Character Animation in Two-Player Adversarial Games

KEVIN WAMPLER, ERIK ANDERSEN, EVAN HERBST, YONGJOON LEE, and ZORAN POPOVIĆ
University of Washington

The incorporation of randomness is critical for the believability and effectiveness of controllers for characters in competitive games. We present a fully automatic method for generating intelligent real-time controllers for characters in such a game. Our approach uses game theory to deal with the ramifications of the characters acting simultaneously, and generates controllers which employ both long-term planning and an intelligent use of randomness. Our results exhibit nuanced strategies based on unpredictability, such as feints and misdirection moves, which take into account and exploit the possible strategies of an adversary. The controllers are generated by examining the interaction between the rules of the game and the motions generated from a parametric motion graph. This involves solving a large-scale planning problem, so we also describe a new technique for scaling this process to higher dimensions.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*

General Terms: Algorithms

Additional Key Words and Phrases: Character animation, optimal control, game theory

ACM Reference Format:

Wampler, K., Andersen, E., Herbst, E., Lee, Y., and Popović, Z. 2010. Character animation in two-player adversarial games. ACM Trans. Graph. 29, 3, Article 26 (June 2010), 13 pages. DOI = 10.1145/1805964.1805970 http://doi.acm.org/10.1145/1805964.1805970

1. INTRODUCTION

Some of the most complicated and intricate human behaviors arise out of interactions with other people in competitive games. In many competitive sports, players compete for certain goals while simultaneously preventing the opponents from achieving their goals. These scenarios create very dynamic and unpredictable situations: the players need to make decisions considering both their own actions and the opponent's strategy, including any biases or weaknesses in the opponent's behavior. We propose that a mathematical framework based upon game theory is the appropriate choice to animate or control characters in these situations. Furthermore, we show that a game-theoretic formulation naturally accounts for real-world behaviors such as feints and other intelligent uses of nondeterminism which are ubiquitous in real life but have thus far been difficult to incorporate believably into games without significant hand-tuning. This is particularly of value in video games where an intelligent use of nondeterminism is an absolute necessity for a virtual character in a competitive situation.

The root assumption upon which our method is based is that the characters act *simultaneously*, in contrast to previous adversarial character animation techniques which model the players as taking turns. This closely matches the structure of many real-world games and sports, and captures the reason it often pays to be unpredictable in these games. In turn-based approaches the best way to act is always deterministic, and any randomness must be postprocessed in an ad hoc, and often difficult to hand-tune, manner. This significantly complicates the design of the animation controller and is prone to errors, leading to characters that don't behave randomly when they should or that choose randomness when it is not appropriate. By allowing simultaneous actions we arrive at a game-theoretic formulation which incorporates nondeterminism in its definition of optimal behavior. This not only allows for intelligent and random controllers to be automatically constructed, but also gives rise to emergent behaviors such as feints and quick footsteps which exploit unpredictability for their effectiveness.

The particular mathematical model we employ for character animation is known as a zero-sum Markov game. In this model each character acts according to the probability distribution that maximizes the likelihood of winning, assuming that the opponent is capable of this same line of reasoning and is attempting to stop them as effectively as possible. This approach also allows for an easy integration of long-term planning where characters choose their moves based not only on what will happen immediately but also taking into account what the future ramifications might be. This is necessary for the method to be applicable in real games, and gives rise to intelligent-looking anticipation, such as "leading" the motion of a runner in order to tackle them or planning a feint in a sword fight.

Unfortunately, building optimal game-theoretic controllers is hard because we plan for optimal policies by considering both adversaries simultaneously. This magnifies all the issues of high dimensionality, making it significantly harder than creating a controller for a single character. This is particularly problematic as existing MDP and Markov game planning algorithms require *exponential* time and storage in the dimension of the game's state space.

Authors' address: K. Wampler (corresponding author), E. Andersen, E. Herbst, Y. Lee, and Z. Popović, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350; email: wampler@cs.washington.edu.

We provide a new offline learning algorithm that employs an iterative function learning and compression heuristic. This algorithm effectively mitigates this difficulty and is particularly well suited to the sorts of problems encountered in character animation. By incorporating this with a model which uses the character's motion graph directly rather than relying on a simplified abstraction, we are able to produce detailed and realistic results.

Finally, we provide a method for altering the behavior of our characters in light of knowledge about an opponent's biases and inefficiencies. For example, if it is known that an opponent tends to prefer a specific move, the character can adjust its behavior so that this particular move is not successful. We believe that the ability to naturally account for randomness and other nuances resulting from competing characters will be of great use in defining complex, intelligent, unpredictable, and adaptable behaviors that contribute to the realism and uniqueness of repeated play experiences.

2. RELATED WORK

In a game it is desirable to have a character controller which behaves intelligently both in its motion choices and in its use of randomness. Previous approaches to animating characters that are competing against each other have typically approached the problem from a turn-based framework [Shum et al. 2008, 2007]. In a somewhat different approach, Liu et al. [2006] use a sequence of alternating spacetime optimizations to generate a tackling-dodging motion. In these methods an ordering is imposed on the characters' actions, forcing them to "take turns" in selecting their movements. This allows a deterministic strategy to be obtained using standard alternating minimax search techniques. In some cases this is appropriate, and it allows the authors generate some impressive animations, but in other cases these approaches can perform poorly. The difficulty with these methods is that the imposed ordering is often arbitrary and does not reflect the true nature of physical games in which actions are taken simultaneously. The result of this is twofold. Firstly the player "going second" in a turn-based approach has a distinct advantage, often leading to unrealistic motions in which the player seems to "predict" their opponent's moves before they happen. Secondly, optimal behavior in these approaches is always deterministic. By allowing actions to occur simultaneously we naturally arrive at a game-theoretic approach and remedy both of these problems. The nondeterministic policies seem to capture fakes, jukes, and quickstep motions that we expect in competitive games.

Attempts to layer nondeterminism on top of a deterministic framework have been proposed. In Shum et al. [2007], the authors suggest setting the probability that a motion will be chosen based on its score in the minimax search where each agent picks the single action which maximizes their reward based on the opponent's current state. Lee and Lee [2006] hand-label a small set of actions that the character can select from at random. Such approaches generally produce unsatisfactory results because the degree and nature of randomness required can vary in complicated ways depending on the states and available actions of the characters. For example, in some cases there is only a single motion that is appropriate, such as running directly away from an opponent in the game of tag. Other motions, such as turning or slowing down slightly, might result in a score only slightly worse than the best actions, but would look very odd indeed were a character to choose them. Conversely, there are some cases such as dodging around an oncoming tackler where it pays to be unpredictable and the character would wish to choose randomly between a left and a right dodge. Our game-theoretic framework naturally incorporates randomness into its definition of optimality and determines the stochastic policy which trades between diversity and effectiveness as necessitated by the state of the game.

Our approach is also related to reinforcement learning techniques which have been employed to generate intelligent single-character behavior in real time. Lee and Lee [2006] use value iteration to produce boxing avatars. Ikemoto et al. [2005] and Lau and Kuffner [2006] store the discounted sum of reward obtained by simulating for a short amount of time. McCann and Pollard [2007] construct a value function that adapts to user command patterns. Lee and Lee [2006] use a discrete representation to compute an optimal path for a boxer to hit a target, obtaining a two-character animation by having each character treat the other's body as the target. Treuille et al. [2007] and Lo and Zwicker [2008] produce compact representations for value functions to obtain a policy quickly at runtime. Our work extends these ideas to multiple characters to produce strategic behaviors not possible in single-character frameworks.

A particularly interesting single-character approach to controlling competitive characters is presented by Graepel et al. [2004]. They use reinforcement learning to gradually build an effective policy by having a character play a fighting game over and over. This approach does allow adaptation to an opponent's behavior, but also neglects the adversarial aspects to the game because it treats the opponent as a stochastic process rather than something capable of reasoning. This means that the generated controllers are still deterministic, and may require a large amount of training data before being applied in new situations.

3. OVERVIEW

We consider the problem of animating one character competing against another in a game in which each character seeks to do well while simultaneously preventing their adversary from doing the same. This means that any actions taken by the characters will be chosen not only by how well they help them perform in the game but also by how their opponents might counteract them. For convenience we will henceforth call one character the *agent* and the other the *opponent* and describe the game from the agent's point of view. However, the agent and opponent are interchangeable in our model and our technique can be used to animate either or both.

The fundamental assumption upon which our approach is based is that the characters choose their actions *simultaneously*, in contrast to the turn-based approaches previously used in character animation. This assumption better models the way that most real-world physical games are played and why it is often useful to behave unpredictably. As a simple illustration of this property, consider the difference between tic-tac-toe and rock-paper-scissors. The former is turn-based, and there is an optimal deterministic strategy. In rock-paper-scissors and in many real games, however, the lack of turns necessitates a nondeterministic strategy.

