Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2008)
M. Gross and D. James (Editors)

Real-Time Planning for Parameterized Human Motion

Wan-Yen Lo and Matthias Zwicker

University of California, San Diego

Abstract

We present a novel approach to learn motion controllers for real-time character animation based on motion capture data. We employ a tree-based regression algorithm for reinforcement learning, which enables us to generate motions that require planning. This approach is more flexible and more robust than previous strategies. We also extend the learning framework to include parameterized motions and interpolation. This enables us to control the character more precisely with a small amount of motion data. Finally, we present results of our algorithm for three different types of controllers.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Animation

1. Introduction

An important component of applications in virtual environments and computer games is to generate human motion in real-time and with interactive user control. The most common approach is based on large databases of motion capture data. Motion capture data provides a high degree of realism, but it needs to be processed and organized into suitable data structures to allow for interactive control and the synthesis of new motion. Typically the data is split into short motion clips. The clips are organized in a graph structure, and new motion is generated by traversing the graph.

For interactive applications, the main challenge of these approaches is to determine an appropriate graph traversal, i.e., a sequence of motion clips, in real time and under continuous user control. In computer games, this problem is usually solved by manually constructing elaborate state machines that determine the next best clip based on the current environment and user input. This approach involves a large amount of manual work to carefully craft the state machines and the rules specifying all potential state transitions. This becomes particularly cumbersome in modern games that use thousands of motion clips. In addition, it is hard to incorporate behaviors that require planning for a distant goal.

Reinforcement learning is a promising approach to address these issues. Using reinforcement learning it is possible to formulate higher level goals, such as obstacle avoidance or grasping objects at specific locations, and generate control policies to achieve these goals. One does not need to manually construct transitions to indicate how to fulfill the tasks in different situations. Instead, the controller is constructed in a pre-process by exploring and learning from all possible situations. At run-time, the controller can make near-optimal decisions automatically and instantaneously, reacting to user input or changes in the environment.

In this paper we describe a reinforcement learning technique for interactive control of human characters. Our approach can generate motions that require planning, and it allows for precise control using parametric blending of several motions. Our algorithm includes two main contributions:

- We use a tree-based fitted iteration algorithm to learn control policies. This approach is more flexible and more robust than previous methods to construct motion controllers using reinforcement learning.
- We extend the reinforcement learning framework to include parameterized motions and interpolation. This allows us to control characters more precisely without requiring an excessive amount of input data.

The rest of this paper is organized as follows: We describe previous work in Section 2. We present the reinforcement learning framework and our tree-based fitted iteration algorithm in Section 3. We describe how to include parameterized motions in Section 4. Finally, we present results of several motion controllers in Section 5 and conclusions and future work in Section 6.

2. Background

Motion capture data is extensively used in computer animation, because it is able to capture all the subtleties of real human motion. By piecing together many short motion clips, we can further create novel but realistic motions. Consequently, segmenting the acquired motion and rearranging the clips to achieve specific goals is an important research topic. A number of algorithms have been developed to represent plausible transitions between motion clips with graph structures [KGP02, AF02, LCL06]. With these techniques, novel motions can be generated simply by building walks on the graph. For off-line applications, where the full motion specification is known in advance, a global sub-optimal or close-to-optimal solution that minimizes an objective function, such as a certain energy, can be found [KGP02, AFO03, SKG05, LK05, SH07]. In interactive applications, new input is continuously arriving and the decision for selecting the next clip needs to be made in a very short amount of time. Therefore, only local search can be performed to generate motions in response to user-input [PSKS04, LK05, KS05, SO06, HG07].

The challenge for local search methods is to synthesize motions that require planning. Motion planning is important to achieve realistic results in many scenarios. For example, one may need to prepare well in advance to grasp an object at a particular location. Hence, instead of trying to search a point-to-point path on the graph, we use reinforcement learning techniques [KLM96, SB98] to train a motion controller off-line, which can make on-line decision quickly in any given situation. Several methods have been proposed in computer animation to utilize reinforcement learning to obtain policies for choosing actions that will increase long term expected rewards [LL04, IAF05, LL06, TLP07, MP07]. Lee and Lee [LL04, LL06] discretize the state space and use dynamic programming to construct a sample-based value function for boxing. Ikemoto et al. [IAF05] exploit a more complete state space in order to aid a global planner. More recently, McCann and Pollard [MP07] integrated a model of player behavior into an existing reinforcement learning method, enabling highly responsive character animation in real-time. Instead of discretizing the state space, Treuille et al. [TLP07] approximate the value function over a continuous state space, enabling the construction of low-dimensional, near-optimal motion controllers. We adopt tree-based regression algorithms [EGW05] to better approximate the value function. Compared to previous work, our approach converges faster without any assumption about the shape of the value function. We can support value functions with any possible shapes. Moreover, the above methods suffer from the limitation that the space of available motions is discrete. This makes it harder to achieve precise control such as walking in an exact direction or stepping on an exact point.

Parametric synthesis allows interpolating motions from a parametric space, and thus it can provide fine control. The parametric space is an abstract space defined by kinematic or physical attributes of motions. By parameterizing all motion samples in the space, and by blending among multiple motions, motion interpolation can create novel motions that have specific kinematic or physical attributes [WH97, RCB98]. Kovar and Gleicher [KG03, KG04] proposed an automated method for identifying and registering logically similar motions. They also build a continuous parameterized motion space for similar motions that provide efficient control for interpolation. Mukai and Kuriyama [MK05] improve motion interpolation with the use of geostatistics, treating interpolation as statistical prediction of missing data in the parametric space. Safonova and Hodgins [SH05] analyze interpolated human motions for physical correctness and show that the interpolated results are close to the physically-correct motions. Cooper et al. proposed active learning to adaptively sample the parametric space so that the space can be well sampled with a reduced number of clips [CHP07]. Recently, researchers have also combined motion graphs with parametric synthesis to form richer, more complete motion spaces [SH07, HG07]. In order to provide accurate control, we present a way to learn parametric motion controllers, which can compute near-optimal parameters for motion synthesis in real-time.

