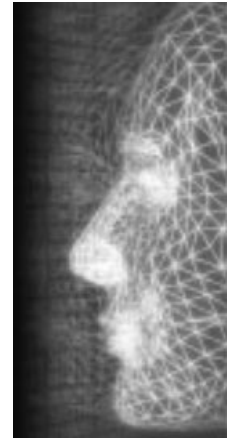


Fast and learnable behavioral and cognitive modeling for virtual character animation

By Jonathan Dinerstein*, Parris K. Egbert, Hugo de Garis and Nelson Dinerstein



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. Copyright © 2004 John Wiley & Sons, Ltd.

Received: May 2003; Revised: September 2003

KEY WORDS: computer animation; synthetic characters; behavioral modeling; cognitive modeling; machine learning; reinforcement learning

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 lifelike properties (walking lamps, etc). Their uses include entertainment, training, and simulation. 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 char-

acters 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.

*Correspondence to: Jonathan Dinerstein, Brigham Young University, 3366 TMCB, Provo, UT 84602, USA.
E-mail: jondinerstein@yahoo.com

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 suboptimal 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 suboptimal 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 *offline 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 offline 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.

Related Work

Previous computer graphics research in the area of autonomous virtual characters includes automatic generation of motion primitives.¹⁻⁷ This is useful for reducing the work required by animators. More recently, Faloutsos *et al.*⁸ present a technique for learning the preconditions 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 multilevel 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.⁹⁻¹² 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 prespecified 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). Online behavioral learning has only begun to be explored in computer graphics.¹³⁻¹⁵ A notable example is Blumberg *et al.*,¹⁶ 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 online, 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.¹⁷ 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.

Introduction to Cognitive Modeling

Cognitive modeling¹⁷⁻²⁰ 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 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.

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.*¹⁷). 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

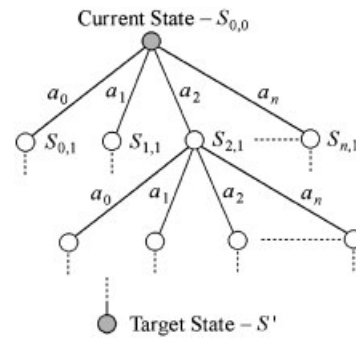


Figure 1. Planning is performed with a tree search of all candidate actions throughout time. To perform planning in real time without dedicated hardware, it is usually necessary to greatly limit the number of candidate actions and to only formulate short (suboptimal) plans.

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 (suboptimal) 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 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).

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.²² A more thorough introduction can be found in Grzeszczuk *et al.*⁵ There are many libraries and applications publicly available* (free and commercial) for constructing and executing artificial neural nets.

*For example, SNNS (<ftp.informatik.uni-tuebingen.de/pub/SNNS>) and Xerion (<ftp.cs.toronto.edu/pub/xerion>).

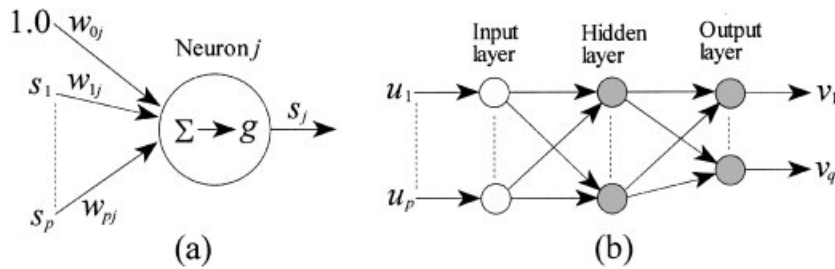


Figure 2. (a) Mathematical model of a neuron j . (b) A three-layer feedforward neural network of p inputs and q outputs.

A *neuron* can be modeled as a mathematical operator that maps $\mathbf{R}^p \rightarrow \mathbf{R}$. Consider Figure 2(a). 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 $\mathbf{u} = [1, s_1, s_2, \dots, s_p]$ is the input vector and $\mathbf{w}_j = [w_{0j}, w_{1j}, \dots, w_{pj}]$ is the connection weight vector. The neuron outputs a signal $s_j = g(z_j)$, where g is an activation function:

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

A *feedforward artificial neural network* (see Figure 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*.²³ Note that it has been proven that neural networks are universal function approximators (see Hornik *et al.*²⁴).

