Learning to Move Autonomously in a Hostile World

Leslie Ikemoto[1]
University of California at Berkeley

Okan Arikan[2]
University of California at Berkeley

David Forsyth[3]
University of Illinois at Urbana-Champaign

Report No. UCB/CSD-5-1395

June 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

This paper describes a framework for controlling autonomous agents. We estimate an optimal controller using a novel reinforcement learning method based on stochastic optimization. The agent's skeletal configurations are taken from a motion graph which contains seamless transitions, guaranteeing smooth, natural-looking motion. The controller learns a parametric value function for choosing transitions at the branch points in the motion graph. Since this query can be completed quickly, synthesis is performed online in real-time. We couple the local controller with a global path planner to create a system which produces realistic motion even in a rapidly changing environment.

Keywords: Motion synthesis, reinforcement learning

1 Introduction

As virtual worlds become richer and more complex, the agents that populate them face increasingly complicated control decisions. Not only must they interact with the environment in a natural way, but they are also often tasked with complex objectives, such as hunting monsters or exploring a labyrinth.

Many of these interactions and objectives can easily be phrased as a *reward function*. The agent receives positive scores for accomplishing goals, such as reaching a rendezvous point. He is punished with negative scores for engaging in harmful interaction with the environment (such as bumping into solid objects), and for committing actions which detract from his goals (like being seen by enemies). An optimal controller will generate a motion that maximizes the agent's reward. *Reinforcement learning* harbors several well-studied methods for calculating an optimal or near-optimal controller from a state space and reward function.

In our system, we place an agent into a hostile environment that contains both obstacles and enemies. We introduce a framework in which we estimate a controller for the agent using reinforcement learning. The agent's skeletal configurations are taken from a motion graph ([Kovar et al. 2002], [Lee et al. 2002], [Arikan and Forsyth 2002]), which contains seamless transitions, guaranteeing smooth motion. The controller determines how the motion graph is traversed. We perform the estimation by sampling control parameters for a series of randomized scenarios. The control parameters that yield the maximum reward for each state are written into a scattered data interpolator. During synthesis, we query the interpolator for control parameters every time we reach a branching point in the motion graph. Since this query can be completed quickly, synthesis is performed online in *real-time* (Figures 1 and 2).

One of the major advantages of this framework is that it addresses the *short-horizon* problem inherent in local search. When a branching point in a motion graph is reached, one can look only a few frames into the future in real-time. The long-term effects of choosing a branch are hidden. We attack this problem by estimating the long-term effects during training.

Additionally, we combine our local controller with a *global path planner*, yielding two levels of control. In computer games, the path planner in an autonomous entity is separate from the controller. The path planner computes an optimal path, and the local controller tries to follow it. However, the local controller may not have immediate access to motions that will meet the path plan, and will therefore produce awkward sequences. In our framework, the two are strongly coupled, because the local controller periodically queries the global path planner for an updated plan.