3. Learning the Motion Controllers

In this section we describe a reinforcement learning framework to obtain motion controllers for interactive character animation. Using a database of atomic motion clips, our goal is to generate natural character motion as a sequence of clips. At each time step, the motion controller decides which motion clip best follows the user input and respects constraints imposed by the environment. This decision must be made quickly, since time lags are not allowed in interactive environments. The controller should also be able to achieve user objectives that require planning ahead of time. In addition, both user input and the environment should be represented using continuous parameters to allow for precise control. We describe in Section 3.1 how this problem can be framed in the context of reinforcement learning. We discuss different strategies to solve the learning problem in Section 3.2, and we describe our tree-based approach for learning motion controllers in Section 3.3.

3.1. Problem Formulation and the Optimal Policy

We consider the motion control problem as a time-invariant stochastic system with discrete-time dynamics,

$$s_{t+1} = f(s_t, a_{t+1}), \tag{1}$$

where for all discrete time steps t, s_t is an element of *state space* $\mathcal{S}$, a_{t+1} is an element of *action space* $\mathcal{A}$, and f is the *transition function*. In our context, action space is the set of motion clips in the database. State space $\mathcal{S}$ is composed of all possible configurations of the current character motion, the environment, and user input. The parameterization of state space is application specific, and we give concrete examples in Section 5. In general, an element s of state space is a vector $(a, x_1, \ldots, x_n)$, where $a \in \mathcal{A}$ is the clip currently played, and $x_1, \ldots, x_n \in \mathbf{R}^n$ are parameters describing the character's current situation in the environment, such as the relative position to the goal, or deviation from desired orientation.

The transition function f describes how the current state is updated when a certain action is executed. At time t, when the action a_{t+1} for the next time step is picked, the corresponding clip is selected from the database and concatenated with the current motion. Then the state s_{t+1} is updated with the character's new situation in the environment. For any state s and next action a' the transition function can be expanded as

$$f(s,a') = s' = (a', x_1 + dx'_1, \ldots, x_n + dx'_n), \tag{2}$$

where dx'_i represents the amount of change in x_i with the use of clip a'. Take the navigation controller for example: the goal of this controller is to let the user control the walking direction of the character, so x_1 represents the deviation from the desired walking direction, and dx_1 of each clip represents the character's change of orientation during that clip.

A basic component of reinforcement learning is the *instantaneous reward* $r_t = R(s_t, a_{t+1})$, which is a scalar valued function defined for each current state and next action. In our context, the reward is composed of a *state reward* R_s and a *transition reward* R_t,

$$R(s,a') = R_s(s) + R_t(a,a'). \tag{3}$$

The state reward measures how well the character respects user objectives and environmental constraints, while the transition reward is used to ensure smooth motion transition between different clips. Note that the state reward is application specific and usually designed manually. We provide the details for several motion controllers in Section 5. As for the transition reward, we compute the weighted sum of squared differences of the positions and orientations across all joints in the blended region of motion clip a and a', and reward them if the difference is small.

The main idea of reinforcement learning is to determine an *optimal policy* that, at every time step t and state s_t, selects a next action a_{t+1} so that the *long term reward* is maximized. The long term reward is defined as $\sum_{t=0}^{\infty} \alpha^t r_t$, where $\alpha \in [0,1)$ is the *discount factor*. The discount factor accounts for future uncertainty and gives more weight to the near than the distant future. Maximizing the long term reward enables planning. The optimal policy may not always choose the action with the highest instantaneous reward, but select an action that results in a higher reward over time.

Suppose the decision is made by some policy Π, that is $\Pi(s_t) = a_{t+1}$. The *value function* V^Π represents the expected long-term reward from an initial state s following this policy. It is defined by the *Bellman equation*,

$$\begin{aligned} V^\Pi(s) &= E^\Pi\left[\sum_{t=0}^{\infty} \alpha^t r_t \middle| s_0 = s\right] \\ &= R(s,\Pi(s)) + \alpha V^\Pi(s'), \end{aligned} \tag{4}$$

where $s' = f(s,\Pi(s))$. One can prove [SB98] that there is always at least one optimal policy $\Pi^\star$, for which $V^\star(s) \geq V^\Pi(s)$ for all states s and all policies Π, where the value function $V^\star$ measures the long-term state reward under the optimal policy. Note that, if we have the value function $V^\star$, we can construct the optimal policy as follows:

$$\begin{aligned} \Pi^\star(s) &= \operatorname*{argmax}_{a' \in \mathcal{A}} \left[E^\star\left[\sum_{t=0}^{\infty} \alpha^t r_t \middle| s_0 = s, a_1 = a'\right]\right] \\ &= \operatorname*{argmax}_{a' \in \mathcal{A}} \left[R(s,a') + \alpha V^\star(s')\right] \\ &= \operatorname*{argmax}_{a' \in \mathcal{A}} \left[R_t(a,a') + \alpha V^\star(s')\right], \end{aligned} \tag{5}$$

where $s' = f(s,a')$. In our application, we compute the value function $V^\star$ in a pre-process. Once this value function is known, the optimal policy in Equation 5 can be efficiently evaluated at run-time.

