).

K_{length} and K_{verify} have very little effect on the CPU utilization of our technique. This is because we only have to simulate one partial segment to perform verification, and the length of the segments is counter-balanced by how often a new segment must be selected. However, K_{nclust} has a notable effect on CPU utilization because this is the branching factor of our cluster tree. A higher value of K_{nclust} provides more accurate segment selection but at the cost of more CPU cycles.

7.6 Using Our Technique in Practice

Modularity: In our technique, the character's goal is implicitly defined by the fitness function f. Therefore, a change in goal at run-time simply requires switching f. Most virtual characters have a small number of candidate goals — thus this set of goals can be represented by a set of fitness functions $\{f_1, ..., f_e\}$. It may be useful to construct a set of behavior trajectories for each fitness function $\{\langle f_1, B_1\rangle, ..., \langle f_e, B_e\rangle\}$. This is because useful behavior trajectories may be unique for each goal.

Temporal antialiasing: We have found that penalizing temporal aliasing in the segment selection algorithm can be aesthetically useful when actions are of short duration (i.e.,

Δt is small). We do this by computing the Euclidean distance between the last action in the previous segment and the first action in the candidate segment: $d = \|\mathbf{a}_{1,K_{length}} - \mathbf{a}_{2,1}\|$. We then scale the fitness of the candidate segment: $f(b_i(j, j+K_{length}))\ /\ (1+d)$. Thus, segments that match well with the previous segment are favored for selection.

Online learning: In many virtual environments, a human user will interact with the environment and characters living therein. This is the case with many training simulators, computer games, etc. We can non-obtrusively leverage these user interactions to gather additional behavior trajectories online. We do this as follows. First, the virtual character infers the goal of the human user. This is usually trivial in virtual environments, because the environment often constrains or assigns the user's goal (for more discussion on this topic see [Blumberg et al. 2002; Dinerstein and Egbert 2005]). If the inferred goal is a goal that the character may engage in, then the user's current behavior may be of interest. The character observes and records the user's behavior as a behavior trajectory. Finally, if the user achieves her goal, the new behavior trajectory is segmented and added to the cluster tree (either by re-clustering or by adding each segment to the best-fitting leaf node cluster). Otherwise, the trajectory is deemed ineffectual and is discarded.

Online behavior trajectory acquisition is interesting because it allows a character to "steal" the user's best tricks. Also, any important gaps in a character's behavior repertoire may be filled in.

Motion synthesis: Motion synthesis is often performed in an online fashion since the motion needs of a character may not be known *a priori*. Unfortunately, highly flexible cut-and-paste motion synthesis requires a notable amount of CPU time to perform. One well-known technique, [Arikan et al. 2003], can be executed interactively but with high CPU utilization.

An interesting aspect of our technique is that it provides a convenient platform upon which to perform motion synthesis in an *offline* fashion. This follows from the fact that the set of behavior trajectories is known and therefore motion can be synthesized for each of these trajectories offline. In other words, we know *a priori* the sequences of actions

the character may engage in, and thus can pre-compute and store the synthesized motion. Because the number of behavior trajectories ($|B|$) is small, only a manageable amount of storage is required. Motion synthesis between concatenated behavior trajectory segments can then be performed online. Since concatenation only occurs occasionally, this selective online synthesis is far less expensive than performing all synthesis online.

7.7 Experimental Results

We used four test beds in our experiments. We briefly summarize these test beds (see Figure 7.9), followed by a discussion of our findings. Our results are summarized in Tables 7.2–7.5 and Figures 7.9–7.14. See the accompanying video for demonstrations of our technique in practice (available from `http://rivit.cs.byu.edu/a3dg/publications.php`).

7.7.1 Summary of Test Beds

In all of our test beds, there is no explicit communication between the characters. Also, the characters must rely on "visual" perceptions to ascertain the current state of the virtual world $\mathbf{s}_t$. Perception is performed by each character individually, and semi-realistic sensory honesty is enforced (i.e., an character can't see through the back of its head, etc). Human input is provided through a joystick and/or keyboard.

Submarine Pilot: Our first test bed involves a virtual submarine pilot (see Figures 7.9 and 7.11). The pilot's goal is to cross an unknown school of whales as quickly as possible with no collisions. The fitness function is given in pseudo-code below. The maneuverability of the sub is limited and the whales are placed close together, making this a challenging problem. Actions are performed at a rate of $\Delta t = 1/15$ second. The action space $A \subset \mathbb{R}^2$ is composed of a continuous range of changes in the sub's orientation $(\Delta\theta, \Delta\phi)$, where $\theta, \phi \in [-\pi/2, \pi/2]$ and θ is yaw and ϕ is pitch. Straight forward motion (directly crossing the school) corresponds to an orientation of $(\theta = 0, \phi = 0)$. The fitness function uses the following state information: (θ, ϕ) is the sub's current orientation and $(\Delta x, \Delta y, \Delta z)$ is