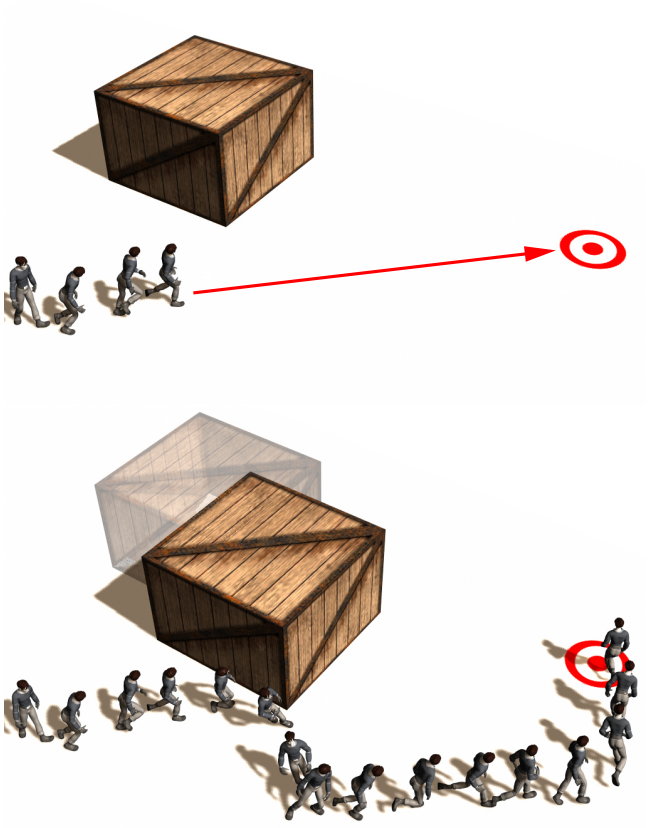

Figure 1: **Avoiding an Obstacle.** This sequence was recorded from a live, interactive demo, in which the user controls the crate and can move it anywhere at anytime. A virtual agent is tasked with traveling from the left of the scene to the target on the right. While the agent is running towards the crate (**top**), the user suddenly moves it into the agent's path. The controller successfully replans the motion on the fly. The agent not only dodges the crates and meets the target position (**bottom**), but does so seamlessly, since transitions are derived from a pre-computed motion graph. This sequence further illustrates that the controller has learned to balance competing goals of (a) reaching the target position quickly, and (b) avoiding obstacles.

2 Related Work

The literature in *data-driven synthesis* ([Pullen and Bregler 2002], [Arikan et al. 2003], [Molina-Tanco and Hilton 2000], [Li et al. 2002]) reveals a continuum of graph-based search methods, ranging from local to global. For example, [Kovar et al. 2002] consider local information only, while [Lee et al. 2002] appraise a longer horizon by expanding their search tree to a fixed depth. At the global end of the spectrum, [Arikan and Forsyth 2002] compute the entire motion at once. Our method falls into the local side of the continuum, but considers a longer horizon than [Lee et al. 2002] by pre-computing the expected future value of choosing a particular frame. [Lee et al. 2002] and [Arikan and Forsyth 2002]

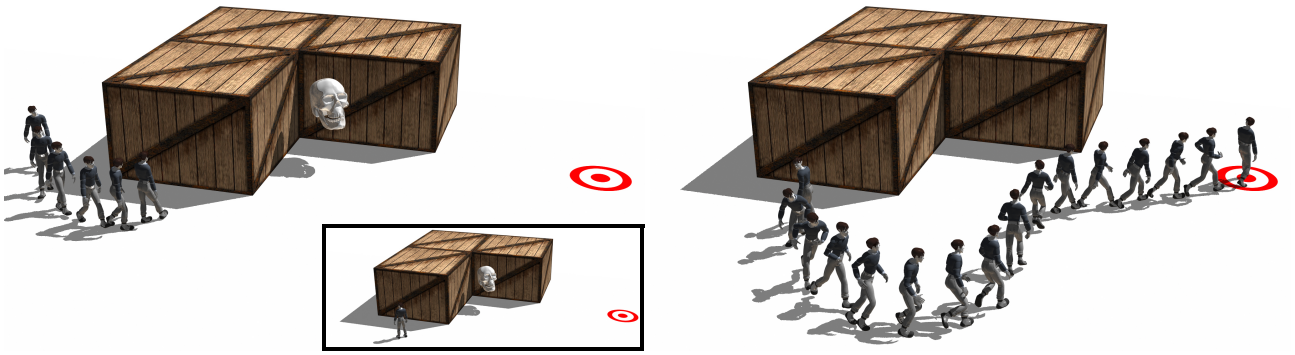

Figure 2: **Hiding from an Enemy.** In this scenario, the agent must hide from the enemy skull who scares him. On the **left**, the agent begins walking towards the goal, but notices said skull around the corner. He hides just behind the crate (**inset**) until the enemy disappears. The controller then leads the agent to his goal position (**right**). This sequence demonstrates that the controller *independently* learned the emergent behavior of using obstacles to hide from enemies; behaviors are not explicitly encoded. Note that this controller is the same one that produced the sequence in Figure 1.

also describe methods for allowing an avatar to interact with its environment, but these interactions are pre-recorded. We address the problem of interacting with arbitrary environments.