3.2. Regressions

In the case of discrete state spaces, a simple approach to compute the value function $V^\star$ is the so-called *value iteration* [SB98]. This approach, however, does not apply to continuous state spaces. Instead, a general strategy is to use a *fitted iteration algorithm*. These algorithms start with an initial value function V that equals zero everywhere on the state space, and iterate towards a better V by running the following steps until convergence:

1. Generate a set of state samples s, compute their long-term state rewards v based on the current value function, and add each pair (s,v) to a training set $\mathcal{T}$,
$$\mathcal{T} \leftarrow \mathcal{T} \cup \{(s,v) | s \in \mathcal{S}, v \in \mathbf{R}\}.$$
The long term reward based on the current value function is given by
$$v = \max_{a' \in \mathcal{A}} \left[R(s,a') + \alpha V(s')\right], \tag{6}$$
where $s' = f(s,a')$.
2. Use a regression algorithm to update V using the training set $\mathcal{T}$ so that, for any tuple (s,v) in $\mathcal{T}$, $V(s)$ is as close to v as possible. Update all tuples in $\mathcal{T}$ using the new V.

Note that the value of v would equal to $V(s)$ for all possible $s \in \mathcal{S}$ if and only if $V = V^\star$. The goal of the regression is to improve V to better approximate $V^\star$ in every iteration.

Designing a suitable regression algorithm for Step 2. is crucial for the robustness and efficiency of reinforcement learning. Different alternatives have been proposed in the literature. Lee and Lee [LL04] used piece-wise linear grids

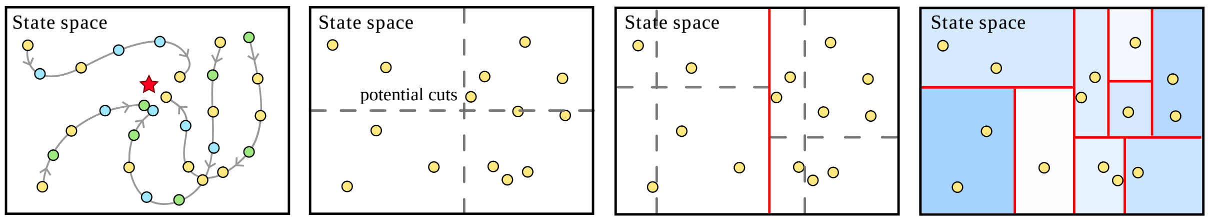

Figure 1: *Illustration of one step in our tree-based fitted iteration. The figure shows an abstract visualization of state space; it should not be interpreted as the 2D position of a character. Please refer to Section 5 for the precise definition of state space for different motion controllers. (a) We use the current value function to generate trajectories that approach the goal, depicted by the red star. Each circle represents a sample state. The colors denote different clips included in the states. (b) Each sample is added to a training set according to the clip it includes. For each training set we build a separate tree in a top-down manner starting with the whole space as the root node. We generate a potential cut randomly for each dimension and select the cut with highest score. (c) We recursively split the nodes until the number of samples in each node is below or equal to n_{min}. (d) After tree construction, the leaf nodes of the tree build a piece-wise constant approximation of the value function.*

as their approximation architecture for precomputing avatar behavior from human motion data. Although they obtain good results, their approach is limited. For carefully tuned grids this type of approximation architecture can lead to good results. However, it is not very robust with respect to grid size and slight variations may strongly influence the result [EGW05].

Treuille et al. adopted parametric regressions [TLP07] for training animation controllers. They approximate the value function by a linear combination of manually designed basis functions, which are either polynomials or Gaussians. Their approach is efficient in storage. However, it relies on the assumption that the desired value function can be approximated using only few basis functions. Their regression algorithm becomes very inefficient for larger number of basis functions, because it requires the solution of a linear programming problem in each iteration. In addition, we have observed that this approach often does not converge if the basis functions are not carefully chosen to match the shape of the value function. We show more comparison results in Section 5.2.

Ernest et al. [EGW05] provide a comprehensive comparison among several regression methods for reinforcement learning, including kNN, piecewise constant and piecewise linear grids, and various tree-based methods. Their study of several application cases shows that the so-called *Extra-trees* [GEW06] perform significantly better than the other methods. Therefore, we use this approach for regression in Step 2., and we will present details of our fitted iteration algorithm in Section 3.3.

Generally, a regression tree partitions the training set $\mathcal{T}$ into several regions and determines a constant prediction in each region by averaging the values of the elements from the training set that fall into this region. The Extra-trees algorithm builds the partition in a top-down manner. For each node, it selects a random cut position for each dimension. It then computes a score for each of the potential cuts, and chooses the one that maximizes the score. The algorithm stops splitting a node when the number of samples in this node is less than a parameter n_{min}.

3.3. Tree-based Fitted Iteration for Motion Controllers

Our state space $\mathcal{S}$ is continuous except for the dimension along the clips A, which is discrete. Hence, we build a regression tree for each clip in the database, but we optimize all the trees at the same time. Here, we present our tree-based fitted iteration algorithm that includes three steps: initialization, iteration, and pruning.

Initialization. We initialize the value function V to zero everywhere on S. We maintain a training set $\mathcal{T}_A$ for each clip A. The training sets are initialized as empty sets.

Iteration. We illustrate one step of our iteration process in Figure 1. In each step, we first add samples to the training sets $\mathcal{T}_A$ by generating a number of trajectories in the system. Each trajectory starts from an randomly chosen initial state and finishes when the task goal is achieved, or when a predefined maximal number of steps is reached. During the trajectories, the action a_{t+1} selected at time t is chosen according to Equation 5 using the current value function V. For every state s generated in the trajectories, we compute v using Equation 6, and then add (s, v) to the the training set of the chosen clip.