Figure 7.9: *Summary of our test beds*. From top-left to bottom-right, submarine pilot, predator & prey, crowd behavior, and capture the flag.

the translation-invariant separation between the sub and the closest whale. The default parameters are used.

f_sub_pilot(θ, ϕ, Δx, Δy, Δz) {

const float WHALE_SIZE = 1.0;

float fitness = $2\pi - (|\theta| + |\phi|)$; /* Zero angle is straight forward */

if ($\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2} <$ WHALE_SIZE)

fitness = fitness $-$ 1000.0; /* Submarine has hit a whale. */

return (fitness);

}

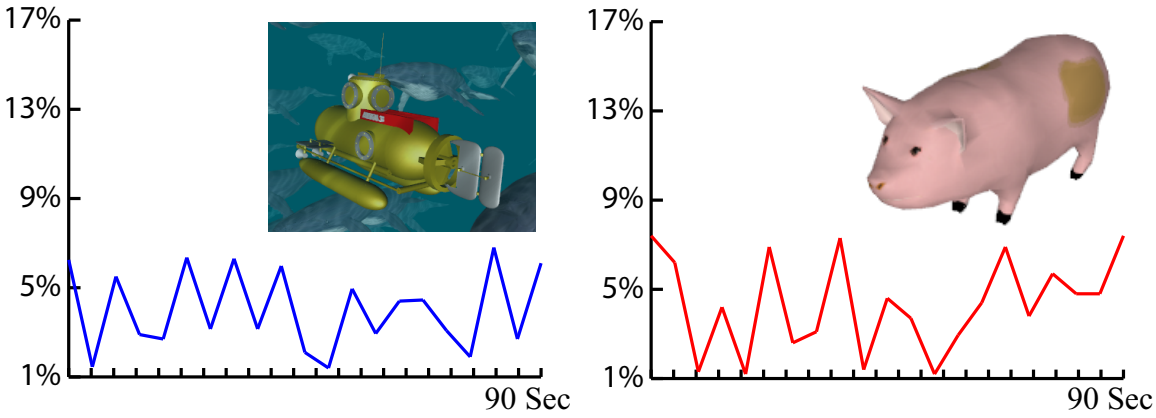

Figure 7.10: Average online CPU usage by our submarine pilot and pig characters on a 3 GHz processor. Mean CPU load is ∼4.2%. In comparison, A*-based deliberative decision making utilized ∼17.9% of the CPU. We achieved similar results with our other characters.

Predator & Prey: Our second test bed depicts a predator-prey scenario, involving a synthetic dog and pig (see Figures 7.9 and 7.12). The dog is controlled by a traditional cognitive model, while the pig is controlled by our technique. The dog's goal is to catch the pig as quickly as possible. The pig's goal is to slip past the dog and hide in the forest located at the top of the environment.

Character motor control is performed through skeletal animation, based on a library of motion capture data. Specifically, there is a motion capture clip associated with each action a character may perform. These clips are blended together when necessary (using quaternion interpolation) to avoid jittering or discontinuities in the motion. A fixed decision-making time step of $\Delta t = 1/15$ second is used (once per animation frame). The action space $A \subset \mathbb{R}^2$ is composed of a continuous range of 2D acceleration vectors, which represent the change in running velocity of a character.

The pig's fitness function is shown in pseudo-code below. The translation-invariant separation between the pig and the closest dog is $(\Delta x, \Delta y)$. The default parameters are used.

$f\_$ pig(Δx, Δy) {

const float DOG_SIZE = 0.4;

float fitness = 0.0;

if ($\sqrt{\Delta x^2 + \Delta y^2} <$ DOG_SIZE)

```
        fitness = fitness − 1000.0; /* Pig can be caught. */
    if (Δy > DOG_SIZE)
        fitness = fitness + 10.0; /* Pig has passed dog, can escape. */
    return (fitness);
}
```

Crowd behavior: This test bed involves crowds of synthetic humans, situated in a virtual museum (see Figures 7.9 and 7.13). Each character's goal is to move through the museum, observing the art without bumping into static obstacles or other characters. Motor control is performed through skeletal animation, based on a library of motion capture data. The characters have deformable skin and clothing. The decision-making time step is $\Delta t = 1/4$ second. The action space is composed of two components: the direction to walk or look, and the specific motion to perform while walking/looking. The default parameter settings are used.

In each experiment there are 5 or more characters. Each character is independent and fully autonomous. To allow for timely training, only one or two sets of behavior trajectories are demonstrated — each character uses one of these sets, sharing it with other characters. The demonstrations are performed in an empty environment, and provide the characters with basic navigation maneuvers and art-gazing behaviors.