This insight leads to the use of a *Markov game* to model the character animation problem. In particular we employ a *zero-sum* Markov game in which two characters are competing with opposing goals. In such a game each player has at their disposal a set of available moves at each state, which we will refer to as *actions*. Our method is agnostic to the source of these actions, but in order to animate human characters we generate them by employing a parametric motion graph. In this scenario a character's actions at a state correspond to the available transitions through the motion graph starting at the character's current state and ending some short amount of time in the future. Our approach allows a continuous state space, performs its planning directly on the character's motion model, and fully captures the issue of simultaneous actions. This naturally yields nuanced and intelligent nondeterministic results.

We further wish to be able to control a character in real time. This is particularly important as one of the most interesting applications for an adversarial animation controller is in computer games. In order to allow real-time control while still allowing for long-term planning we precompute a structure known as a *value function* for the game. A value function is a general concept in planning algorithms, but in the case of adversarial games it represents a function which maps each possible state of a game into a real number representing the expected future reward of the agent assuming optimal play by both the agent and the opponent. This allows us to determine the expected long-term reward of an action by simply looking at the state immediately resulting from taking the action, and then querying the value function to determine that state's long-term expected value. Since the value function can be entirely precomputed offline, this method is very efficient at runtime.

In the remainder of the article we will begin by describing the parametric motion graph model used to drive our animations. Following this, we give a brief background on the fundamentals of game theory as it applies to our method. We then discuss the value function precomputation phase of our method, and in particular a new technique for estimating our value function in high-dimensional Markov games and MDPs that are required to apply our method to character animation. Finally we will show a simple technique that can incorporate knowledge of an opponent's biases and suboptimalities into our model and automatically plan against them.

4. MOTION MODEL

Before we discuss the game theory behind our animation technique we will describe the motion model we employ. Our method can be applied to any motion model which allows simultaneous actions, so we do not consider our particular choice to be a contribution of our work. Nevertheless, we hope that having a concrete model will aid in understanding the mathematics in Sections 5 and 7 for those unfamiliar with game theory and reinforcement learning.

In order to phrase a character animation problem as a game theory problem, we must define a motion model for each character describing how they can move. At each possible configuration of the players this motion model defines a set of actions that each of the players may take, as well as the motions that will result for each possible action choice. For our work we have chosen to use a Parametric Motion Graph (PMG) [Heck and Gleicher 2007; Shin and Oh 2006], since it provides a high degree of variability in the generated motions and uses a compact parameterized representation.

In our parametric motion graph, a node represents a space of possible motions, parameterized in one or more dimensions, for example, walking speed, and an edge represents a transition between two points in the parameter spaces of its start and end nodes, respectively. We consider a transition as happening instantaneously, but apply some blending between the two motions as a postprocessing step to help eliminate any visual artifacts.

A state within the graph is specified by (*node*, *time*, *parameters*), where *time* ranges from 0 to 1 within each node and *parameters* $\in R^m$, with m being the dimension of $node$'s parameter space. A node is constructed from a set of motion capture clips which have been aligned with dynamic time warping. Each clip is assigned a point in the node's parameter space and these clips are blended to generate a new clip at any point within the parameter space. To prevent generation of unrealistic motions, we only use convex combinations of the points assigned to the clips in each node.

We construct edges as in Heck and Gleicher [2007], but hand-select some pairs of nodes between which we allow transitions at multiple points in time. This allows us to create a graph with more possible transitions allowing for better responsiveness and greater motion variability. In experimenting with this motion representation we have found that we can represent a reasonably wide range of motions with each node. By interpolating only between the few clips closest to the desired point in a node's parameter space, we can approach the continuity of real motion. There is, though, a trade-off between the variety of motions represented by a node and its complexity: the more varied the motions in a node, the fewer dimensions in which it can be parameterized without producing unrealistic animation. In our games we limit our nodes to a one-dimensional parameter space. This allows us to have a parameterization that is simple to compute and reduces the number of motions required to construct each node while still providing a useful degree of variability within each node.

4.1 Action Lookahead

In order to apply our PMG-based motion model within a Markov game framework we must define what an action constitutes for a character. There is a hidden complexity in doing this for multiple characters which is not encountered in single-character contexts. Since we consider actions as occurring simultaneously, and since clips composing each node are of different durations, the transitions in two PMGs will not necessarily be synchronized. We therefore define an action not as a transition in the underlying PMG, but as a state which is reachable at some fixed interval of time in the future.

We construct such an action as a series of (possibly partial) transitions through clips in a motion graph that ends at some interval of time in the future. We refer to the length of interval as the *lookahead* of a game's actions. Generating this set of actions is straightforward: we simply consider all possible paths through the motion graph from the current state that end at the given time interval using depth-first search. In the case where a motion node is parameterized, we uniformly choose samples from its parameter space as links in the DFS. We have found that as few as three such samples are sufficient to define a useful controller.

In addition to ensuring that agent and opponent actions are synchronized, this method has a useful interpretation within the context of character behavior. The nondeterminism inherent in physical games is a result of the small amount of time required for a character to consider their possible choices and decide what to do. Because of this they must commit to an action before they know what the opponent is going to do. This "planning latency" coincides with the lookahead interval used to determine the agent and opponent action sets. In our tests, we achieve good results with a threshold of a quarter to a third of a second. Smaller lookahead values result in mostly deterministic games because a player can recover very quickly from an incorrect response to an opponent's feint. Larger lookahead values give games with a large degree of nondeterminism, but in which the players appear to respond too slowly to the actions of the opponent.

5. CONSTRUCTING OPTIMAL STRATEGIES

5.1 Background: Stateless Games

The core component of our method is based on using the motion model described in Section 4 to formulate a game-theoretic

problem. Since the language of game theory is not common in computer graphics, we will first describe the basics of the required mathematics in a more abstract form. A more detailed description of these concepts can be found in Sutton and Barto [1998] and any introductory text on game theory such as Morris [1994].

A standard pedagogical example of a zero-sum Markov game is rock-paper-scissors. In this game, as in all Markov games, two players simultaneously select one of a set of available *actions*, and a score is given based on these actions. Although both players have the same three actions available in rock-paper-scissors, in most games the players have different sets of available actions.

In a Markov game one player is arbitrarily labeled the *agent* and the game is described from this point of view. Any positive score awarded benefits the agent. Since we are using a zero-sum game this score also simultaneously counts against the *opponent*. At any point in time, we can describe the possible ways in which this score might be awarded based on the actions chosen by each player with a *reward matrix*. To construct this matrix we order the actions available to the agent and opponent and denote these action sequences with $\mathcal{A}$ and $\mathcal{O}$, respectively. Thus, for instance, $\mathcal{A}_i$ denotes the ith action available to the agent. The element of the reward matrix at row i and column j is the score given to the agent if they were to pick $\mathcal{A}_j$ while the opponent picked $\mathcal{O}_i$. In the case of rock-paper-scissors we can let $\mathcal{A} = \mathcal{O} = [r, p, s]$ and define the reward matrix, $\mathbf{R}$ as follows.

$$\mathbf{R} = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix} \tag{1}$$

Reward matrices such as these are used at runtime to determine how the characters in our games should move, and we are able to compute the optimal behavior for each character given such a matrix. One critical observation required to do this is that since the actions are chosen *simultaneously* the optimal behavior will in general be nondeterministic. This is illustrated in the game of rock-paper-scissors, and is in contrast to turn-based games such as tic-tac-toe where the optimal behavior is always deterministic.

We will specify the behavior of a player with a *policy* vector, typically denoted by π. Each element of such a policy gives the probability with which the player will choose the associated action, so for example if π is the policy for the agent, π_i gives the probability with which the agent will act according to $\mathcal{A}_i$.

Since a player in a game wishes to accumulate as much reward as possible, we find the policy which maximizes the expected value of their reward. Assuming that the agent acts according to policy π_a and the opponent acts according to π_o we can compute this expected reward as $v = \pi_o^T \mathbf{R} \pi_a$. Since we treat the agent and opponent symmetrically, we assume that the agent picks π_a so as to maximize v while the opponent simultaneously picks π_o to minimize it. Under these assumptions we can compute the policy π which maximizes the agent's expected reward despite the opponent's best attempts to the contrary using a simple Linear Program (LP).

$$\begin{aligned} &\max_{\pi, v} v \\ &\text{such that } \sum_i \pi_i = 1 \\ &\pi \succeq \mathbf{0} \\ &v \preceq \mathbf{R}\, \pi \end{aligned} \tag{2}$$

The first two constraints in this LP ensure that π is a valid probability distribution, while the notation $v \preceq \mathbf{R}\ \pi$ means that the inequality holds independently for each row of $\mathbf{R}$. The optimal policy for the opponent can be found by replacing $\mathbf{R}$ with $-\mathbf{R}^T$. In the aforementioned case for rock-paper-scissors this method gives the expected result of $\pi_a = \pi_o = [\frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}]^T$.