After updating the training sets $\mathcal{T}_A$, we build an Extra-tree for regression for each training set. We treat the whole state space as a root node, and recursively split the nodes until the number of samples contained in each node is equal or less than n_{min}. To determine a split at each node, we randomly pick a cut position for each dimension and compute a corresponding score that measures the relative variance reduction,

$$Score = \frac{var(\mathcal{N}) - \frac{\#\mathcal{N}_l}{\#\mathcal{N}} var(\mathcal{N}_l) - \frac{\#\mathcal{N}_r}{\#\mathcal{N}} var(\mathcal{N}_r)}{var(\mathcal{N})}, \quad (7)$$

where $\mathcal{N} \subseteq \mathcal{T}_A$ denotes the subset of samples in the node, $\mathcal{N}_l$ and $\mathcal{N}_r$ denote the samples on the two sides of the cut in the node, and *var* is the empirical variance of the sample values v. We make the cut with highest score. When no more nodes can be split the value function V is updated using the new regression trees, and all existing tuples in the training sets are also updated using Equation 6.

The sampling and regression steps are repeated until a stopping condition is reached. We measure the quality of the current value function using the Bellman residual [Bai95], which is defined as the difference between the two sides of the Bellman equation 4,

$$V(s) - \max_{a' \in \mathcal{A}} \left[R(s,a') + \alpha V(s') \right]. \quad (8)$$

Note that $V^\star$ is the only function leading to a zero Bellman residual for every possible state. Therefore, the residual measures how close the value function is to the optimal one. In our system, we compute the mean square of the Bellman residual over the training sets. The iteration is stopped if the residual is below some predefined threshold.

Pruning. It is nontrivial for the user to specify n_{min}, and the optimal value may vary for different tasks and for different sizes of training sets. Therefore, we use pruning as a post-processing step to automatically determine the maximal number of samples in a leaf. Pruning is carried out by selecting at random two thirds of the elements of $\mathcal{T}$, re-building trees for every possible value of n_{min} with this smaller training set, and determining with which value of n_{min} the square error over the last third samples is minimized. Then, we run the Extra-trees algorithm again on the whole training set $\mathcal{T}$ using this optimal value of n_{min}.

3.4. Convergence

Although the Extra-trees algorithm can well extract information from the training sets, it does not guarantee convergence, since it readjusts the approximation architecture, i.e., the tree structure, to the new expected rewards at each iteration. However, and contrary to many parametric approximation schemes, it does not lead to divergence to infinity problems, but just oscillates around some value [EGW05]. To ensure convergence in our system, we freeze the tree structure and stop adding new samples after it begins to oscillate or after the number of iterations exceeds some predefined number. It converges fast with a frozen tree structure.

4. Incorporating Parameterized Motion Groups

One of the challenges of character animation based on motion data is that it may require large databases and excessive sampling of the continuous space of motions to allow for precise control of generated motion. Moreover, in our context it would lead to large precomputation and memory requirements to learn and store a value function for each clip in a large database. Here, we present an approach to incorporate parameterized motion groups in our reinforcement learning framework. We effectively reduce the number of actions by clustering similar motions, alleviating the precomputation and storage cost. In addition, each cluster forms a parameterized subspace of motion. This allows us to obtain precise control over the synthesized motion by continuously interpolating in this space. In the following paragraphs, we first explain how we pre-process our motion data. Then we describe how we cluster motion clips into groups and how we define transition costs between clusters. Finally we demonstrate how to modify the transition function so that the planning controller can work with parameterized groups.

Pre-processing. We manually segment motion data into short clips. Our current database includes walking and grasping motions. We define constraint frames for each clip similar to Treuille et al. [TLP07]. The constraint frames are used to temporally align consecutive clips. This allows us to construct a valid animation from any sequence of clips, and to prevent foot-skating. We do not need to construct an explicit graph structure, because unnatural and non-smooth transitions will be avoided by the reinforcement learning approach with the transition reward. To construct a desired controller, we design a state space represented by parameters $x_1, \ldots, x_n$. We parameterize each clip by determining the change $dx_1, \ldots, dx_n$ caused by the clip. We illustrate this using a navigation controller in Figure 2a. For more examples we refer to our results in Section 5.

Clustering. In the clustering step, we group similar motions together to share a single value function. Because clips are characterized by their instantaneous reward for the purpose of learning, it is reasonable to classify clips with similar reward functions into one cluster. Our reinforcement learning algorithm is then based on clusters of motion clips rather than on individual clips.

To determine clips with similar rewards, remember that the reward is composed of a transition reward and a state reward. Any two motions that are similar numerically [KG04] will have a similar transition reward. On the other hand, the state reward measures how well a state fulfills the goal of the controller. According to the transition function in Equation 2, if two motions have similar parameters $dx_1, \ldots, dx_n$, they will lead to similar states. Therefore they will have similar state rewards. Following these observations, we group together motions that are numerically similar and close in the parametric space. In our current implementation, clustering is performed manually. We illustrate clustering for the navigation controller in Figure 2b. Note that a clip may belong to several clusters. This allows us to make sure that the parameter domain is covered completely by the clusters.

Clusters allow us to interpolate in a continuous space of motions spanned by the cluster members. We use registration curves [KG03] to perform the interpolation. In other words, each cluster represents a range of motions instead of a single clip. This allows us to achieve more precise control as shown in Section 5.

Transition Cost. We define the transition cost $R_t(a,a')$ between two clusters simply as the average of the pairwise transition costs of all clips in the two groups. This is justified because the clips in each group are similar and each pair of clips will have a similar cost.

Parametric Transition Function. We define a modified transition function to incorporate parameterized groups into the reinforcement learning framework. Now an action $a \in \mathcal{A}$ corresponds to the selection of a parameterized group, which represents a continuous range of motions. We determine the long-time reward for each group by finding the one motion represented by the group that maximizes the value function.