Because there are several characters in a confined space, collision avoidance requires that the motion of neighboring characters be considered during simulation. To keep these simulations tractable, each character considers only her two nearest neighbors and assumes that her neighbors will merely continue along their current 2D motion vectors. This simple prediction scheme has proven sufficient in this case study.

Capture the flag (CTF): This case study is based on a well-known research test bed called *Gamebots* [Kaminka et al. 2002]. This test bed modifies the popular computer game *Unreal Tournament 2003*, allowing a programmer to replace the built-in behavioral model. The object of team #1 is to protect the flag, while team #2 seeks to capture it. The players are armed with "tag guns"; once a player is tagged, he is out for a period of time. Each

	Dem. time	Memory	$\|B\|$	$\|Q\|$
Sub Pilot	1 min	24 KB	3	900
Pred. & Prey	2 min	41 KB	8	1700
Crowd	1 min	9 KB	1	300
Capture the Flag	7 min	5 KB	6	80

Table 7.2: *Average programming-by-demonstration statistics for successful character behavior in our case studies.* $|B|$ is the number of behavior trajectories and $|Q|$ is the number of trajectory segments.

	Human	A*	$\|Q\| = 1000$	$\|Q\| = 3000$
Sub time (min)	.58	.53	.62	.57
Sub close calls	5	0	2	1
Pig escapes	80%	81%	79%	81%

Table 7.3: *Average effectiveness of our data-driven technique compared to the demonstrator and a traditional decision making technique (A*).* A* may produce slightly more effective behavior than our technique because it is not constrained to natural behavior — but it is less realistic, non-stylized, and requires more CPU time. *Sub time* is the time taken to cross the school of whales. *Sub close calls* is the number of times the submarine nearly crashed. *Pig escapes* is the percentages of situations where the pig escaped the dog.

team has 5 players, each of which is either an autonomous character or user avatar. A slide show of this case study is given in Figure 7.14.

We have modified the Gamebots test bed so that, rather than overriding the characters' standard behavioral model, we simply select the category of behavior that the character will engage in: e.g., guard the flag, approach other team's flag, hold position, etc. We used a decision-making time step of $\Delta t = 5$ seconds. This high-level approach to control is interesting, as notable success has been achieved in this genre of computer game by selecting from a small set of specific behaviors [Laird 2001]. In a sense, our technique is assigning tasks or behaviors rather than specific actions to the character. Because of targeting such high-level decision making, we set the parameter $K_{length} = 30$ seconds. The other two parameters are not modified.

	Instruction time	*Demonstration time*
Author	Not applicable	∼2 Min.
Artist	∼4 Min.	∼3.4 Min.
Game Player	<1 Min.	∼1.8 Min.

Table 7.4: *Results of an informal user case study, involving one of the authors, an artist, and a computer game player*. Results are time to instruct the user, followed by time for user to demonstrate behavior (pred-prey test bed). Our programming-by-demonstration interface has proven intuitive and applicable for non-technical users.

	default	$K_{nclust} \ / \ 2$	$K_{nclust} \times 2$	$K_{nclust} \times 4$
default	34.5 s	36.5 sec	34.1 sec	33.9 sec
$K_{length} \ / \ 2$	32.2 s	33.0 sec	31.9 sec	31.8 sec
$K_{length} \times 2$	38.5 s	38.9 sec	37.3 sec	36.5 sec
$K_{verify} \times 2$	34.6 s	36.5 sec	34.2 sec	34.2 sec

Table 7.5: *Average time to cross the school of whales using different parameters*. All parameter values are default, some with modification. We achieved similar stable results with our other test beds. As can be seen, our technique is largely insensitive to parameter changes. Most variation is caused by modifying K_{length}, which controls the degree of behavior synthesis allowed.

7.7.2 Findings

Programming by demonstration in our technique has empirically proven to take very little time and storage (see Table 7.2). In fact, a small amount of behavior trajectory data is sufficient to achieve interesting character behavior (see Table 7.3). To provide evidence that our programming-by-demonstration interface is intuitive, we have performed an informal user case study (see Table 7.4).

As detailed in Subsection 7.7.1, we usually use the default parameter settings in our test beds. Our technique has empirically proven largely insensitive to parameter change (see Table 7.5), though tuning the parameters may provide some improvement.

We have found that exploiting symmetry can reduce the number of demonstrations that must be performed. In the submarine test bed, behavior trajectories can be mirrored by inverting all the $\Delta\theta$ and/or $\Delta\phi$ actions. In our experiments, symmetry can reduce the required number of behavior trajectories by up to an order of magnitude.

Of course, our technique does not work well for all types of character decision making. Behavior that must be extremely specific may not be plausible given our use of contiguous segments of demonstrated behavior. For example, the game of chess is not a good fit, because a single sub-optimal action can have disastrous results. Nevertheless, our technique has been shown empirically to work well for a broad range of virtual character behaviors.