5.2 Background: Games With State

Although this technique gives the core of the method we use at runtime to control the motion of a character, the games described thus far do not allow any notion of the *state* of a game. In reality the actions available to each player as well as the possible rewards will generally change based on where the players are relative to each other, what motion each is currently performing, etc. Therefore the previous definition of a game must be generalized to allow the actions and reward to depend on the state of the game. In addition we must describe how a game's state changes over time based on the actions of the players.

We represent the set of all possible states that a game can take with a *state space*, denoted by $\mathcal{S}$. Each element x in this state space gives a complete state for the game, including the state of both players and any state of the environment. The actions available to the agent and opponent at this state will now be denoted with the functions $\mathcal{A}(x)$ and $\mathcal{O}(x)$, respectively. We generate these actions directly from each character's motion graph as described in Section 4.1. We also allow the possible rewards to depend on the state of the game by using a *reward function* $\mathbf{R}(x)$, which gives the reward matrix assuming that the game is in the state x. Typically this will be identical or closely related to the scoring function of the game. The number of actions in $\mathcal{A}(x)$ and $\mathcal{O}(x)$ and thus size of $\mathbf{R}(x)$ vary as a function of x. Finally, we define how the state of the game can change over time using a *transition function*, $y = \mathcal{P}(x, a, o)$, which gives the new state of the game, y, if the current state is x and the agent and opponent behave according to the actions $a \in \mathcal{A}(x)$ and $o \in \mathcal{O}(x)$. We define this transition function by having the characters move according to their motion graphs and additionally accounting for any necessary player-player interaction (such as handling sword collisions in our sword fighting game).

By considering games with a notion of state, we must also account for the fact that since the game changes over time a player's actions can impact the possibility for future rewards. This is in contrast to stateless games where we only have to consider immediate rewards. It is therefore necessary to change the definition and computation of the optimal policies for the players. In order to allow long-term planning in such a scenario we take an approach closely analogous to that already in use for single-character MDP controllers as in Treuille et al. [2007] and Lo and Zwicker [2008], and refer the reader to either of these papers for a more thorough discussion of these concepts in the context of single-character animation. In a game with state we define an "optimal" policy as that which maximizes the expected value of the infinite sum of the player's immediate and future rewards, where rewards t steps into the future are multiplied by a factor of γ^t. Here γ is a *discount factor* with $0 \leq \gamma < 1$ that determines how much the player cares about near-term versus long-term rewards.

In order to efficiently compute each player's policy at runtime, we take the same approach as in single-character animation and employ a *value function*. The value function V is defined so that $V(x)$ gives the expected value of the agent's future reward, discounted by γ^t, assuming both the agent and opponent play optimally. In Section 6 we will discuss how to compute such a value function for a Markov game, but for the moment note that given V we can solve for an optimal policy for the agent with future rewards taken into account

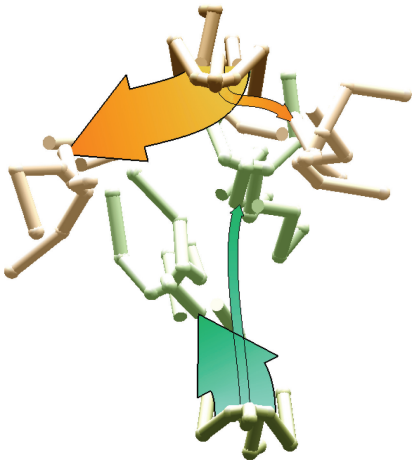

Fig. 1. An overhead view of a state in the tag game displaying the characters' policies. The width of each arrow corresponds to the probability that motion will be chosen. Notice how the agent (pictured top) picks randomly between two dodges so that the opponent cannot predict their movement.

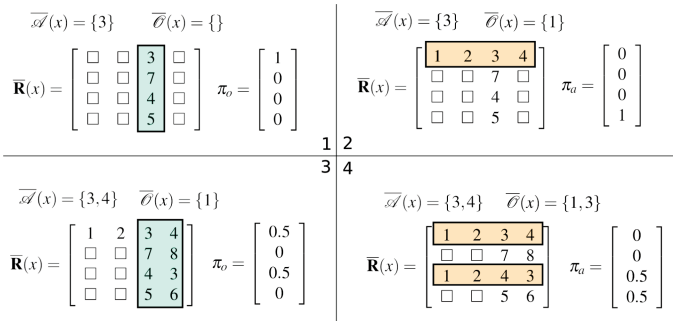

Fig. 2. An example of incrementally learning the player policies. The process terminates with the correct policies without computing the entire matrix (the missing elements are 5, 6, 7, and 8 in this case). For larger matrices it is common that a small fraction of the elements will have been computed when the algorithm terminates. The green and orange rectangles show the submatrices used to solve for the opponent and agent policies, respectively.

by solving a modification of Eq. (2) in which $\mathbf{R}$ is replaced with $\overline{\mathbf{R}}(x)$ [Lagoudakis and Parr 2002].

$$\overline{\mathbf{R}}(x)_{i,j} = \mathbf{R}(x)_{i,j} + \gamma V(\mathcal{P}(x, \mathcal{A}(x)_j, \mathcal{O}(x)_i)) \tag{3}$$

As is the case with stateless games, $\overline{\mathbf{R}}(x)$ represents the reward matrix for a zero-sum Markov game. The only difference is that $\overline{\mathbf{R}}(x)$ represents long-term expected reward, whereas $\mathbf{R}$ only represents instantaneous reward.

An illustration of typical policies computed using this method in a game of tag is shown in Figure 1.

5.3 Fast Markov Game Solutions

The linear program presented in Eqs. (2) and (3) is sufficient to solve for the policies by which the players in a game should move. It has the disadvantage, however, that it requires a computation time which is quadratic in the number of actions available to the players. This can make its use in a real-time scenario problematic. To allow for the easy use of these methods in real time, we present a heuristic algorithm which is more efficient in practice and easily allows for early termination.

This algorithm is very closely related to one of Sadovskii [1978]. Since this article is difficult to obtain in English, we will briefly describe it here. Those desiring further information may find another description in Petrosjan and Zenkevich [1996, pp. 41–43].

In this method, we do not compute the entire reward matrix all at once, but rather iteratively compute columns and rows of it by only considering subsets of the possible agent and opponent actions respectively, and then iteratively growing these subsets. We will denote these subsets of agent and opponent actions with $\overline{\mathcal{A}}(x)$ and $\overline{\mathcal{O}}(x)$ and initialize them by $\overline{\mathcal{A}}(x) = \{a_i\}$ for some random $a_i \in \mathcal{A}(x)$ and $\overline{\mathcal{O}}(x) = \{\}$.

We will alternately grow $\overline{\mathcal{O}}(x)$ and $\overline{\mathcal{A}}(x)$ by adding new actions to them, starting with $\overline{\mathcal{O}}(x)$. To add an action to $\overline{\mathcal{O}}(x)$, we consider the reward matrix formed by taking only the actions currently in $\overline{\mathcal{A}}(x)$, and the full set of opponent actions in $\mathcal{O}$. This matrix consists of a subset of the columns in the full reward matrix. If we solve for the optimal opponent policy given this reward matrix using Eq. (2) we get a probability for each possible opponent action. We then grow $\overline{\mathcal{O}}(x)$ by adding the opponent action not already in $\overline{\mathcal{O}}(x)$ which has the highest associated probability in this policy, unless this probability is zero in which case we leave $\overline{\mathcal{O}}(x)$ as is. We can then grow $\overline{\mathcal{A}}(x)$ by swapping the roles of the agent and opponent actions, now considering a subset of the rows (instead of columns) in the full reward matrix.

If we ever reach a case where the only actions which have a nonzero probability are already in $\overline{\mathcal{A}}(x)$ and $\overline{\mathcal{O}}(x)$, we terminate the algorithm and return the current policies for each player. Note that it is possible (and indeed very common) that this algorithm will terminate before we have considered all rows and columns in $\mathbf{R}(x)$, meaning that we often need to calculate only a fraction of all values in this matrix, as illustrated in Figure 2. In the games we have implemented this gives us a speedup of a factor if 3 to 5, and has the additional advantage that the larger speed increases are generally in cases where the players have many possible actions, and thus the method in Eq. (2) would be particularly slow.

Additionally, this algorithm may be terminated at any time and still provide a useful approximation to the correct policy. In fact, if we terminate the algorithm after the first iteration we achieve a solution which matches that of a turn-based algorithm, with an essentially equivalent computation time. This allows the implementer better flexibility in ensuring real-time performance while maintaining the ability to efficiently solve for the correct policies if desired.

A plot of the runtime requirements of this technique versus constructing a full reward matrix and solving Eq. (2) directly is given in Figure 3. The time required to compute the policies in our games is dominated by the computation of the reward matrix, so the ability of the iterative method to determine policies without computing every element in this matrix is very useful. Because we do not require new policies to be computed every frame, but only when the previously selected actions expire, the iterative method runs in real time for all the examples shown in this plot. The amount of time before an action expires is given by the lookahead time described in Section 4.1. In practice, we find that in both the tag and sword games solving for the policies of the players requires approximately 0.01s on average.