Assuming there are m members in the group, we use blending weights $\{w_1 \dots w_m\}$ to characterize any motion interpolated in the group. We further denote the change of parameter i induced by member j of the group by dx_{ij}. The parametric change given by a set of blending weights is therefore the vector

$$\left(\sum_{j=1}^{m} w_j dx_{1j}, \dots, \sum_{j=1}^{m} w_j dx_{nj}\right).$$

Given a current state $s = (a, x_1, \dots, x_n)$, the maximum of the value function for a potential next action a' happens at

$$\{w_1 \dots w_m\} = \underset{\substack{0 \le w_1,\dots,w_m \le 1 \\ w1+\dots+w_m=1}}{\operatorname{argmax}} V^\star\left(a', x_1 + \sum_{j=1}^{m} w_j dx'_{1j}, \dots, x_n + \sum_{j=1}^{m} w_j dx'_{nj}\right). \quad (9)$$

This is also illustrated in Figures 2c and 2d. In our implementation we solve the above equation by uniformly sampling the space of blending parameters $\{w_1 \dots w_m\}$ and picking the one with the highest value. Once the blending weights are determined, the transition function for any state is

$$f(s,a') = s' = \left(a', x_1 + \sum_{j=1}^{m} w_j dx'_{1j}, \dots, x_n + \sum_{j=1}^{m} w_j dx'_{nj}\right).$$

At run time, whenever a clip finishes, the controller uses the optimal policy described by Equation 5 to select the next group. This is achieved by scanning every group as a potential next action a'. We compute the optimal blending parameters with Equation 9 for each group, and obtain the corresponding state s'. We also evaluate the transition cost $R_t(a,a')$ between the current group a and each candidate a'. We select the next group a' that maximizes $R_t(a,a') + \alpha V^\star(s')$.

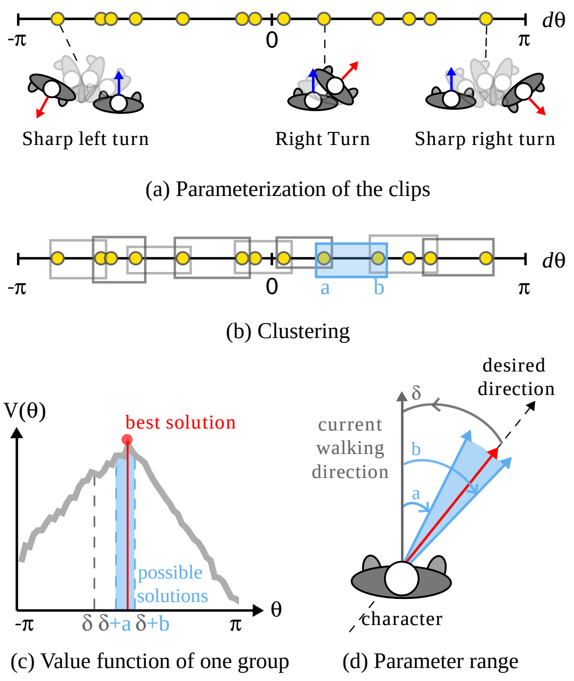

Figure 2: *Parameterized groups for the navigation controller, which has only one parameter θ. The value of θ represents the difference between the current and desired walking directions. (a) The clips (yellow circles) are parameterized by the change in torso orientation dθ. Blue and red arrows indicate the character's orientation in the first and last frame of the clip, respectively. (b) Parameterized groups. Each group has one corresponding value function. (c) The example value function of the group shaded in (b). Assume the current value of θ is δ, the shaded area represents the possible values of θ in the next time step, with the use of any motion from this group. (d) Visualization of the variables in (c). If the controller picks the best solution in the shaded area in (c), the character will turn exactly toward the desired direction in the next time step.*

5. Results

We learned three different controllers to demonstrate our method: navigation, grasping, and guidance.

Navigation. The navigation controller allows a user to navigate a character through an environment by specifying its desired walking direction. A state is given by $s = \{a, \theta\}$, where the parameter $\theta \in [-\pi, \pi)$ measures the angle between the current and the desired walking direction. We define the state reward function as

$$R_s(s,a') = -\gamma|\theta|, \quad (10)$$

where γ is a scaling parameter.

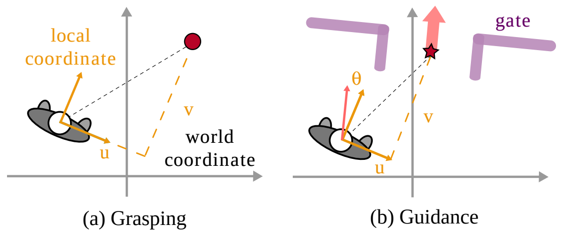

Figure 3: *State variables for the grasping (a) and guidance controller (b). The red star and arrow indicate the position and orientation of the gate, and θ is the angle between the the gate and the walking direction of the character.*

Grasping. With the grasping controller, the goal of the character is to grasp an object that is at an arbitrary position relative to the character. The object can be moved around in real-time, and the character will immediately adjust its path accordingly. Once the character is within reach of the object, the controller picks an optimal way to grasp it according to its relative position.

We define a state for this controller as $s = \{a, u, v\}$, where u and v represent the position of the object projected onto the ground plane in local coordinates of the character, as shown in Figure 3a. The state reward function is

$$R_s(s, a') = \begin{cases} \lambda & \text{if the object is reachable,} \\ 0 & \text{else,} \end{cases} \tag{11}$$

where λ is a constant. The motion database for this controller includes walking steps and grasping motions. All clips are parameterized with the change in the object's relative position and clustered as described in Section 4. The parameterized grasping motions allow the character to reach the object accurately using motion blending.