As detailed in Table 7.3 and Figure 7.10, our technique quickly produces empirically effective character behavior. Our technique also produces natural, realistic, and aesthetically pleasing behavior as is demonstrated in the accompanying video. This is in contrast to traditional behavioral models, which are time-consuming to program and produce entirely synthetic decision making.

Our technique is faster than some traditional decision-making techniques, such as deliberation through A* (see Figure 7.10). This is partly because, as discussed in Section 7.5.1, our technique is O(log n) where $n = |Q|$. Thus our technique is quite scalable. In comparison, most other deliberative techniques are O($|A|^m$) where m is the number of steps to plan. The use of behavior trajectories provides us with a compact, effective search space representation. The demonstrator provides the character with a small but effective subset of all possible action sequences.

We have found that developing an autonomous virtual character with our technique is significantly faster than the traditional approach where the behavioral model is implemented by a programmer. This is because explicitly designing character AI is challenging. In contrast, our technique automatically creates a behavioral model from brief demonstrations.

7.8 Summary

We have presented a novel technique for virtual character decision making that performs data-driven behavior synthesis using data gathered through an intuitive programming-by-demonstration interface. Our technique produces stylized and natural virtual character behavior that can be effectively and quickly demonstrated by non-technical users.

Figure 7.11: *Animation of the Submarine Pilot test bed.*

A weakness of our approach is that it is more computationally expensive than most reactive approaches to decision making. However, it can be argued that deliberative techniques (like ours) can produce superior behavior [Funge et al. 1999]. Another weakness is that our technique does not scale indefinitely (because it is O(log n)). Nevertheless, we have shown that it is quite scalable. Finally, our technique is heuristic because we take a greedy-hierarchical approach to selecting behavior trajectory segments. Thus optimal decision making is not guaranteed.

We provide empirical evidence establishing the notable features of our approach, including that it is robust, is effective in complex virtual environments, and is less computationally expensive than many traditional deliberative techniques. Our technique is applicable to virtual characters whose target behavior: (1) can be demonstrated using an input device; and (2) can be effectively created from segments of demonstrated behavior. While our technique is not plausible for every virtual character, we have empirically shown that it works well for a broad range of applications.

Figure 7.12: *Animation of the Predator & Prey test bed.* The pig slips past the dog and hides in the forest.

Figure 7.13: *Slide show of the Crowd Behavior test bed.* The synthetic humans inhabit a virtual museum. They move about, examining the diverse pieces of art.

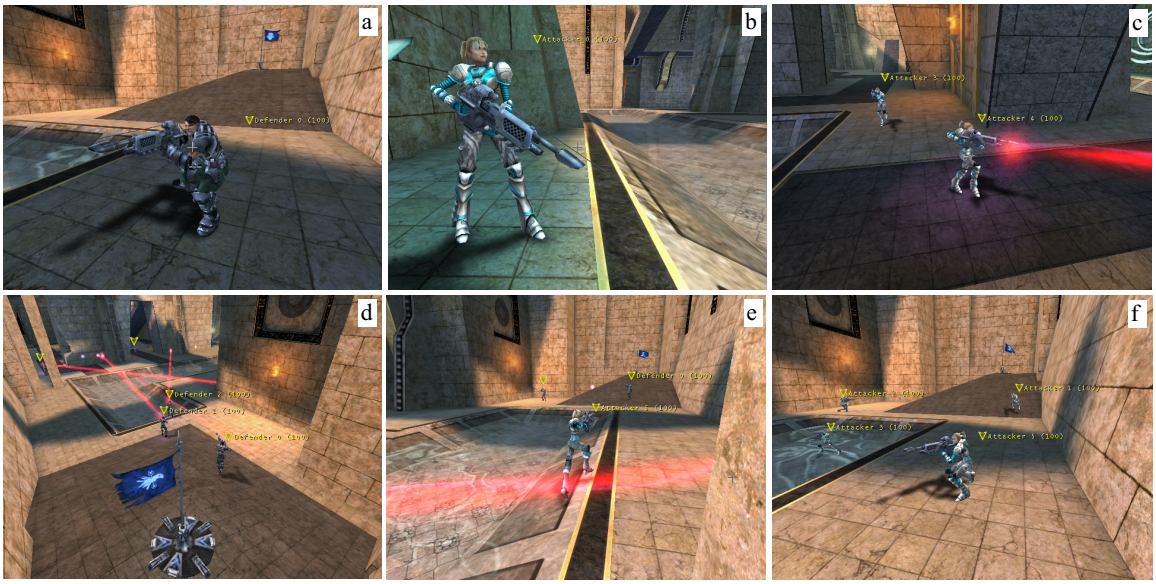

Figure 7.14: *Animation of the Capture the Flag test bed.* (a) Characters on team #1 are represented with this model. Their flag is shown in the background in blue and white. (b) Characters on the opposing team (#2) are represented with this model. (c–f) Team #2 attacks, tagging all defending characters on team #1 and capturing the flag.