In the area of *autonomous motion*, several papers address the problem of learning behaviors to achieve objectives ([Funge et al. 1999], [Terzopoulos 1999], [Blumberg and Galyean 1995], [Go et al. 2004], [Mataric 2000]). [Reynolds 1987] did seminal work in this field; he controlled animals in a flock by blending between different behavioral forces. [Terzopoulos et al. 1994] successfully demonstrate the animation of autonomous fish in a virtual marine environment. More recently, [Lee and Lee 2004] describe a method that uses dynamic programming to animate a human boxing avatar. They train their controller to perform two behaviors: (a) moving towards a target, and (b) hitting it. However, cleanly dividing actions into behaviors is very difficult. A controller should learn how to achieve goals for itself (*e.g.*, it should learn on its own that it first needs to step in range of a target before hitting it). Furthermore, the mapping between behaviors and motivations quickly becomes complex as motivations become longer-term and more difficult to satisfy.

Finally, our system is inspired by work in the well-established field of *reinforcement learning*. In the standard framework, an agent interacts with his environment to learn a policy for choosing actions that will increase his expected reward. [Sutton and Barto 1998] provide a good overview of the work in this area. [Baxter and Bartlett 1999] and [Baxter and Bartlett 2001] demonstrate a technique for learning a parametric value function. We implement some of their ideas in this paper.

3 Overview

In our system, we would like the agent to (a) avoid obstacles, (b) hide from hostile enemies, and (c) reach target positions. (A sample environment is shown in Figures 1 and 2.) The controller must determine how best to meet these objectives. At every branching point in the motion graph, it chooses one of the agent's available alternatives. Each alternative yields a reward. The goal is to maximize the agent's expected reward. Therefore, the controller cannot simply choose the alternative with the highest immediate reward, because it may lead to small future rewards. Instead, we would like it to choose the alternative that will generate the highest cumulative reward. In reinforcement learning, the expected cumulative reward of an alternative is called its *value*.

In our system, we evaluate a reward function r for each frame in the sequence. The reward function penalizes the agent for walking into objects or for being seen by an enemy. It also punishes the agent for his distance to the current waypoint on the path that the global path planner determined. The global path planner has knowledge of the current obstacle configuration only; it has no information about the future, nor does it predict configurations. The local controller asks the path planner to update the plan periodically.

Based on the agent's position at the frame (p), intersections with objects (I), enemy visibility (F), and distance from the waypoint (y), r is defined as:

$$r = -w_I * I(p) - w_F * F(p) - w_Y * \| y - p \| \qquad (1)$$

Each term is weighted with w_I, w_F, or w_Y based on how much it affects the agent's objectives. We set w_I and w_F very large because either event is very detrimental. w_Y is fixed at 1 for scale invariance.

Unfortunately, calculating the value of an alternative is difficult because the controller does not know what the agent's future truly entails. There are several well-known algorithms in reinforcement learning that address this problem. In the standard reinforcement learning framework, a controller learns how to achieve goals by having the agent interact with its world and collect rewards. The controller then uses those rewards to estimate a *value function* that outputs the value of an alternative given the agent's current state. For example, a proper value function returns a low value for an alternative which leads the agent through solid walls (Figure 3).

It is difficult to apply classic methods, such as value-iteration, policy-iteration, and dynamic programming, to our problem domain because of the large number of possible states in which the agent could find himself. [Baxter and Bartlett 1999] attack this problem using a parametric value function, which they optimize using gradient descent. Unfortunately, computing a reliable gradient in our domain is too computationally expensive, again because of the vast state space.