Guidance. The goal of this controller is to guide a character to walk through a gate without bumping into walls and doors. We define a state as $s = \{a, \theta, u, v\}$. Here, (u, v) is the position of the center of the gate and θ its orientation, both relative to the character. We illustrate the set-up in Figure 3b. Our state reward function is

$$R_s(s, a') = \begin{cases} -\lambda & \text{if close to the obstacles,} \\ -\gamma_1|\theta| - \gamma_2 \frac{v}{u^2+v^2} & \text{else,} \end{cases} \tag{12}$$

where γ_1 and γ_2 are scaling parameters, and λ is a very large constant. Intuitively, this reward function penalizes states where the character is close to the walls and doors around the opening of the gate. In addition, it guides the character towards the center of the gate while maintaining the appropriate torso orientation to pass through the gate.

Figure 4 shows the statistics of our controllers. At run time, it takes less than 1ms to select the best next clip on a Quad Core 2.4 GHz Intel processor.

	Time	#groups	#$\mathcal{T}$	Storage
Navigation	5 s	18	13752	138KB
Guidance	4 min	8	48000	125KB
Grasping	20 min	17	178007	460KB

Figure 4: *Statistics of our controllers, where Time means the learning time.*

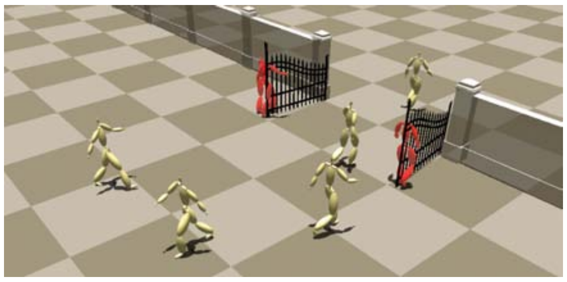

Figure 5: *The greedy strategy in the guidance controller. In this experiment five out of ten characters bumped into the gate. With our planning controller, however, every character successfully enters it. The starting position and orientation of each character is randomly decided in real-time.*

5.1. Motion Planning

One of the advantages of motion planning is that we can generate animations by just specifying a goal, rather than indicating how a character should fulfill a task. We simply formulate the goal as a reward function. For example for our grasping controller, we only need to specify that the character will obtain a reward if he reaches the object. We do not need to tell the character how to approach the object. In a traditional greedy controller, which picks the best next action without planning into the future, this approach would fail completely. The greedy policy is usually defined as

$$\Pi_{greedy}(s) = \underset{a' \in \mathcal{A}}{\operatorname{argmax}} \left[R_s(s') + R_t(a, a') \right]. \tag{13}$$

In the grasping controller the state reward $R_s(s')$ for every possible next state s' is zero when the character is far away from the object. Therefore the character would just walk away as shown in Figure 9b. Planning also allows characters to prepare well for obstacles. For example, if we use the greedy strategy with the guidance controller, characters often run into the gate as illustrated in Figure 5. With our approach the characters plan several steps ahead and they successfully avoid the obstacles.

5.2. Extra-trees Regression

We compare the navigation controller learned with the work proposed by Treuille et al. to our method. We use twenty motion clips for training, and each clip contains one single walk cycle. Treuille et al. approximate the value function by solving the coefficient vector **r** of 2nd-degree polynomials

$V \approx r_1 + r_2\theta + r_3\theta^2$. They observed that the quadratic polynomials are only about 11% different from 10th-degree polynomials. Since their method takes much longer to learn with higher order polynomials, they adopted 2nd-degree polynomials for this controller. We use two metrics to asses the performances of the algorithms. The first one measures the quality of the regressions: we randomly sample one thousand states (which may or may not be in $\mathcal{T}$) and average the Bellman residuals in Equation 8. The lower the residual, the closer the approximation is to the optimal value function. The second metric measures the quality of the policy: we randomly sample one thousand initial states, trace ten steps from them individually by each policy, and average the long term rewards achieved. The higher the average reward, the better the policy performs. We show quantitative results in Figure 6, illustrating that our method is scores much better according to these metrics. Our learning time is also much shorter, although we need more memory, because we store tree representations of the value function, while they only need to store coefficient vectors **r**. In Figure 7 we visualize the approximated value functions for one clip. This shows that the second order polynomial is not an accurate model for the navigation controller.

	$\#\mathcal{T}$	Time	Bellman residual	Average reward	Storage
[TLP07]	15920	7s	6490.31	-328	1k
Ours	15960	3s	48.45	-231	198k

Figure 6: *Comparison between previous work and our regression method.*

We also did the same experiment with the fixed-obstacle-avoidance controller proposed in previous work [TLP07]. Treuille et al. never require more than one hour to learn the controller. Our approach yields a useful controller in less than ten minutes.

For more complex control systems it is hard to approximate the shape of the value function well with a small number of manually designed basis functions. In our experience, the regression algorithm [TLP07] often fails to converge if the basis functions are chosen inappropriately. In contrast, our tree-based algorithm does not make any assumptions about the shape of the value function. Figure 8a illustrates the complex shape of the value function for the grasping controller. We used dense sampling (about 2 million samples) to obtain a "ground truth" solution in about three hours. Polynomials and gaussians are ill-suited for approximating value functions such as this one. It is asymmetric, has abrupt changes and even a pit in the middle; but this does not pose a problem for our algorithm. The v axis is asymmetric because it is better to face an object to grasp it, rather than turning one's back to it. Since the reward for successfully grasping the object is a constant, there is a constant plateau near the center. The plateau is slightly off-center because grasping is performed with the right hand. The other plateaus represent