An alternative approach that we considered was to use the *continuous k-nearest neighbor algorithm*.²¹ Unlike neural nets, k -nearest neighbor 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 correctly reproduce the examples that it has been provided (whereas no such guarantee exists with neural nets). However, k -nearest neighbor requires the explicit sto-

rage of many examples of the target function. Because of this storage issue, we opted to use a neural net approach.

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 offline and then the trained network is used online. 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.

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 feed-forward neural net to approximate a specific policy μ . We denote the neural net approximation of the policy $\hat{\mu}$ (see Figure 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.

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 trade-off). We have found 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, then 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 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 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).

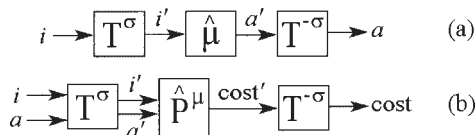


Figure 3. (a) Neural net approximation of a policy μ . The network input is the current state, the output is the action to perform. T^σ and $T^{-\sigma}$ normalize the input and denormalize the output, respectively. (b) Neural net approximation of a priority function.

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 of 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 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 5000 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 keep only 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.

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,²² 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 not only for reducing the total number of inputs but also for 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, in some cases, to 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 Mitchell²¹) 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.

Using the Neural Network

After a neural net is trained offline, it can be used online 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,¹⁷ 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 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 is 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.

Note that 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^{11,16}). 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.

Discussion

There is a great deal of pre-existing 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 suboptimal plans must be used. However, since in our technique we train a neural net offline, 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

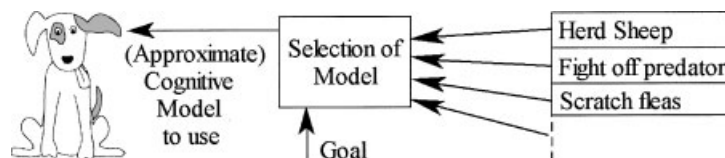


Figure 4. Example of a set of (approximate) cognitive models, integrated into a synthetic brain architecture for a virtual character (in this case, a sheepdog). The character's current goal determines which (approximate) cognitive/behavioral model will be used.

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 5000 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 Mitchell²¹ 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.

Offline Character Learning

In this section we introduce offline character learning for autonomous virtual characters. By *offline 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 offline 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 offline character learning.

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.¹⁻⁴ As an exam-

ple, 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.*¹⁶).

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 elsewhere^{26,27}). However, the most popular and general approach is known as *Q-learning*.²⁸

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 elsewhere.^{26,27}

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,²⁷ 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.

Offline Character Learning through Searching

The technique for offline 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,²⁷

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 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 offline, 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 Norvig²⁵ 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 offline 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’). 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.

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.^{21,27}

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 deter-

mine 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.

Discussion

Our planning-based character learning technique has proven to be fast, simple, and robust (see the Results section of this paper 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; offline learning required 1 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 suboptimal 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.

Experimental Results

We implemented our techniques (rapidly approximating an explicit cognitive model and offline 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). 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 OpenGL²⁹ on a 1.7GHz PC with an ATI Radeon 8500 (commodity) video card. Quicktime videos are available from <http://neptune.cs.byu.edu/egbert/a3dg/projects/bcm/animation.html>.

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.*,¹⁷ 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 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 discrete). 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 discrete). 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.



Figure 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 offline character learning. The neural net approximation was significantly faster to execute than the explicit cognitive model (about $1 \mu\text{s}$ vs. 0.2 s). Also, character learning required less time than for us to program an explicit model (a few hours vs. a few days).

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 3 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.

Spaceship Pilot and Asteroids

In this experiment, the virtual character was a spaceship pilot (see Figure 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 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 nine possible actions the pilot could do). We also had to limit planning to a tree depth of five 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 offline 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-grained 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 (offline character learning). The fitness function



Figure 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 determined by how direct 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.