Part V

Conclusion

In this part we conclude this dissertation. We also suggest several possible directions for future work.

Chapter 8

Conclusion

Behavioral animation is an important topic that overlaps with diverse fields such as computer graphics, artificial intelligence, multi-agent systems, and machine learning. Current techniques suffer from three notable limitations, as discussed in the introduction of this dissertation:

1. Cognitive models are very slow to execute, distinctly limiting their usefulness.
2. Interactive virtual characters cannot adapt on-line due to interaction with a human user — their behavior is static.
3. There are no simplified techniques for creating behavioral models. Traditionally, a model must be explicitly designed and programmed by a skilled developer.

We have presented several novel methods, rooted in machine learning, that overcome these problems.

We now enumerate the key contributions made in this dissertation and discuss possible directions for future work.

8.1 Contributions

- *A formalization of the needs and requirements of machine learning in behavioral animation.* We have formalized several needs for machine learning in behavioral animation and the requirements for a candidate solution to fulfill these needs. These

needs are listed in Chapter 1 and are analyzed in Chapters 2 through 7. Requirements of some of these needs include rapid learning and execution, robustness when learning from or about human behavior, etc. This formalization is important not only for our work in this dissertation but also for future work as well.

- *Empirical analysis of the advantages gained by applying machine learning in behavioral animation*. We empirically show that machine learning can provide significant benefit to behavioral animation. This should help promote future work in this area.

- *Rapid approximation of cognitive models*. We have presented an original technique for rapidly approximating cognitive models through regression. The function approximation can be performed with one of many machine learning techniques, though we have suggested either continuous k-nearest neighbor or the artificial neural network. As we empirically show, our approximation scheme is effective and very fast. Thus a cognitive model can be practical for use in an interactive setting.

- *Simplified construction of behavioral/cognitive models through planning and approximation*. This method leverages our cognitive model approximation technique but provides it with state-action pairs that are generated through planning. The planning is guided by a user-supplied fitness function. Thus an unknown behavioral/cognitive model can be learned in a matter of hours with limited user intervention—significantly easier than programming an explicit behavioral/cognitive model. However, while this technique is effective, it is difficult to achieve specific or stylized behavior. Thus we have presented our techniques based on programming-by-demonstration for simplified construction of behavioral/cognitive models.

- *Incremental action prediction*. It is very difficult for a synthetic agent to adapt to the behavior of a human because human behavior is non-deterministic, non-stationary, etc. We present a novel technique that constructs a *state* $\rightarrow$ *action* model of the observed behavior of a human user (or a synthetic agent). Through this model an autonomous character can predict the future behavior of the user and thereby act

more intelligently than it could without any knowledge of the future. This technique is compelling because it is very fast.

- *Multi-level learning for autonomous characters*. This technique leverages our action prediction method and other learning schemes to allow a character to adapt in a multi-level fashion. Specifically, the character's action selection, task selection, and goal selection are all able to adapt such that the character can better interact with a unique human user.

- *Behavioral model creation through programming-by-demonstration*. This technique constructs a policy through generation of observed state-action pairs. Similar techniques exist in the robotics/agents literature, but our method is unique in that it is applied specifically to behavioral animation. Also, our method contains a novel conflict elimination algorithm that reduces undesirable behavior due to incorrect generalization.

- *Data-driven specification and synthesis of character behavior*. Learning policies is simple and fast, but policies are limited to shallow decision making. Moreover, reactive generalization can result in incorrect behavior. We have presented a novel technique that captures action sequences from observation of human demonstration. Afterwards, disjoint segments of the action sequences are combined to form novel behavior. This "cut and paste" approach is similar to current trends in motion synthesis. Because we synthesize using real human behavior, the synthesized behavior is likely to appear highly realistic.

8.2 Future Work

In the future, we would like to extend and compliment the research in this dissertation. Below we list some notable and general directions we would like to pursue in the future. For specific ideas of approaches to this future work, see the end of each chapter.

- *Character adaptation that utilizes little or simple domain knowledge*. Our online character adaptation techniques (Chapters 4 and 5) are interesting and powerful but

require significant domain knowledge. For example, action prediction requires that a compact and effective state space be defined. This is not always plausible and limits the scalability of our adaptation technique. An ideal solution to the adaptation problem would require less domain knowledge or domain knowledge that is easier to acquire.

- *More scalable model approximation*. Our model approximation techniques are distinctly limited with respect to scalability due to the need for compact state space formulations. For example, some complex virtual environments may not be accurately representable by a small set of features. It may be possible to partially alleviate this limitation through feature discovery and selection but this remains a difficult research problem. Alternatively, better scalability might be achieved through a different approach to model approximation.