Luckily, many of the states are similar to each other. We describe a revised reinforcement learning algorithm in Section 5 that takes advantage of that similarity to successfully estimate an optimal controller. At heart, the algorithm is a form of stochastic optimization, which samples weights for a parametric value function. Because nearby states are similar to each other, we expect the value function to change slowly as the state changes. Therefore, we can safely smooth it (see Equation 2).

Figure 3: **Choosing States.** When the agent reaches a branching point in the motion graph, the controller evaluates the available alternatives. In this example, there are three: walking backwards to the left (**blue**), walking to the right then turning left (**green**), and walking to the right (**red**). The controller estimates the value of each path by predicting where each obstacle and visible enemy will be in one second, assuming they continue moving at their current velocities. For each alternative, the controller calculates the position of the agent after one second. Here, the controller predicts that a crate will land in the path of the green alternative. A good value function will therefore assign that choice a lower expected value than the blue or red paths.

4 Evaluating Alternatives

When the agent reaches a branching point in the motion graph, we would like his controller to choose the highest-valued alternative. The value the controller assigns to the agent's alternatives should naturally depend upon the elements in the environment that affect the agent's reward. Such elements are encapsulated in a *state* representation. The agent's controller must learn to distinguish desirable states with a high value from undesirable ones with a low value. To do this, the controller learns a value function which it uses to evaluate states.

Choosing a good state representation (*i.e.*, one that is strongly correlated to the expected reward) is very important. Without it, the controller will be unable to learn profitable strategies because it will appear as though each state produces a random reward. Additionally, it is important to choose a compact representation, since the controller needs a good estimate of the value of *all* the states.

The state representation we use encapsulates information about the obstacles and enemies. The first part of the state describes the agent's local geometry, which we store as a vector of shape context data [Belongie et al. 2002]. That is, we place bins locally in the nearby space around him. If the center of a bin falls within an object, we mark that bin as occupied. The bin data is stored in a binary vector. The second part of the state contains information about the agent's last visible enemies. Specifically, we store a binary value indicating whether an enemy is visible from the agent's location. Our state representation is discrete, which permits us to enumerate all possibilities.

To evaluate the value function, the controller also asks for information about the goal from a high-level, omniscient path planner. The path planner determines the shortest path to the goal given the current obstacle configuration (it does not have any information about future configurations). The path is a set of waypoints connected by linear segments. When computing the value of an alternative, the controller uses the distance between the position of the agent and the next waypoint in the path.

The value function returns the goodness of choosing a particular branch given the agent's current state. In our framework, we predict a state that the agent will likely find himself in if he chooses the given branch (Figure 3). The value of the branch is the value of the predicted state given the current state.

More formally, the controller first computes the agent's current state s_t, which determines the parameters to use in the value function. The controller computes control parameters $\alpha(s_t)$, a vector that weights different components of the predicted state and environment. Its decision is affected more heavily by those aspects with higher weights. Section 5 describes our method for learning α.

For each possible transition, the controller determines a goodness value f by estimating the state s_{t+1} that results from taking the transition and traveling along the motion the transition lead to for one second.

To do this, the controller first predicts the agent's local geometry by computing his future world position and estimating future object locations. It extrapolates where each obstacle will be one second into the future, assuming they all continue moving at their current velocities. Once the controller has determined the local geometry portion of s_{t+1}, it calculates the enemy component by predicting the position of the agent's last visible enemy using the enemy's position and velocity. The controller then calculates whether there is a line of sight between the agent's future position and the enemy's predicted one.

We calculate f with the following continuous equation:

$$f(s_t, s_{t+1}) = \sum_{i=1}^{n-1} (\alpha_i(s_t) * K(c_i, s_{t+1})) + \alpha_n(s_t) \parallel y - p \parallel \qquad (2)$$

To evaluate f, we use the control parameters computed at s_t to weight the terms. The first terms are kernel functions that compare s_{t+1} to states c_i. c_i is a vector where the i^{th} component is 1 and every other element is 0. c_1 for example represents a state where there is an obstacle in bin 1 only. Giving these vectors to the kernel function isolates the effect of each component of the state. The kernel function returns 1 if the element in s_{t+1} corresponding to the sole non-zero component of c_i is 1, and returns 0 otherwise.