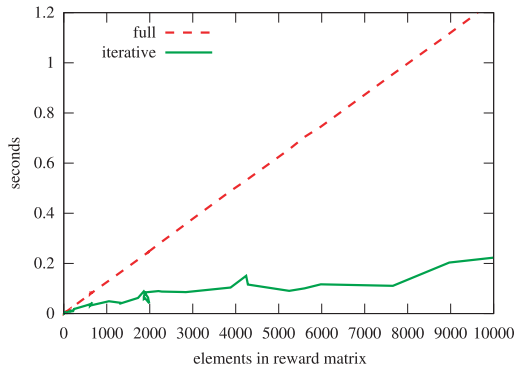

Fig. 3. A plot of the runtime requirements of the naive (dashed line) and iterative (solid line) methods for solving for the player policies. For larger reward matrices the iterative method is significantly faster. The reward matrices for this plot were generated from a tag-like Markov game where one player attempts to grab the other.

6. VALUE FUNCTION LEARNING

In Section 5.2 we used Eq. (3) to define the reward matrix for a Markov game at some given state, which in turn yields the optimal behavior for both characters. Constructing this matrix requires a *value function*, $V(x)$, which for each point, x, in the state space of the games gives the expected value of the sum of the agent's immediate and future rewards assuming both characters play optimally. At the time we deferred discussion of how we compute this value function, and we will now describe how this is achieved.

6.1 Background

The problem of learning a value function has already been addressed in the context of single-character animation based on Markov decision processes [Treuille et al. 2007; Lo and Zwicker 2008]. In doing so for the sorts of problems encountered in character animation (both single-character and two-player games) it is necessary to be able to learn a value function over a state space with continuous variables, such as the position of a character, and to be able to learn this value function in state spaces with a higher dimensionality than can be feasibly be directly represented with a table.

One technique which aids in both these goals was first used in the context of character animation by Treuille et al. [2007] and involves representing a value function with a linear combination of a set of *basis functions*. Given such a set of basis functions, $\phi = \{b_1(x), \ldots, b_n(x)\}$, we can approximately represent the value function with a vector $\mathbf{v}$ of length n such that $V(x) \approx \mathbf{v}_1 b_1(x) + \cdots + \mathbf{v}_n b_n(x)$. We solve for this weight vector in a least-squares sense using a method which minimizes the errors at a set of sample points randomly distributed throughout the game's state space. For the details of this method please see Appendix A.

By representing a value function with a linear combination of a set of basis functions we can easily learn an approximate value function for a game with a continuous state space while still representing the value function with a finite vector. Depending on which set of basis functions we choose, this approximation will resemble the "true" value function to a greater or lesser degree.

One obvious way to increase the accuracy of the approximate value function is simply to use more basis functions. This is an effective strategy for games with a relatively small state space, but unfortunately the number of basis functions for a given degree of accuracy is in general exponential in the dimensionality of the state space, making this approach intractable for games with more than three or four state space variables.

Since we are interested in two-player games, the state spaces needed will in general have twice the dimension of those required in single-character animation. This curse of dimensionality causes a serious difficulty for learning a suitable value function, since it will often be impractical to do so by the brute-force technique of simply using a larger set of basis functions.

Since we cannot learn an effective value function by using more basis functions, we take the approach of employing a smaller number of *better* basis functions. This idea has been noted by many prior authors in the context of artificial intelligence. If the basis functions can somehow be chosen so that they share many features in common with the true value function, then generally few of them will be needed to represent a value function to within a desired level of accuracy.

Unfortunately intelligently choosing a small set of basis functions for a Markov game or an MDP is a nontrivial problem. Often expert knowledge of the domain is employed, but automatic methods have been proposed, such as analyzing the topology of the state space [Sridhar and Maggioni 2007; Mahadevan 2006; Petrik 2007; Smart 2004] or trying to project the state space into a lower-dimensional space [Keller et al. 2006]. Although these existing techniques show some promise in many contexts, they are unsuitable for our approach. The state spaces encountered in the games involved in character animation often do not contain any particularly useful topology, and the state spaces cannot be projected into a small number of dimensions. We therefore propose a new technique for approximate value function learning and representation which automatically builds an intelligent and compact set of basis functions in a manner that scales gracefully with the dimension of the state space and is based directly on the definition of the game itself.

6.2 Learning in Higher Dimensions

In order to extend the basis function approximation algorithm to higher dimensions, we present a novel heuristic algorithm that exploits similarities between different parts of the state space to incrementally build a set of basis functions in an intelligent manner. Our method uses the game's reward and action functions directly to generate these basis functions, and unlike other methods, can be polynomial (rather than exponential) in the dimension of the state space.

The key observation behind our algorithm is that in many actual games, the value function will retain similar features across many of the dimensions of the state space. For example, in learning a value function for a game in which one player attempts to tackle the other, the value function will vary with the relative angle of the two players. However, we expect that it will probably not vary too wildly, and that at each angle it will be desirable, for instance, for the tackler to be both close to and facing their opponent.

6.2.1 *Simplified Example.* The regularity of the features which we expect to see in the true value function motivates us to detect and exploit these features using a sampling-based technique. Before we describe how this is done in full, it will be useful to provide a simplified pedagogical example. For the moment let us consider a standard (nongame-theoretic) example of solving for a controller which can balance a pendulum on its end by applying torques to the base of the pendulum. This problem can be formulated as an MDP with a two-dimensional state space and a reward function that penalizes pendulum positions deviating from the upright. As is the

case in the character animation games we use, the state space for the inverted pendulum can be written as the Cartesian product of a set of dimensions, in this case the pendulum's angular position and angular velocity.

$$\mathcal{S}_{pendulum} = \text{pos} \times \text{vel} \tag{4}$$

We begin by defining a set of basis functions over each of these dimensions, denoted by ϕ_{pos} and ϕ_{vel} respectively. These basis sets can be defined in any manner, but we use the bases of a third-degree B-spline. We also define an *outer product* operator on a pair of basis sets in the same manner as Treuille et al. [2007].

$$(\phi_\alpha \otimes \phi_\beta)(x) = \{b_1(x)b_2(x) | b_1 \in \phi_\alpha, b_2 \in \phi_\beta\}$$

This definition allows us to construct a set of basis functions in a high-dimensional space as the outer product of sets of basis functions defined in low-dimensional spaces. For instance, if $\phi_\alpha(x)$ is a set of basis functions of size $|\phi_\alpha|$ over X and $\phi_\beta(x)$ is a set of basis functions of size $|\phi_\beta|$ over Y, then $\phi_\alpha \otimes \phi_\beta$ is a set of $|\phi_\alpha| \cdot |\phi_\beta|$ basis functions defined over $X \times Y$. This allows us to define a set of basis functions over the entire state space for the inverted pendulum with $\phi_{pos} \otimes \phi_{vel}$. Since we are assuming for the sake of argument that it is intractable to learn a value function over $\phi_{pos} \otimes \phi_{vel}$ directly, as indeed it will be for higher-dimensional state spaces, we instead construct such a value function one dimension at a time using a sampling-based approach.

We begin by randomly selecting a set of values, $vel_1, \ldots, vel_m$, for the velocity of the pendulum. For each such vel_i we formulate a problem identical to the original MDP except that the velocity of the pendulum is restricted to always be equal to vel_i by simply setting the velocity to that value whenever we generate a new state. This corresponds to taking a one-dimensional "slice" of the original state space in a direction parallel to the position axis. The method given in Appendix A (or in Treuille et al. [2007]) is then used to learn a (one-dimensional) value function for each slice using the basis function in ϕ_{pos}, yielding a set of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m$, one per sampled velocity.

Although the value functions given for these slices by $\mathbf{v}_1, \ldots, \mathbf{v}_m$ will not generally be identical to the true value function along that slice, they will often have many features in common. Furthermore, since we expect the value function to have some regularity in its features, it will normally be possible to span $\mathbf{v}_1, \ldots, \mathbf{v}_m$ with a low-dimensional subspace. This means that we can represent all the features which are expected to be found along the position axis of the pendulum with a few basis vectors, which we determine using truncated PCA. Each of these vectors can be used in the same manner as the v_i vectors to define a single basis function over the position dimension of the problem. We denote the set of these basis functions with ϕ_{pos}^{PCA}.

Instead of learning a value function over $\phi_{pos} \otimes \phi_{vel}$, we can now instead learn a value function over $\phi_{pos}^{PCA} \otimes \phi_{vel}$. Since ϕ_{pos}^{PCA} will generally contain fewer basis functions than ϕ_{pos}, this problem will normally be computationally simpler. Furthermore, since we expect that ϕ_{pos}^{PCA} spans the set of all features found along the position dimension of the state space, the final value function will normally still be relatively accurate. A graphical illustration of this process is given in Figure 4.