- *Alternative uses for machine learning in behavioral animation*. In this dissertation, we have identified and addressed three limitations of behavioral animation that can be solved through machine learning. There may exist other interesting avenues for applying machine learning in this field.

Bibliography

C Aggarwal, A Hinneburg, and D Keim. On the surprising behavior of distance metrics in high dimensional space. *Lecture Notes in Computer Science*, 1973:420–435, 2001.

T Alexander. Gocap: Game observation capture. In E. Rabin, editor, *AI Game Programming Wisdom*, pages 579–585. Charles River Media, Hingham MA, 2002.

M Anderson, E McDaniel, and S Chenney. Constrained animation of flocks. In *Proceedings of Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 286–297, 2003.

O Arikan, D Forsyth, and J O'Brien. Motion synthesis from annotations. *ACM Trans. on Graphics*, 22(3):402–408, 2003.

R C Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, Mass., 1998.

N Badler, M Palmer, and R Bindiganavale. Animation control for real-time virtual humans. *Communications of the ACM*, 42(8):65–73, 1999.

E Bizzi, S F Giszter, E Loeb, F A Mussa-Ivaldi, and P Saltie. Modular organization of motor behavior in the frog's spinal cord. *Trends in Neuroscience*, 18:442–446, 1995.

B Blumberg, M Downie, Y Ivanov, M Berlin, M P Johnson, and B Tomlinson. Integrated learning for interactive synthetic characters. In *Proceedings of SIGGRAPH 2002*, pages 417–426. ACM Press / ACM SIGGRAPH, 2002.

B Blumberg and T Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of SIGGRAPH 1995*, pages 47–54. ACM Press / ACM SIGGRAPH, 1995.

R Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

H Bui, D Kieronska, and S Venkatesh. Learning other agents' preferences in multiagent negotiation. In *Proceedings of Thirteenth National Conference on Artificial Intelligence*, pages 114–119, 1996.

R Burke, D Isla, M Downie, Y Ivanov, and B Blumberg. Creature smarts: The art and architecture of a virtual brain. In *Proceedings of the Computer Game Developers Conference*, 2001.

R Byrne and A Russon. Learning by imitation: A hierarchical approach. *Journal of Behavioral and Brain Sciences*, 6(3), 1998.

S Carberry. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11:31–48, 2001.

D Carmel and S Markovitch. Incorporating opponent models into adversary search. In *Proceedings of Thirteenth National Conference on Artificial Intelligence*, pages 120–125, 1996a.

D Carmel and S Markovitch. Learning models of intelligent agents. In *Proceedings of Thirteenth National Conference on Artificial Intelligence*, pages 62–67, 1996b.

F Devillers, S Donikian, F Lamarche, and J-F Taille. A programming environment for behavioural animation. *Journal of Visualization and Computer Animation*, 13:263–274, 2002.

J Dinerstein, T Crow, and P K Egbert. Intelligence capture. Technical report, Brigham Young University, 2004a. http://rivit.cs.byu.edu/a3dg/publications/intelcap.pdf; accessed January 17, 2005.

J Dinerstein and P Egbert. Improved behavioral animation through regression. In *Proceedings of Computer Animation and Social Agents (CASA '04)*, pages 231–238. Computer Graphics Society, 2004.

J Dinerstein, P Egbert, H de Garis, and N Dinerstein. Fast and learnable behavioral and cognitive modeling for virtual character animation. *Journal of Computer Animation and Virtual Worlds*, 15(2):95–108, 2004b.

J Dinerstein and P K Egbert. Fast multi-level adaptation for interactive autonomous characters. *ACM Trans. on Graphics*, 24(2), 2005.

R Duda, P Hart, and D Stork. *Pattern Classification, 2nd ed.* Wiley Interscience, 2000.

J Duncan. Ring masters. *Cinefex*, 89:64–131, 2002.

A Egges, S Kshirsagar, and N Magnenat-Thalmann. Generic personality and emotion simulation for conversational agents. *Journal of Computer Animation and Virtual Worlds*, 15:1–13, 2004.

R Evans. Varieties in learning. In E. Rabin, editor, *AI Game Programming Wisdom*, pages 567–578. Charles River Media, Hingham MA, 2002.

P Faloutsos, M van de Panne, and D Terzopoulos. Composible controllers for physics-based character animation. In *Proceedings of SIGGRAPH 2001*, pages 39–48. ACM Press / ACM SIGGRAPH, 2001.

J Funge. *AI for Games and Animation: A Cognitive Modeling Approach*. A.K. Peters: Natick, MA., 1999.

J Funge. Cognitive modeling for games and animation. *Communications of the ACM*, 43 (7):49–58, 2000.

J Funge, X Tu, and D Terzopoulos. Cognitive modeling: Knowledge, reasoning, and planning for intelligent characters. In *Proceedings of SIGGRAPH 1999*, pages 29–38. ACM Press / ACM SIGGRAPH, 1999.