$\alpha_n(s_t)$ weights the distance between the player's future position (p) and the position of the next waypoint (y) on the global path plan. The controller computes f for every alternative the agent faces, then chooses the one with the largest f.

5 Estimating a Controller

We expect that nearby states will use similar control parameters. For example, the decisions the controller makes when the agent faces an obstacle 5 feet away should be similar to the decisions it makes when an obstacle is 7 feet away. Because of this similarity, we can estimate $\alpha(s_t)$ (which we used in Equation 2) by interpolating the control parameters for states nearby s_t. α is thus represented as a scattered data interpolator. During training, we populate the interpolator by sampling a set of disparate states. For each sample state, we sample control parameters and write the ones that generate the maximum reward into the interpolator.

More specifically, we first randomly distribute obstacles and enemies, then pick arbitrary starting and target positions for the agent. After recording the agent's starting state, we sample a random α and fix it for the entire motion graph. Fixing the control parameters leads to a reasonable approximation of the value of the starting state, provided that we do not travel too far in the motion graph, and that we discount future rewards. We limit the controller to run for four seconds to generate a motion sequence. To compute the reward earned for this sequence, we evaluate the reward function r

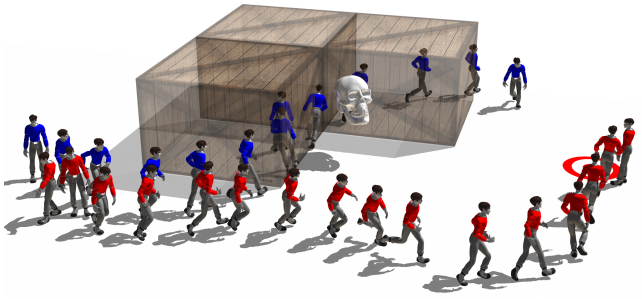

Figure 4: **Reduced-state Controllers.** We compare our controller to others with smaller state representations. The reduced-state controllers are trained with the same reward function as the full version. The first uses states which ignore the agent's local geometry (**blue**). This controller learned to avoid enemies by hiding the agent inside solid objects, which is unrealistic. The second controller uses states which ignore the agent's enemies (**red**). This controller immediately drives the agent in full-view of the skull to the target position, which is also undesirable.

(Equation 1) for each frame. Each frame's reward is discounted by γ^j, where j is the frame number. We sum the discounted rewards to get the reward for the sequence.

We then reset the agent's environment, and run the controller again using a different sample of control parameters. After executing this loop repeatedly (we sample 3000 parameter sets), we save the parameters that generated the maximum reward. We then choose a different starting state and sample the parameter space again. In total, we sample 25 states each at approximately 30 random skeletal configurations.

The parameters we saved for the sample states are stored in a scattered data interpolator. The scattered data interpolator provides our estimate to the function $\alpha(s)$. To compute the control parameters for a state s, it smoothes the parameters for nearby sample states s_i:

$$\alpha(s) = \frac{\sum_{i=1}^{k} w(s,s_i)\alpha(s_i)}{\sum_{i=1}^{k} w(s,s_i)}, \quad (3)$$

$$\text{where } w(s,s_i) = \frac{1}{\| s-s_i \|^2 + \varepsilon} \quad (4)$$

It is important to note that when we run the controller during training, we do not always choose the edge with the maximum f. A greedy controller will lead to a discontinuous value function, which is hard to interpolate. As [Baxter and Bartlett 1999] note, defining a probability distribution over the transitions renders the value function continuous. We define the probability of choosing a transition as the goodness of the transition divided by the sum of the goodness of all transitions.