6.2.2 *Nested-PCA Value Functions.* In the preceding example of learning a value function for an inverted pendulum, we described a procedure which worked on a two-dimensional state space by learning a series of one-dimensional value functions sampled from this state space, using PCA to find a reduced set of basis functions, and using an outer product operation to expand these basis functions

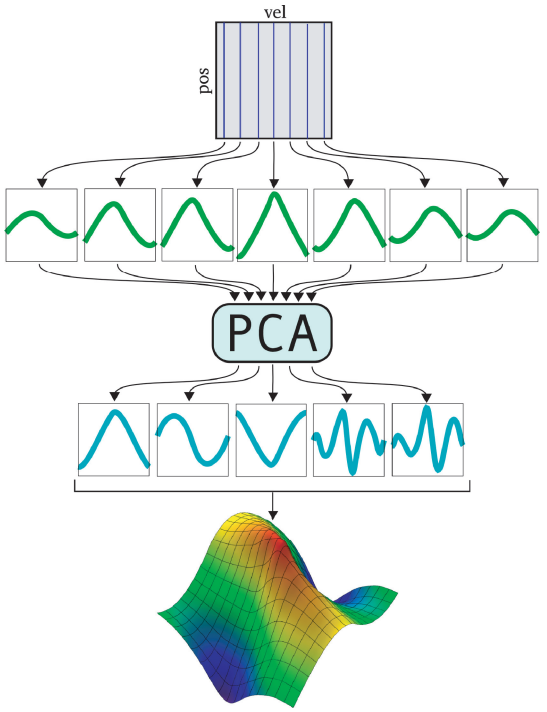

Fig. 4. A graphical illustration of the learning process for an inverted pendulum MDP using a nested-PCA representation. At top the entire state space is shown, with blue lines indicating a set of one-dimensional "slices" formed by fixing the pendulum's velocity to a particular value. The value function learned for each of these slices is shown in green. These slice value functions are then compressed into a reduced set of new basis functions using PCA. These basis functions are outer-producted with a set of B-spline basis functions over the velocity axis to form the set of basis functions used to learn the final two-dimensional value function, shown at bottom.

over the entire state space. The key insight which allows us to apply this procedure to higher-dimensional problems is that this process can be repeated recursively.

We start with a character animation problem formulated as a Markov game with an n-dimensional state space, represented as the Cartesian product of n variables.

$$\mathcal{S} = s_1 \times \cdots \times s_n \tag{5}$$

For each variable in the state space we also define a set of basis functions, $\phi_1(x), \ldots, \phi_n(x)$. Thus $\phi_1(x)$ is a set of basis functions defined over s_1, $\phi_2(x)$ is a set of basis functions defined over s_2, etc. As before, we begin by learning a set of one-dimensional value functions over s_1 by sampling $x_2 \in s_2, \ldots, x_n \in s_n$ and defining a set of games identical to the original one except that the allowed values in the state space along dimensions $2, \ldots, n$ are restricted to $x_2, \ldots, x_n$. Using truncated PCA on the set of vectors describing these value functions gives a new set of basis functions, ϕ_1^{PCA}.

We can now continue the process by randomly sampling $x_3, \ldots, x_n$, and for each sample learning a value function for the associated restricted game. These games are three-dimensional and we learn the value function using the set of basis functions $\phi_1^{PCA} \otimes \phi_2$. Each of these value functions is again represented with a single vector, so we can use PCA in the same manner as before to determine a reduced set of two-dimensional basis functions over $s_1 \times s_2$, denoted as ϕ_2^{PCA}. This process is continued, yielding ϕ_3^{PCA}, ϕ_4^{PCA},

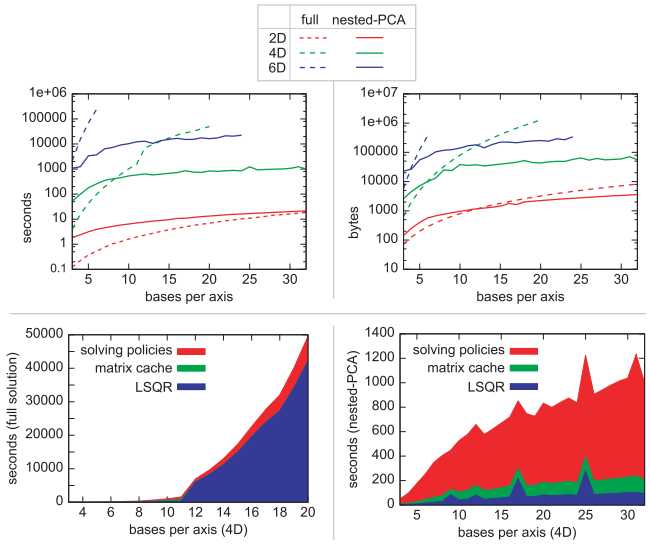

Fig. 5. From top left to bottom right: The precomputation time needed to learn a value function in the persuit/evasion games, the memory needed to store the value function, a breakdown of the learning time for learning a full value function, and a breakdown for learning using the nested-PCA procedure.

etc., until we arrive at the final set of basis functions spanning the entire n-dimensional space of our problem, $\phi_{n-1}^{PCA} \otimes \phi_n$.

There are a few properties about this method which we would like to note. Firstly, since we adaptively reduce the current set of basis functions with each added dimension, this technique produces a set of basis functions which can be orders of magnitude more compact than those required by the naïve approach as well as orders of magnitude faster to learn. This allows it to be applied to problems which would otherwise be intractable due to speed or memory requirements. At each iteration, however, the nested-PCA approach only has a limited view of a single "slice" of the full state space, so there is no strict guarantee that the final value function will be a good approximation. Nevertheless, we have observed this method to work well in practice, and we are unaware of any other approach which can learn suitable value functions for the problems required in our character animation scenarios.

In order to numerically evaluate the nested-PCA technique for value function learning we examine its impact on the time needed to precompute a value function, the memory needed to store this value function, and the impact on the accuracy of the final value function.

In order to illustrate the computational requirements of learning a full value function versus using the nested-PCA approach, we consider a simplified pursuit/evasion game where one particle attempts to "catch" another. The particles have both a position and a velocity and move at each step by applying an acceleration parallel to one of the principal axes. This game has the advantage that it can be defined for particles moving in a space of any dimensionality, allowing us to illustrate the impact that a problem's dimension has on the time and memory needed to learn and store a value function. We test this problem with state spaces of 2, 4, and 6 dimensions (half for position and half for velocity).

A plot of the time and memory needed to compute a value function for the pursuit/evasion game is shown at the top of Figure 5. The x-axis of these plots shows the number of B-spline basis functions used along each dimension in the state space (i.e., the size of each ϕ_i).

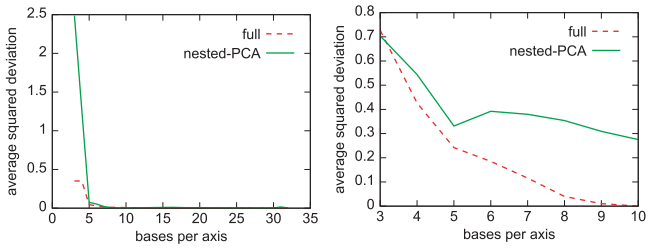

Fig. 6. The accuracy of the full versus nested-PCA methods for a two-dimensional inverted pendulum problem and a four-dimensional pursuit/evasion game. The accuracy is measured as the average squared deviation from the highest resolution value function learned with the full method.

The set of basis functions used by the full approach is taken to be the outer product of these basis sets, yielding $\prod_{i=1}^{n} |\phi_i| = \prod_{i=1}^{n} x = x^n$ total basis functions for an n-dimensional state space. Note the use of the log scale for the vertical axis of these plots.

For a small-dimensional game such as pursuit/evasion in a two-dimensional state space, this shows that the nested-PCA method is slower than the full approach. This is a result of the time involved in learning the value functions of the sampled "slices" of the state space which we then perform PCA on. In the higher-dimensional state spaces, however, the nested-PCA method is far more efficient, giving a 60-fold increase in speed and a 30-fold decrease in storage for the four-dimensional problem with 20 basis functions per axis. For the game with a six-dimensional state space we observe a 65-fold increase in speed and a five-fold decrease in storage with a mere six bases per axis. Furthermore, in practice the nested-PCA approach exhibits a superior asymptotic complexity as the number of bases per axis is increased, allowing us to learn value functions at a significantly higher resolution (i.e., bases per axis) in this example than is possible with the full approach.

In addition to considering the total precomputation time required it is useful to examine the breakdown of how this time is spent. This is shown in the bottom of Figure 5 for the four-dimensional pursuit/evasion game. For the full-value-function approach, we see a large spike in the running time where the time required to solve the least squares system with LSQR begins to dominate (see Appendix A). This jump is due to the size of the problem increasing beyond the point where we are able to cache the matrix used in this computation. Since the nested-PCA method is far more compact in its representation this is not an issue, and most of the time is spent solving for the policies of the players using the technique in Section 5.3.