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., Reynolds³⁰). 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 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 1 microsecond per iteration). Due to this performance, we were able to achieve a spaceship battle of 'epic proportions' on our PC (see Figure 7). Specifically, we had 2000 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.

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 char-

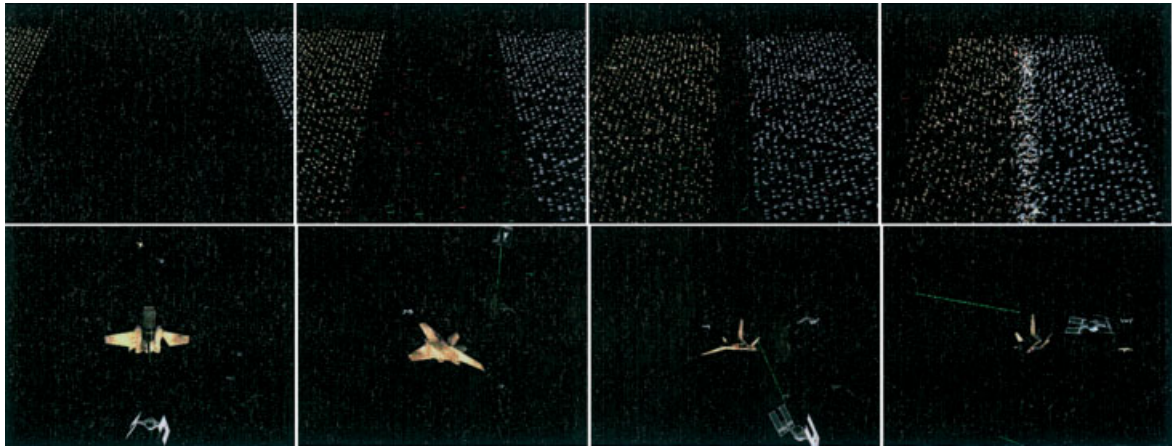


Figure 7. With the high computational speed of our technique, it is possible to perform animations of an epic scale in real time. Here we have 2000 spaceship pilots in combat. Our pilots are self-taught (i.e., character learning), so this animation took very little human effort to create.

acters in real time on a PC). Further, because training is done offline, we can use much larger action spaces and higher-quality plans than previously possible.

The second technique we have introduced is offline 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 offline 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 offline character learning. Online learning in the literature has thus far been limited to behavioral models (short-term utility). Is it possible to learn a cognitive model online 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.

References

1. van de Panne M, Fiume E. Sensor-actuator networks. In *Proceedings of ACM SIGGRAPH*, 1993.
2. van de Panne M, Kim R, Fiume E. Synthesizing parameterized motions. In *Proceedings of 5th Eurographics Workshop on Simulation and Animation*, 1994.
3. Sims K. Evolving virtual creatures. In *Proceedings of ACM SIGGRAPH*, 1994; pp 15–22.
4. Grzeszczuk R, Terzopoulos D. Automated learning of muscle-actuated locomotion through control abstraction. In *Proceedings of ACM SIGGRAPH*, 1995; pp 63–70.
5. Grzeszczuk R, Terzopoulos D, Hinton G. NeuroAnimator: fast neural network emulation and control of physics-based models. In *Proceedings of ACM SIGGRAPH*, 1997; pp 9–20.
6. Hodgins J, Pollard N. Adapting simulated behaviors for new characters. In *Proceedings of ACM SIGGRAPH*, 1997; pp 153–162.
7. Gleicher M. Retargetting motion to new characters. In *Proceedings of ACM SIGGRAPH*, 1998; pp 33–42.