6 Results and Discussion

We demonstrate our results on three environments, using a motion graph computed from motion capture sequences of standing, walking, and running. Our motion graph does not contain sequences for dodging obstacles or enemies. The agent's motion reflects the underlying motion graph. Our motion graph contains footskate in it. If a motion graph without footskate was used, there would not be any footskate in the agent's motion.

Scene	Full Controller	Ignore Obstacles	Ignore Enemies
Figure 2	-351.33	-3,761.20	-9,253.00
Figure 5	-215.00	-1,602.97	-1,935.4

Table 1: **Average Reward.** We run the full controller and two reduced-state controllers on two scenes. The first scene is run for 2000 frames, and the second scene for 5000. We then compute the average reward per frame that each controller earns. The reward function is given in Equation 1. Since the function penalizes for distance to the goal at every frame, it is impossible to earn the maximum score of 0 in these examples because the agent does not start at the target. The reward earned by the the full controller is an order of magnitude greater than the reward earned by either reduced-state controller.

In the first scene, the agent attempts to reach a target position while an adversarial user continually pushes a crate into his path (Figure 1). We recorded this sequence from a live, interactive session in which the user controlled the location of the crate. The controller begins leading the agent towards his target position when the user suddenly pushes the crate directly into the agent's path. On-the-fly, the controller replans the motion seamlessly, choosing frames that meet the agent's competing goals of (a) reaching the target position quickly, and (b) avoiding the crate. Note that we do not teach the controller specific behaviors; it learns to seek the goal and dodge obstacles on its own.

In the second scene, an enemy hides behind a group of crates (Figure 2). The agent is scared of being sighted by the enemy, and waits behind the corner of a crate, just out of sight. Once the enemy disappears, the agent continues on his path towards the target position. The motion is smooth and is a plausible response to the scenario. In this scene, it is evident that the controller has learned how to use obstacles to hide from enemies.

To determine whether our controller works, we compare it to controllers with smaller state representations. The first controller uses states which ignore the agent's local geometry, and the second uses states which ignore information about enemies. Both controllers are trained with the same reward function as our original controller (Equation 1). When we run the first controller on the scene described in the previous paragraph (Figure 4, in blue), the agent runs directly into the crates. This controller learned that an optimal strategy is to hide the agent inside the crates as much as possible to avoid the enemy. Thus, this sequence is unrealistic. When we run the second controller (Figure 4, in red), the agent runs to the target immediately, in full view of the enemy, violating one of his objectives. Both sequences are not only visually suboptimal, but also accumulate low rewards. Table 1 exhibits the average reward each controller collects in this scene. The full controller earns rewards that are an order of magnitude greater than either reduced-state controller.

The last scene contains a patrolling enemy and sliding crates (Figure 5). The agent waits until the enemy disappears, then runs through the moving crates to reach a target position. This scene demonstrates that our technique can handle rapidly changing environments on-the-fly. As was the case in the second scene, our controller earns a reward that is an order of magnitude greater than either reduced state controller.

We have demonstrated that we can learn a simple yet effective controller for a motion graph from a reward function and state space. As the accompanying video shows, we have created a real-time, online system that can adapt to a rapidly changing environment and drive the agent to achieve his goals.