When using the nested-PCA method for learning a value function there is potential for some loss of accuracy. We examine the magnitude of this loss in learning value functions for both a single-agent MDP and a two-player Markov game. For the MDP we choose the two-dimensional inverted pendulum problem used as an example in Section 6.2.1 and for the Markov game we use the aforementioned pursuit/evasion game with a four-dimensional state space. A plot of the accuracy for these two problems is shown in Figure 6. For the inverted pendulum problem both methods quickly converge to the correct value function. In the four-dimensional game, however, nested-PCA levels off in accuracy due to the errors inherent in the dimension-by-dimension reduction. Despite these remaining inaccuracies, we find the value functions generated with this technique still perform relatively well in practice as they generally capture the most important features of the value function, providing useful controllers in games which are intractable with the full approach.

7. BIASING FOR SUBOPTIMAL OPPONENTS

The framework described thus far is based on the assumption that the opponent is acting optimally. This is often not the case, and may pose an undesirable restriction for use in computer games or other applications. Human players, particularly novices, do not play rationally and instead exhibit certain biases. The optimal play against such a biased opponent will in general differ from the optimal play against a fully rational opponent. We show a method for incorporating such suboptimalities into our framework. This not only allows a richer model of the opponent, but allows turn-based and stochastic turn-based controllers as special cases.

We use this ability to model a suboptimal opponent primarily to compare against turn-based methods, but it also has the potential to learn an opponent model from traces of user game play. Such a methods are known as *opponent modeling* and a form of this has been applied to Markov games before [Uther and Veloso 1997]. Our method differs in that it is somewhat more general and is not integrated as deeply into the value function learning method. We feel this is an advantage in the higher-dimensional and continuous spaces we encounter as it allows a greater variety of techniques and heuristics for learning a bias model from data. We do not address the problem of learning a user's biases from game play data here, and only discuss how to incorporate a prespecified model of such biases into our controller framework.

We represent an opponent's bias via a function $\mathbf{M}_b(x)$ which yields a left stochastic square matrix at each point in state space. That is, each column in this matrix should be a valid probability vector. The number of rows and columns in $\mathbf{M}_b(x)$ should also each be equal to the number of actions in $\mathcal{O}(x)$.

The interpretation of this bias matrix is that if π is the rational policy at some point x in the state space, then $\mathbf{M}_b(x)\pi$ gives the suboptimal policy which the opponent is assumed to choose at x. We can then solve for the agent's optimal policy at x with the opponent's suboptimality taken into account with a simple modification to Eq. (2), obtained by replacing $\mathbf{R}(x)$ with $\tilde{\mathbf{R}}(x)$ where

$$\tilde{\mathbf{R}}(x) = \mathbf{M}_b(x)\mathbf{R}(x). \tag{6}$$

Although there are many possible ways to define $\mathbf{M}_b$, including learning a model from examples of user play, one particularly simple and useful one is to model $\mathbf{M}_b$ as a matrix of the form

$$\mathbf{M}_b(x) = \rho(x)\mathbf{I} + (1 - \rho(x))\,[\zeta_s(x)\ \cdots\ \zeta_s(x)],$$

where $\zeta_s(x)$ is a function which yields a vector of probabilities over $\mathcal{O}(x)$ and $\rho(x)$ is a constant between zero and one determining the "predictability" of the opponent. If we choose $\rho(x) = 1$ then we arrive at a Markov game optimal controller. On the other hand if we choose $\rho(x) = 0$ we can reproduce a turn-based controller by letting $\zeta_s(x)$ be zero except at the action which would be chosen by the first player. Or, for instance, we can model an opponent which only turns to the left by setting $\zeta_s(x)$ to only be nonzero on the indices corresponding to left-turning actions. If we wish the opponent to represent a turn-based player with some randomness added, this can also be easily incorporated by appropriately defining $\zeta_s(x)$ to be nonzero on the indices corresponding to the actions to be selected between. We should note that if $\rho(x) = 0$ the agent's policy is always deterministic, and so the Markov game aspects of the controller disappear and the resulting animations are often unconvincing.

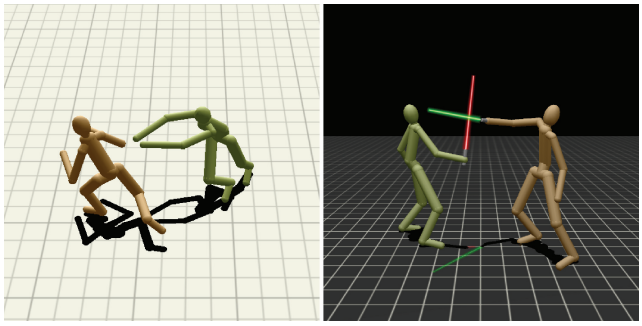

Fig. 7. Screen shots of our tag and sword controllers in action.

8. RESULTS

We have tested our framework on several simple games and on character controllers for tag and sword fighting. We have found that these techniques are sufficient both to learn value functions for these games and to animate the characters in real time. In these games we have observed intelligent nondeterministic behaviors and the use of moves such as feints and quick-steps which decrease a player's predictability.

It is unfortunately impossible to perform a direct numeric comparison between our method and previous approaches, as they are based on different models of character interaction (i.e., simultaneous versus turn-based). We instead provide this evaluation qualitatively. We have found in both games that our technique added a significant amount of randomness to the observed policies while retaining intelligent-looking behavior. We illustrate this randomness in the tag game where it is easy to visualize (Figure 8). We have also tested the tag and sword games with a turn-based controller and found that the resulting controller appears very unintelligent since the actual mechanics of the game allow for concurrent actions by the two players. We have also tested a randomized variation of a turn-based controller which selects randomly between the three actions with the highest value. This randomized controller exhibits nondeterminism, but is not intelligent in its usage, and in practice often appears less believable than the deterministic controller due to its selection of visually suboptimal actions. In addition we have tested both games using only the reward function (instead of the value function) and find that the results appear significantly less believable. Readers are encouraged to view the video accompanying this article, that can be accessed through the ACM Digital Library, to make these qualitative judgments for themselves.

8.1 Tag

We have designed a Markov game controller for a game in which one player attempts to tag another, shown in Figure 7. The characters have a variety of motions available, including walks, runs, quick-steps, tackles, and dodges, all of which are parametrized by angle. Each motion consists of a single step or a portion of a step, and we blend each motion into the next using a combination of interpolation and a postprocessing phase to smooth out changes in momentum. This allows our model to be as reactive as possible while still producing visually pleasing results.

We construct the reward function for this game as a sum of four components, requiring approximately 80 lines of C++ to define:

(1) the agent should desire to be near the opponent;
(2) if the agent is tackling they want their hands near the opponent;

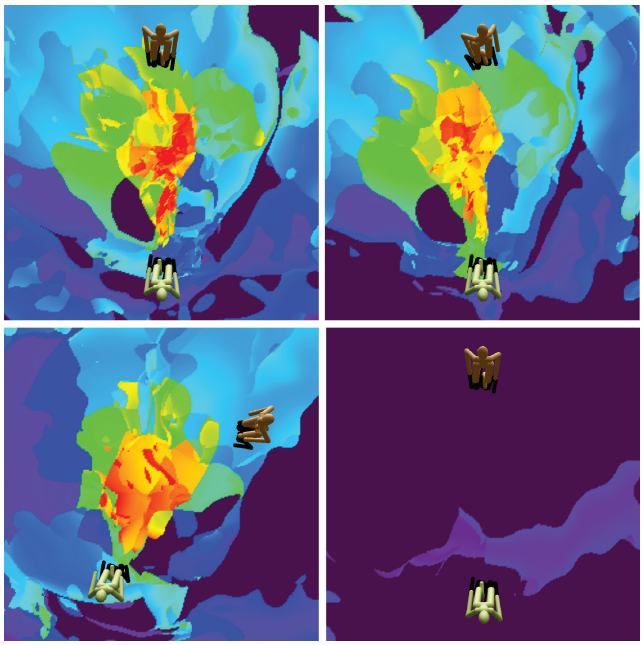

Fig. 8. Colorplots showing states with low versus high amounts of nondeterminism, shown from purple to red, respectively, as functions of the agent's position. The color is calculated as a function of the average weighted deviation of the two-dimensional positions of the agent's optimal policy at each possible two-dimensional position of the agent. The weights in this average are the probabilities with which each action was taken. The lower right shows the characters in walking states while the others show running states at different agent orientations. Our method automatically determines rich nondeterminism structures and that when walking, the players are moving too slowly for nondeterminism to be of use.

(3) a tackle is less effective if the opponent is dodging;

(4) dodges and tackles are not useful if the characters are far away.