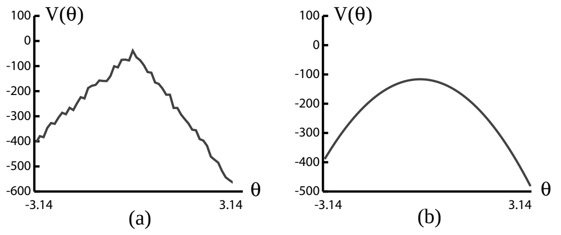

Figure 7: *Value function of the navigation controller. (a) Tree-based regression. (b) Regression using polynomials.*

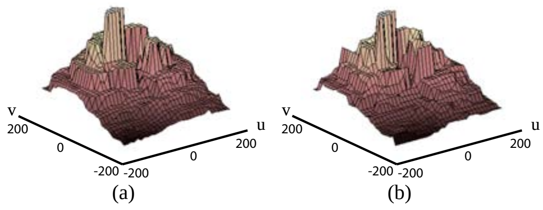

Figure 8: *Value function of the grasping controller. (a) "Ground truth" using about two million samples, three hours learning time. (b) Regression from six thousand samples, less than three minutes learning time.*

the steps needed to walk toward the peak region. For example, if the character's state is in the first plateau near the peak, he needs one more step before being able to grasp. If the character is too close to the object, it is also very difficult to grasp, hence there is a pit in the center of the value function. Figure 8b shows a regression computed from six thousand samples in less than three minutes. Here we store about 1700 leaves in the regression tree. This shows that we can obtain a reasonable approximation in a short time.

5.3. Parametric Synthesis

The advantage of parametric synthesis is that we gain more precise control. As shown in Figure 2c the optimal value may lie between existing motions. Therefore controllers with non-parameterized motions can only pick sub-optimal clips. Figure 11 shows quantitative improvements of parameterized motions for the navigation controller. Figures 9c and 9d show a comparison with the grasping controller. Without parameterized motions, none of the existing clips initially leads to a good position for grasping the object, so the character takes a detour (Figure 9c). With parametric synthesis, a novel motion with optimal value is synthesized and the character turns immediately to grasp the object (Figure 9d).

	$\#\mathcal{T}$	Time	Bellman residual	Average reward
non-parametric	15960	3s	48.45	-231
parametric	13752	5s	29.63	-161

Figure 11: *Comparison between parametric and non-parametric planning controllers.*

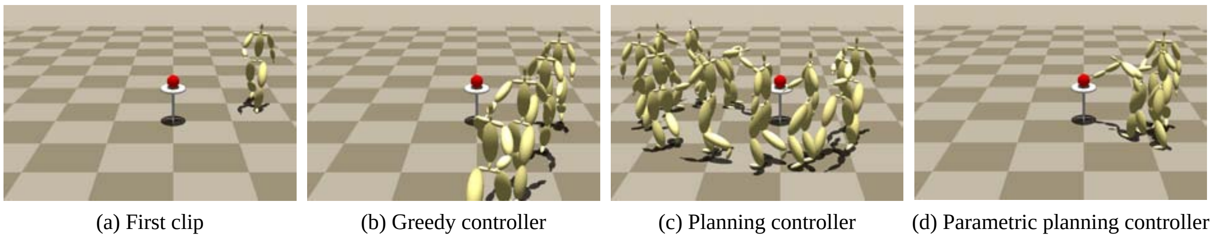

Figure 9: *Comparison of different grasping controllers. (a) Every controller starts with the same first clip. (b) Greedy controller. (c) Planning controller with non-parameterized motions. (d) Planning controller with parameterized motions.*

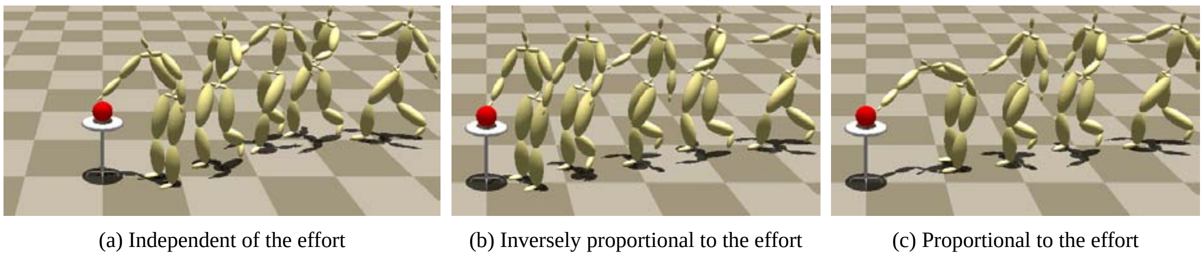
(a) Independent of the effort
(b) Inversely proportional to the effort
(c) Proportional to the effort

Figure 10: *Real-time near-optimal control. The character is asked to grasp the object (a) in any way, (b) in the easiest way, and (c) in the hardest way.*

5.4. Near-optimal Control

In this experiment, we demonstrate that our controller makes near-optimal decisions in real-time. We modified the reward function of the grasping controller to take into account the effort required for the grasping motion. In general, the effort needed to perform a motion can be approximated by the sum of squared torques computed via inverse dynamics. Figure 10b shows the result when the reward is inversely proportional to the effort required for grasping. This motivates the character to step close to the object and pick it in the easiest way. If the reward is positively proportional to the effort required for grasping the character stops at a distance and picks the object in the hardest way, as shown in Figure 10c. This demonstrates that the character plans ahead to maximize the long term reward.

6. Conclusions

We presented a reinforcement learning framework to obtain motion controllers with parameterized motions. We use a tree-based fitted iteration algorithm to approximate the optimal long-term reward function. This approach is more flexible and more robust than previous methods, and enables us to design reward functions in a straightforward way. We also described how to incorporate parameterized motions into the learning framework. This allows us to control characters more precisely with a limited amount of input data. We demonstrate that our approach generates natural animation in real-time for different tasks that require planning.