S Gadanho. Learning behavior-selection by emotions and cognition in a multi-goal robot task. *Journal of Machine Learning Research*, 4:385–412, 2003.

M Gillies and N Dodgson. Eye movements and attention for behavioural animation. *Journal of Visualization and Computer Animation*, 13:287–300, 2002.

M Gleicher. Retargetting motion to new characters. In *Proceedings of SIGGRAPH 1998*, pages 33–42. ACM Press / ACM SIGGRAPH, 1998.

M Gleicher, H J Shin, L Kovar, and A Jepsen. Snap-together motion: assembling run-time animations. In *Proceedings of Eurographics/SIGGRAPH Animation Symposium*, pages 181–188, 2003.

P Gmytrasiewicz and E Durfee. Rational coordination in multi-agent environments. *Autonomous Agents and Multi-Agent Systems*, 3(4):319–350, 2000.

P Gmytrasiewicz and E Durfee. Rational communication in multi-agent environments. *Autonomous Agents and Multi-Agent Systems*, 4:233–272, 2001.

R Grzeszczuk and D Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *Proceedings of SIGGRAPH 1995*, pages 63–70. ACM Press / ACM SIGGRAPH, 1995.

R Grzeszczuk, D Terzopoulos, and G Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of SIGGRAPH 1998*, pages 9–20. ACM Press / ACM SIGGRAPH, 1998.

I Guyon and A Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

S Haykin. *Neural Networks: A Comprehensive Foundation 2nd edition*. Prentice Hall: Upper Saddle River, NJ., 1999.

L W He, M Cohen, and D Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of SIGGRAPH 1996*, pages 217–224. ACM Press / ACM SIGGRAPH, 1996.

J Hodgins and N Pollard. Adapting simulated behaviors for new characters. In *Proceedings of SIGGRAPH 1997*, pages 153–162. ACM Press / ACM SIGGRAPH, 1997.

K Hornik, M Stinchcomb, and H White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

J Hu and M P Wellman. Online learning about other agents in a dynamic multiagent system. In *Proceedings of Second International Conference on Autonomous Agents*, pages 239–246, 1998.

C Isbell, C Shelton, M Kearns, S Singh, and P Stone. A social reinforcement learning agent. In *Proceedings of Fifth International Conference on Autonomous Agents*, pages 377–384, 2001.

D Isla and B Blumberg. New challenges for character-based AI for games. In *AAAI Spring Symposium on AI and Interactive Entertainment*, 2002.

D Isla, R Burke, M Downie, and B Blumberg. A layered brain architecture for synthetic creatures. In *Proceedings of IJCAI*, pages 1051–1058, 2001.

D L James and K Fatahalian. Precomputing interactive dynamic deformable scenes. *ACM Trans. on Graphics*, 22(3):879–887, 2003.

L Kaelbling, M Littman, and A Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

G Kaminka, M Veleso, S Schaffer, C Sollitto, R Adobbati, A Marshall, A Scholer, and S Tejada. Gamebots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45, 2002.

M Kasper, G Fricke, K Steuernagel, and E von Puttkamer. A behavior-based mobile robot architecture for learning from demonstration. *Robotics and Autonomous Systems*, 34: 153–164, 2001.

B Kerkez and M Cox. Incremental case-based plan recognition with local predictions. *International Journal of Artificial Intelligence Tools: Architectures, languages, algorithms*, 12(4):413–463, 2003.

A Koschan and M Abidi. A comparison of median filter techniques for noise removal in color images. In *7th Workshop on Color Image Processing*, pages 69–79, 2001.

J Laird. It knows what you're going to do: Adding anticipation to the quakebot. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385–392, 2001.

M Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of Machine Learning '94*, pages 157–163, 1994.

M Mataric. Getting humanoids to move and imitate. *IEEE Intelligent Systems*, pages 18–24, July 2000.

G Matthews. *Personality, Emotion, and Cognitive Science*. Elsevier, Amsterdam, 1997.

A Meltzoff and M Moore. Early imitation within a functional framework. *Infant Behavior and Development*, 15:479–505, 1992.

R Metoyer and J Hodgins. Reactive pedestrian path following from examples. In *Proccedings of Computer Animation and Social Agents*, pages 149–156, 2003.

J Millar, J Hanna, and S Kealy. A review of behavioural animation. *Computer and Graphics Journal*, 23:127–143, 1999.

M Minsky. *Society of Mind*. Simon & Schuster, New York, 1985.

T Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

J Monzani, A Caicedo, and D Thalmann. Integrating behavioural animation techniques. *Computer Graphics Forum*, 20(3), 2001.

S Musse and D Thalmann. Hierarchical model for real time simulation of virtual human crowds. *IEEE Trans. on Vis. and Computer Graphics*, 7(2), 2001.

L Nadel. *Encyclopedia of Cognitive Science*. Nature Pub. Group, London., 2003.

A Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

M Nicolescu. *A Framework for Learning From Demonstration, Generalization and Practice in Human-Robot Domains*. PhD thesis, University of Southern California, 2003.

K Perlin and A Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of SIGGRAPH 1996*, pages 205–216. ACM Press / ACM SIGGRAPH, 1996.

A Pina, E Cerezo, and F Seron. Computer animation: from avatars to unrestricted autonomous actors (a survey on replication and modelling mechanisms). *Computers and Graphics Journal*, 24:297–311, 2000.

R Price. *Accelerating Reinforcement Learning through Imitation*. PhD thesis, University of British Columbia, 2002.

A Rao and M Georgeff. BDI agents: From theory to practice. In *Proceedings of the First Intl. Conference on Multi-Agent Systems*, 1995.

J H Reif and H Wang. Social potential fields: A distributed behavioral control for autonomous robots. *Robotics and Autonomous Systems*, 27:171–194, 1999.

L Reissell and D Pai. Modeling stochastic dynamical systems for interactive simulation. *Computer Graphics Forum*, 20(3):339–348, 2001.

C Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Proceedings of SIGGRAPH 1987*, pages 25–34. ACM Press / ACM SIGGRAPH, 1987.

C Reynolds. Competition, coevolution and the game of tag. In *Proceedings of Artificial Life IV*, pages 59–69, 1994.

M Rovatsos, G Weib, and B Wolf. Multiagent learning for open systems: A study on opponent classification. In D.Ǩudenko E.Ãlanso and D. Kazakov, editors, *Lecture Notes on AI 2636*, pages 66–87. Springer, 2003.

D Rumelhart, G Hinton, and R Williams. Learning internal representations in error backpropagation. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1:318–362, 1986.

S Russell and P Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

A Safonova, J Hodgins, and N Pollard. Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. *ACM Trans. on Graphics*, 23(3), 2004.

C Schaffer. A conservation law for generalization performance. In *Proceedings of the Eleventh International Conference on Machine Learning (ML'94)*, 1994.

P Schyns, R Goldstone, and J Thilbaut. The development of features in object concepts. *Behavioral and Brain Sciences*, 21(1):1–54, 1998.

S Sen and N Arora. Learning to take risks. In *Collected papers from AAAI-97 workshop on multiagent learning*, pages 59–64, 1997.

K Sims. Evolving virtual creatures. In *Proceedings of SIGGRAPH 1994*, pages 15–22. ACM Press / ACM SIGGRAPH, 1994.

P Stone. *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*. MIT Press, Cambridge, MA, 2000.

P Stone and M Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 1997.

M Sung, M Gleicher, and S Chenney. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3), 2004.

R Sutton and A Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

D Terzopoulos. Artificial life for computer graphics. *Communications of the ACM*, 42(8): 33–42, 1999.

G Tesauro. Temporal difference learning in TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

C Thornton. Indirect sensing through abstractive learning. *Intelligent Data Analysis*, 7(3): 1–16, 2003.

B Tomlinson and B Blumberg. Alphawolf: Social learning, emotion and development in autonomous virtual agents. In *First GSFC/JPL Workshop on Radical Agent Concepts*, 2002.

T Tran and R Cohen. A reputation-oriented reinforcement learning strategy for agents in electronic marketplaces. *Computational Intelligence*, 18(4):550–565, 2002.

X Tu and D Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of SIGGRAPH 1994*, pages 43–50. ACM Press / ACM SIGGRAPH, 1994.

M van de Panne and E Fiume. Sensor-actuator networks. In *Proceedings of SIGGRAPH 1993*. ACM Press / ACM SIGGRAPH, 1993.

M van de Panne, R Kim, and E Fiume. Synthesizing parameterized motions. In *Proceedings of 5th Eurographics Workshop on Simulation and Animation*, 1994.

M van Lent and J Laird. Learning procedural knowledge through observation. In *Proceedings of International Conference On Knowledge Capture*, pages 179–186. ACM Press, 2001.

J Vidal and E Durfee. Agents learning about agents: A framework and analysis. In *Collected papers from American Association for Artificial Intelligence (AAAI-97)*, pages 71–76, 1997.

C Watkins and P Dayan. Q-learning. *Machine Learning*, 8, 1992.

G Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

D R Wilson and T Martinez. An integrated instance-based learning algorithm. *Computational Intelligence*, 16(1):1–28, 2000.

B Yoon. Real world learning: exploratory efforts. In *Proceedings of DARPA Cognitive Systems Conference*, 2003.

B Yoon, R Burke, and B Blumberg. Interactive training for synthetic characters. In *Proceedings of AAAI*, pages 249–254, 2000.

T Zhu, R Greiner, and G Haubl. Learning a model of a web user's interests. In *Proceedings of Ninth International Conference on User Modeling*, 2003.