The state space for this game consists of the relative translations and rotation of the two characters and their respective motion nodes, giving a five-dimensional space. The value function for this method is constructed from the outer product of third-degree B-spline bases along the x, z, and θ dimensions and discrete bases along the dimensions representing the indices of the PMG nodes of the characters. We use a discount factor of $\gamma = 0.8$. The problem is large enough that even after significant effort we were unable to learn a value function using standard techniques. Using our nested-PCA method, however, we were able to generate effective value functions reasonably efficiently. The total numbers of underlying basis functions along each axis in our final value function is 32, 32, 16, 18, and 20 for the dimensions of x, z, *theta*, *agent_node*, and *opponent_node*, respectively, and were learned using the dimension ordering (x, z), θ, (*agent_node*, *opponent_node*). Learning this value function required approximately eight hours using 64 parallel processes on a cluster of 30 machines (dual quad-core, 2.27 to 2.66 GHz). Despite the fact that the full outer product of the axis bases would yield 5898240 basis functions, the largest number of basis functions ever used in the PCA-slice solution process was only around 10000. Despite never solving a single large-scale value function problem, the result exhibits a high degree of detail and creates convincing animations, showing such behaviors as leading an opponent in one direction and then quickly changing directions to avoid the tag.

We observe a variety of behaviors in the controller generated by this method, including nondeterminism when appropriate in a real game (i.e., when the players are close), as shown in Figure 8. The motions of the two characters are also visibly synchronized, leading to realistic interactions. We also observe emergent behaviors that have analogs in human motion. In particular the dodging character often performs a series of very quick footsteps when near the tackler. This serves to increase unpredictability and the ability of the character to quickly dodge left or right, allowing it to take advantage of nondeterminism more effectively.

Using our opponent-biasing framework we have tested this approach against opponents which act with greedy or randomized-greedy algorithms. In this case the game-theoretic behavior automatically reduces to minimax behavior, with the opponent minimizing the agent's reward and the agent maximizing given the opponent's action. The pure-greedy opponent model gives purely deterministic motions, which lack the variety and depth of those exhibited by our Markov game controller. The randomized-greedy approach behaves even worse in many respects, and often chooses obviously incorrect motions rather than exhibiting intelligent randomness.

8.2 Sword Fighting

We have also designed a Markov controller for a sword fighting game. In this game, shown in Figure 7, each character attempts to swing a sword and hit the other character while trying to prevent the adversary from doing the same. The model has five attacks: an overhead attack, a left-side attack, an uppercut from the left side, a right-side attack, and an uppercut from the right side. Each attack has a corresponding block that the adversary can execute to prevent the attack from succeeding. If the block is not executed in time, the attacking character can follow through to strike the adversary, immediately ending the game. Both characters are able to attack and block, but only one character is allowed to attack at a time. This is analogous to systems of "right of way" in the scoring the sport of fencing. In the event that both characters attempt to begin attacks at the same time, the agent gets priority. Each character is also able to execute a few feinting attacks that begin along a certain line of attack and can either continue on that line or transition to another line.

In this game, the value function is particularly critical because the characters can only visualize outcomes that could occur within a quarter of a second. Without advance preparation, this is often not enough time for a character to defend against an impending attack that will strike within this time threshold. Our value function, through its consideration of future states, is able to synchronize the actions of the two characters such that both are able to time their attacks and blocks correctly. We choose a discount factor of $\gamma = 0.5$ and construct the value function using a B-spline basis over the time within each character's node and discrete bases over the IDs of the nodes, resulting in $16 \times 85 \times 85 = 115600$ basis functions respectively. We were able to solve for a value function in this case directly without using the nested-PCA method, although it is near the limit of what we can solve using the simple approach. The precomputation time for this value function is approximately six hours using 64 parallel processes on the same cluster of 30 machines used for the tag game (dual quad-core, 2.27 to 2.66 GHz).

The resulting animations contain highly coupled intricate interactions between the players as well as a great deal of nondeterminism.

We also see the use of feints in these animations, where a character begins one attack and then switches to another. It is important to note that all such feints were synthesized from appropriate graph transitions; feints were not themselves nodes in the graph. This serves the same game-theoretic purpose as the quick-steps seen in the tackle controller and helps to show that these natural behaviors can occur under our approach.

We have also used our biasing approach to test the sword game with a greedily optimizing opponent, automatically resulting in a turn-based controller. This method works very poorly on the sword game, significantly worse than we expected. The problem is that since the agent knows the opponent's policy, it can take immediate advantage of it to avoid being blocked. The resulting animations are frequently a move or two in length and are highly unconvincing.

9. APPLICABILITY TO OTHER GAMES

Since our technique cannot be applied to all possible games, we feel it would be useful to describe what criteria make a game well suited to our technique and which games our technique is not as well suited for.

We have found that our method works well in situations where two characters are interacting very closely. This is because the game must have the property that for each move one player can make, the other has a countermove in order to exhibit game-theoretic nondeterminism. This is best illustrated in the sword fighting example, and in the tag example when the characters are close to each other. For instances in which the characters are far away from each other, or where there is a single obvious move to make, the optimal policy will be deterministic. Although our approach can automatically identify and handle such situations, if this is known to be the case a priori then the game theory in Section 5 is not strictly necessary. Nevertheless, situations where the characters interact closely in a complicated manner are some of the most interesting and exciting ones that arise in actual games, and our approach provides a valuable tool for automatically analyzing them.

In order to better illustrate this, we provide a few examples of games for which our technique is not the best choice because of the lack of generated nondeterminism. The first game we tested for this was a variation on the tag game with the lookahead time determining the player's actions set to 0.1s instead of the normal 0.25s. In this game the players can react quickly enough so that feints and misdirections are not particularly useful and the resulting behaviors are entirely deterministic. The second game illustrating this involves a "giant" character chasing a much smaller player. Since the giant's motion is slow compared to that of the smaller player, the giant is unable to misdirect the other player and this game is often fully deterministic. When it does exhibit nondeterminism it is also not in any particularly meaningful way. Plots illustrating the nondeterminism in both of these cases are show in Figure 9. Although our approach can still generate controllers for these methods, the results are similar to those which would be produced by a turn-based approach.

There are also some sorts of games for which our technique cannot be applied at all. First, our approach is only suitable for two-player games. Although it is possible to formulate games with more than two players as game theory problems, solving for policies is more involved and probably not yet well suited for use in real-time character control. Furthermore, Markov games are inherently games of perfect information, so they are not capable of modeling cases where it is important that one player knows something the other does not. In real life this is common (since we cannot see what is behind us), but in video games the assumption of perfect information often either holds directly or is a reasonable approximation.

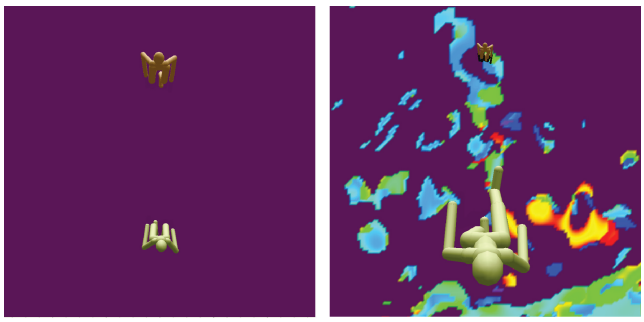

Fig. 9. Plots of the level of nondeterminism for two games to which our method degenerates to a turn-based approach. On the left a game of tag where the players can replan very rapidly. On the right a game where a slow-moving giant chases a smaller player.

A somewhat more subtle point is that we have only formulated our approach for *zero-sum* two-player games. This means that the goals of the players must be exactly opposite. Although this zero-sum assumption is a good model of many of the character animation problems that arise in competitive video games, it does make it difficult to model instincts such as good sportsmanship and self-preservation which require some goal beyond simply defeating the opponent. In addition, designing the reward function for a single-character controller can sometimes require tuning, and this is doubly true in the case of Markov games since the policies of both players are determined by the same reward function.

Finally, although the nested-PCA method for value function learning allows us to increase the size of the state space that can be used to define a game, it is still not possible to learn a value function for games with very large state spaces. This makes our approach best suited to short- or medium-term interactions between characters. Intuitively, our approach allows us to take situations of a comparable complexity to those previously used in single-character MDP-based controllers and scale them up to two players. This means that we cannot handle, for instance, two characters competing in a large and complicated arena with many moving obstacles.

10. CONCLUSION AND FUTURE WORK

We have presented a method for controlling characters interacting within an adversarial context by applying principles from game theory. The generated controllers display intelligent planning and a nuanced use of nondeterminism, and can be adapted to take advantage of an opponent's specific idiosyncrasies. This leads to rich and believable behaviors for virtual characters in competitive situations.

We use a precomputation phase to learn a value function which allows a runtime controller to operate. In order to scale our method to be practical in actual games we have developed a new method for value function representation and learning. We have shown that in the case of two-player adversarial games a Markov game formulation is both a useful mathematical technique and can be made to operate efficiently enough and solve problems of large enough scale to be used in practice. Characters controlled in this manner show convincing, varied motions and behave intelligently, using feints and other techniques to attempt to best the opponent.

In our experience the two biggest difficulties in applying these methods are in areas that are arguably orthogonal to the contributions of this article: designing the reward function and building the

motion model. Since the players base their policies on the user-defined reward function, any unrealistic features in this function lead to unrealistic behaviors. There is no remedy for this other than careful thought and perhaps some trial and error, but this effort must be spent in designing the scoring function for any computer game, so in many instances this does not require much additional effort.