We believe that the major limitation of our approach is the problem of dimensionality. For more complex environments we need more control parameters to define the state space. Unfortunately, computing time and memory requirements increase superlinearly with the number of dimensions, as more motion data and training samples are required to properly cover the state space. Hence, it is important to sample the high dimensional space effectively. The active learning framework proposed by Cooper et al. [CHP07] could be useful, for it can adaptively determine which motions to add to the system, avoiding capturing and storing nonessential motions. Shum et al. [SKY08] also utilize reinforcement learning for human interactions, and they observe that the subspace of meaningful interactions occupies only a small fraction of the whole state space. They thus propose a way to efficiently collect samples by exploring the subspace where dense interaction occurs. This sampling strategy might also help for high dimensional state spaces. We believe that our approach to include parameterized motions in a reinforcement learning framework is a first step to make this technique more practical. However, future research is required to extend it to more complex environments.

Acknowledgments

We are grateful to Adrien Treuille and Yongjoon Lee for generously providing their motion capture data. We also thank Yen-Lin Lee and Todd Espiritu-Santo for help with motion capture. Thanks to Iman Mostafavi for narrating the video, and to Will Chang, Toshiya Hachisuka, and Jesper Højgaard Pedersen for their helpful feedback.

References

[AF02] ARIKAN O., FORSYTH D. A.: Interactive motion generation from examples. In *SIGGRAPH* (2002), pp. 483–490.

[AFO03] ARIKAN O., FORSYTH D. A., O'BRIEN J. F.: Motion synthesis from annotations. *ACM Trans. Graph. 22*, 3 (2003), 402–408.

[Bai95] BAIRD L. C.: Residual algorithms: Reinforcement learning with function approximation. In *ICML* (1995), pp. 30–37.

[CHP07] COOPER S., HERTZMANN A., POPOVIC Z.: Active learning for real-time motion controllers. *ACM Trans. Graph. 26*, 3 (2007), 5.

[EGW05] ERNST D., GEURTS P., WEHENKEL L.: Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research 6* (2005), 503–556.

[GEW06] GEURTS P., ERNST D., WEHENKEL L.: Extremely randomized trees. *Machine Learning 63*, 1 (2006), 3–42.

[HG07] HECK R., GLEICHER M.: Parametric motion graphs. In *SI3D* (2007), pp. 129–136.

[IAF05] IKEMOTO L., ARIKAN O., FORSYTH D.: Learning to move autonomously in a hostile world. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches* (2005), ACM, p. 46.

[KG03] KOVAR L., GLEICHER M.: Flexible automatic motion blending with registration curves, 2003.

[KG04] KOVAR L., GLEICHER M.: Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph. 23*, 3 (2004), 559–568.

[KGP02] KOVAR L., GLEICHER M., PIGHIN F. H.: Motion graphs. In *SIGGRAPH* (2002), pp. 473–482.

[KLM96] KAELBLING L. P., LITTMAN M. L., MOORE A. P.: Reinforcement learning: A survey. *J. Artif. Intell. Res. (JAIR) 4* (1996), 237–285.

[KS05] KWON T., SHIN S. Y.: Motion modeling for online locomotion synthesis. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), ACM, pp. 29–38.

[LCL06] LEE K. H., CHOI M. G., LEE J.: Motion patches: building blocks for virtual environments annotated with motion data. *ACM Trans. Graph. 25*, 3 (2006), 898–906.

[LK05] LAU M., KUFFNER J. J.: Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), ACM, pp. 271–280.

[LL04] LEE J., LEE K. H.: Precomputing avatar behavior from human motion data. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2004), ACM Press, pp. 79–87.

[LL06] LEE J., LEE K. H.: Precomputing avatar behavior from human motion data. *Graphical Models 68*, 2 (2006), 158–174.

[MK05] MUKAI T., KURIYAMA S.: Geostatistical motion interpolation. *ACM Trans. Graph. 24*, 3 (2005), 1062–1070.

[MP07] MCCANN J., POLLARD N. S.: Responsive characters from motion fragments. *ACM Trans. Graph. 26*, 3 (2007), 6.

[PSKS04] PARK S. I., SHIN H. J., KIM T.-H., SHIN S. Y.: On-line motion blending for real-time locomotion generation. *Journal of Visualization and Computer Animation 15*, 3-4 (2004), 125–138.

[RCB98] ROSE C., COHEN M. F., BODENHEIMER B.: Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications 18*, 5 (1998), 32–41.

[SB98] SUTTON R., BARTO A.: *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[SH05] SAFONOVA A., HODGINS J. K.: Analyzing the physical correctness of interpolated human motion. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), ACM, pp. 171–180.

[SH07] SAFONOVA A., HODGINS J. K.: Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph. 26*, 3 (2007), 106.

[SKG05] SUNG M., KOVAR L., GLEICHER M.: Fast and accurate goal-directed motion synthesis for crowds. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), ACM, pp. 291–300.

[SKY08] SHUM H. P. H., KOMURA T., YAMAZAKI S.: Simulating interactions of avatars in high dimensional state space. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (2008), ACM, pp. 131–138.

[SO06] SHIN H. J., OH H. S.: Fat graphs: constructing an interactive character with continuous controls. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2006), Eurographics Association, pp. 291–298.

[TLP07] TREUILLE A., LEE Y., POPOVIC Z.: Near-optimal character animation with continuous control. *ACM Trans. Graph. 26*, 3 (2007), 7.

[WH97] WILEY D. J., HAHN J. K.: Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Applications 17*, 6 (1997), 39–45.