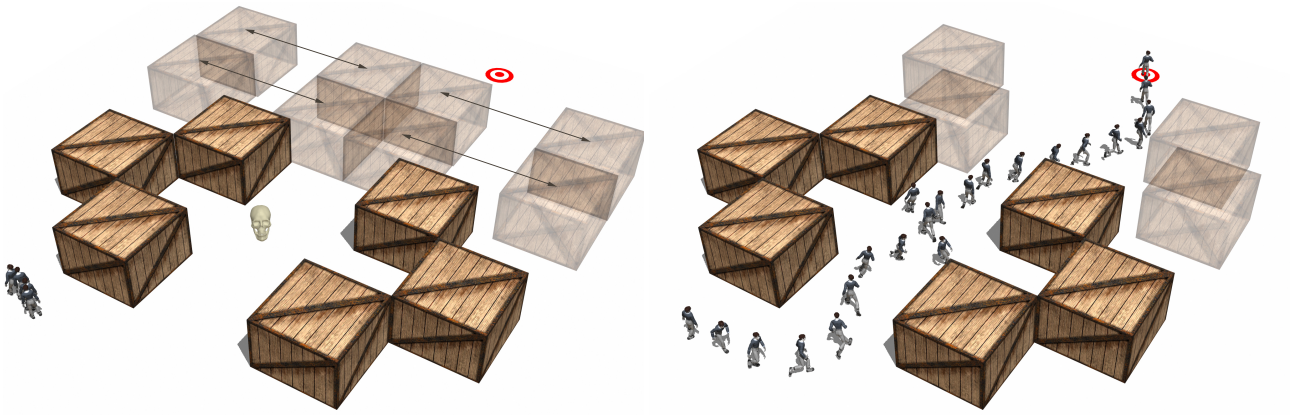

Figure 5: In this final scene, the agent waits behind a crate to avoid a patrolling enemy (**left**). Once the enemy disappears, the agent runs through the opening in the obstacles, then times his exit through the sliding crates (**right**). The sliding crates are shown as translucent, and the arrows on the left side indicate their range of motion.

References

ARIKAN, O., AND FORSYTH, D. 2002. Interactive motion generation from examples. In *Proceedings of ACM SIGGRAPH 02, 2002*.

ARIKAN, O., FORSYTH, D., AND O'BRIEN, J. 2003. Motion synthesis from annotations. *SIGGRAPH*.

BAXTER, J., AND BARTLETT, P., 1999. Direct gradient-based reinforcement learning.

BAXTER, J., AND BARTLETT, P. L. 2001. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research 15*, 319–350.

BELONGIE, S., MALIK, J., AND PUZICHA, J. 2002. Shape matching and object recognition using shape contexts. *IEEE T. Pattern Analysis and Machine Intelligence 24*, 4, 509–522.

BLUMBERG, B., AND GALYEAN, T. 1995. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of SIGGRAPH 1995*, 47–54.

FUNGE, J., TU, X., AND TERZOPOULOS, D. 1999. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Proceedings of SIGGRAPH 1999*, 29–38.

GO, J., VU, T., AND KUFFNER, J. 2004. Autonomous behaviors for interactive vehicle animations. In *ACM SIGGRAPH Symposium on Computer Animation*.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 473–482.

LEE, J., AND LEE, K. H. 2004. Precomputing avatar behavior from human motion data. In *Symposium on Computer Animation*.

LEE, J., CHAI, J., REITSMA, P., HODGINS, J., AND POLLARD, N. 2002. Interactive control of avatars animated with human motion data. *Proc. SIGGRAPH 2002*.

LI, Y., WANG, T., AND SHUM, H. Y. 2002. Motion texture: A two-level statistical model for character motion synthesis. In *Proceedings of SIGGRAPH 2002*, 465–472.

MATARIC, M. J. 2000. Getting humanoids to move and imitate. In *IEEE Intelligent Systems*, IEEE, 18–24.

MOLINA-TANCO, L., AND HILTON, A. 2000. Realistic synthesis of novel human movements from a database of motion capture examples. In *Workshop on Human Motion (HUMO'00)*, 137–142.

PULLEN, K., AND BREGLER, C. 2002. Motion capture assisted animation: Texturing and synthesis. *SIGGRAPH 02*.

REYNOLDS, C. W. 1987. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics 21*, 4, 25–34.

SUTTON, R., AND BARTO, A. 1998. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts.

TERZOPOULOS, D., TU, X., AND GRZESZCZUK, R. 1994. Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life 1*, 4, 327–351.

TERZOPOULOS, D. 1999. Artificial life for computer graphics. *Communications of the ACM 42*, 8, 32–42.