In designing a motion model to be used for an adversarial controller we have found that the continuity and the reactivity of the model are difficult to balance correctly. That is, if we attempt to achieve smoother motions we often reduce how quickly the model allows characters to react and move in the process. Similarly, improving the rate at which characters can respond causes the motions to appear less smooth and realistic. This problem is not unique to our method and is present in all real-time controllable characters. Nevertheless we see it as one of the most fruitful avenues for future research in this area.

Another useful area for future work is in extending these methods to include nonzero-sum games. This would allow the agent's goals to differ from the opponents, and would allow semicooperative and other complex behaviors. Extending our approach to handle more than two characters, even whole teams, is another path that remains to be tackled.

The grand vision which our approach is a step towards is to be able to generate real-time controllers for almost any game in which characters behave in intelligent and convincing ways, including learning from past user behavior. For instance, after a user plays a game for a short while we would like to automatically refine our opponent-model for them and adapt our strategies in the same way in which a human opponent might. We feel that the game-theoretic approach provides a new, powerful, and rich method for addressing problems related to controlling characters in games.

APPENDIX

A. LEAST-SQUARES VALUE FUNCTION LEARNING

This description of the value function learning algorithm we employ as a component of our PCA-slice method is relatively brief. Those desiring a more detailed explanation should consult [Treuille et al. 2007] for an intuitive background on the idea or [Lagoudakis and Parr 2002] for a more detailed description of the algorithm.

Given a game with state space $\mathcal{S}$ we learn a value function using a least-squares policy iteration approach. First, we are given or (randomly) generate a set of sample points $x_1, \ldots, x_n \in \mathcal{S}$ and a set of basis functions $\phi_1, \ldots, \phi_m$ over $\mathcal{S}$. We also implicitly construct a basis matrix Φ such that $\Phi_{i,j} = \phi_j(x_i)$. We represent a value function as a linear combination of the basis functions, and if w is a vector giving the weights in this combination then the value function at each sample point can be calculated by Φw.

We compute a set of blend weights, w, for our basis function by initializing $w = 0$ and alternating two steps until convergence or a maximum number of iterations is reached:

(1) Solve for the optimal policies for the agent and opponent, $\pi_{a,i}$, $\pi_{o,i}$, at each x_i given w using equations 2 and 3.
(2) Recompute w holding each $\pi_{a,i}$, $\pi_{o,i}$ fixed.

In step 2 we recompute the weight vector by attempting to minimize the squared sum of the difference between $V(x_i)$ and $\pi_{o,i}^T \overline{\mathbf{R}(x_i)} \pi_{a,i}$ over all the x_i samples. These differences are known as *Bellman errors* [Williams et al. 1993], and represent the difference between the value functions's value at x_i, and the value expected by performing one-step lookahead with the optimal policies at x_i. If these errors are zero at all points in the state space, then $\pi_{a,i}$ and $\pi_{o,i}$ will be the correct optimal policies for each player and V will be the true value function. Since we are learning an approximate value function, the condition that the Bellman errors be everywhere zero is approximated by minimizing the squares of the Bellman errors over the x_i points. This minimization can be achieved by computing a least squares solution to the system:

$$r = (\Phi - \gamma \mathbf{P}\Phi)\, w$$

Where r is a vector such that r_i gives the expected value of the game's reward function at x_i under the policy π_i, γ is the game's discount factor and $\mathbf{P}$ is a matrix such that:

$$\mathbf{P}_{i,j} = \sum_{y \in \mathcal{S}, a, o} \pi_{a,i}\, \pi_{o,i}\, \mathcal{P}(x_i, a, o, y)\, \phi_j(y)$$

We solve this least squares system using the LSQR algorithm [Paige and Saunders 1982]. The policy iteration procedure terminates when the residual error of this system converges or increases from the previous iteration's value. We note that this algorithm is *not* guaranteed to converge for values of γ close to 1, although this can be remedied by replacing the least-squares system with a linear program as in [Treuille et al. 2007]. We choose the method given here because we have found it to be significantly more computationally efficient than that in [Treuille et al. 2007], and have not found the lack of guaranteed convergence to be a problem for any of the Markov games and MDPs we have tested.

Another advantage of the approach given here over that in [Treuille et al. 2007] is that it is simple to parallelize over a cluster of computers. To achieve this parallelization on a cluster with k cores, we split the $x_1, \ldots, x_n$ samples into k sets of roughly equal size. Each core is then assigned one of these k sets. To parallelize step 1, each core simply computes and locally stores the agent and opponent policies at its assigned x_i samples. Understanding the parallelization of step 2 is only slightly more involved, and involves noting that running LSQR on our least squares system only requires the computation of the matrix-vector products $(\Phi - \gamma \mathbf{P}\Phi)\, x$ and $(\Phi - \gamma \mathbf{P}\Phi)^T\, y$. Since each processor can compute the subset of the rows of $\Phi - \gamma \mathbf{P}\Phi$ corresponding to its assigned sample points, these products can be parallelized by having each processor compute either a sub-sequence of a sub-sum of the result, for the two cases respectively.

REFERENCES

GRAEPEL, T., HERBRICH, R., AND GOLD, J. 2004. Learning to fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*.

HECK, R. AND GLEICHER, M. 2007. Parametric motion graphs. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*.

IKEMOTO, L., ARIKAN, O., AND FORSYTH, D. 2005. Learning to move autonomously in a hostile environment. Tech. rep. UCB/CSD-5-1395, University of California at Berkeley.

KELLER, P. W., MANNOR, S., AND PRECUP, D. 2006. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*. ACM, New York, 449–456.

LAGOUDAKIS, M. AND PARR, R. 2002. Value function approximation in zero-sum markov games. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI'02)*.

LAU, M. AND KUFFNER, J. J. 2006. Precomputed search trees: Planning for interactive goal-driven animation. In *Proceedings of the ACM*

SIGGRAPH/Eurographics Symposium on Computer Animation. 299–308.

LEE, J. AND LEE, K. H. 2006. Precomputing avatar behavior from human motion data. *Graph. Models 68*, 2, 158–174.

LIU, C. K., HERTZMANN, A., AND POPOVIĆ, Z. 2006. Composition of complex optimal multi-character motions. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'06)*. 215–222.

LO, W.-Y. AND ZWICKER, M. 2008. Real-Time planning for parameterized human motion. In *Proceedings of the ACM SIGGRAPH/ Eurographics Symposium on Computer Animation (SCA'08)*.

MAHADEVAN, S. 2006. Learning representation and control in continuous markov decision processes. In *Proceedings of the 21st National Conference on Artificial Intelligence*. AAAI Press.

MCCANN, J. AND POLLARD, N. 2007. Responsive characters from motion fragments. *ACM Trans. Graph. 26*, 3.

MORRIS, P. 1994. *Introduction to Game Theory*. Springer.

PAIGE, C. C. AND SAUNDERS, M. A. 1982. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw. 8*, 1, 43–71.

PETRIK, M. 2007. An analysis of laplacian methods for value function approximation in mdps. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 2574–2579.

PETROSJAN, L. AND ZENKEVICH, N. 1996. *Game Theory*. World Scientific.

SADOVSKII, A. L. 1978. A monotone iterative algorithm for solving matrix games. *Soviet Math Rep. 238*, 3, 538–540.

SHIN, H. J. AND OH, H. S. 2006. Fat graphs: Constructing an interactive character with continuous controls. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'06)*. Eurographics Association, 291–298.

SHUM, H. P. H., KOMURA, T., SHIRAISHI, M., AND YAMAZAKI, S. 2008. Interaction patches for multi-character animation. *ACM Trans. Graph. 27*, 5, 1–8.

SHUM, H. P. H., KOMURA, T., AND YAMAZAKI, S. 2007. Simulating competitive interactions using singly captured motions. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'07)*. ACM, New York, 65–72.

SMART, W. D. 2004. Explicit manifold representations for value-function approximation in reinforcement learning. In *Proceedings of the 8th International Symposium on Artificial Intelligence and Mathematics*. 25–2004.

SRIDHAR, M. AND MAGGIONI, M. 2007. Proto-Value functions: A laplacian framework for learning representation and control in markov decision processes. *J. Mach. Learn. Res. 8*, 2169–2231.

SUTTON, R. S. AND BARTO, A. G. 1998. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

TREUILLE, A., LEE, Y., AND POPOVIĆ, Z. 2007. Near-optimal character animation with continuous control. *ACM Trans. Graph. 26*, 3, 7.

UTHER, W. AND VELOSO, M. 1997. Adversarial reinforcement learning. Tech. rep. In *Proceedings of the AAAI Fall Symposium on Model Directed Autonomous Systems*.

WILLIAMS, R. AND BAIRD, L. C. III. 1993. Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep. NU-CCS-93-14. Department of Computer Science, Northeastern University. November.

Received February 2010; accepted April 2010