8. Faloutsos P, van de Panne M, Terzopoulos D. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH*, 2001; pp 39–48.
9. Reynolds C. Flocks, herds, and schools: a distributed behavioral model. In *Proceedings of ACM SIGGRAPH*, 1987; pp 25–34.
10. Tu X, Terzopoulos D. Artificial fishes: physics, locomotion, perception, behavior. In *Proceedings of ACM SIGGRAPH*, 1994; pp 43–50.
11. Blumberg B, Galyean T. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of ACM SIGGRAPH*, 1996; pp 47–54.
12. Perlin K, Goldberg A. Improv: a system for scripting interactive actors in virtual worlds. In *Proceedings of ACM SIGGRAPH*, 1996; pp 205–216.
13. Burke R, Isla D, Downie M, Ivanov Y, Blumberg B. Creature smarts: the art and architecture of a virtual brain. In *Proceedings of the Computer Game Developers Conference*, 2001.
14. Yoon S, Burke R, Blumberg B. Interactive training for synthetic characters. In *Proceedings of AAAI*, 2000; pp 249–254.
15. Tomlinson B, Blumberg B. Alphawolf: social learning, emotion and development in autonomous virtual agents. In *Proceedings of First GSFC/JPL Workshop on Radical Agent Concepts*, 2002.
16. Blumberg B, Downie M, Ivanov Y, Berlin M, Johnson M, Tomlinson B. Integrated learning for interactive synthetic characters. In *Proceedings of ACM SIGGRAPH*, 2002; pp 417–426.
17. Funge J, Tu X, Terzopoulos D. Cognitive modeling: knowledge, reasoning, and planning for intelligent characters. In *Proceedings of ACM SIGGRAPH*, 1999; pp 29–38.
18. Funge J. *AI for Games and Animation: A Cognitive Modeling Approach*. A. K. Peters: Natick, MA, 1999.
19. Terzopoulos D. Artificial life for computer graphics. *Communications of the ACM* 1999; **42**(8): 33–42.
20. Funge J. Cognitive modeling for games and animation. *Communications of the ACM* 2000; **43**(7): 40–48.
21. Mitchell T. *Machine Learning*. McGraw-Hill: New York, 1997.
22. Haykin S. *Neural Networks: A Comprehensive Foundation* (2nd edn). Prentice-Hall: Upper Saddle River, NJ, 1999.
23. Rumelhart D, Hinton G, Williams R. Learning internal representations in error back-propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* 1986; **1**: 318–362.
24. Hornik K, Stinchcomb M, White H. Multilayer feedforward networks are universal approximators. *Neural Networks* 1989; **2**: 359–366.
25. Russell S, Norvig P. *Artificial Intelligence: A Modern Approach*. Prentice-Hall: Upper Saddle River, 1995.
26. Bertsekas D, Tsitsiklis J. *Neuro-Dynamic Programming*. Athena Scientific: Belmont, MA, 1996.
27. Sutton R, Barto A. *Reinforcement Learning: An Introduction*. MIT Press: Cambridge, MA, 1998.
28. Watkins C, Dayan P. Q-learning. *Machine Learning* 1992; **8**: 279–292.
29. Woo M, Neider J, Davis T, Shreiner D. *OpenGL Programming Guide* (3rd edn). Addison-Wesley: Reading, MA, 1999.
30. Reynolds CW. Competition, coevolution and the game of tag. In *Proceedings of Artificial Life IV*, 1994; pp 59–69.

Authors' biographies:



Jonathan Dinerstein is a PhD candidate in the Computer Science Department at Brigham Young University. He received his BS and MS degrees in Computer Science from Utah State University. His research interests include computer animation, AI-based animation, and machine learning.



Parris Egbert is Associate Department Chair, and an Associate Professor in the Computer Science Department at Brigham Young University. He received his BS in Computer Science from Utah State University. He received his MS and PhD degrees in Computer Science from University of Illinois at Urbana Champaign. His research interests include computer animation, visualization, and image-based modeling and rendering.



Hugo de Garis is an Associate Professor in the Computer Science Department at Utah State University. He received his Bachelors in Applied Mathematics and Theoretical Physics from Melbourne University. He received his PhD in Computer Science from Brussels University.



Nelson Dinerstein is an Associate Professor in the Computer Science Department at Utah State University. He received his Bachelors and Masters in Mathematics from the University of Massachusetts. He received his PhD in Mathematics from the University of Utah.