Published in ACM Transactions on Graphics 26(3) (SIGGRAPH 2007).

Active Learning for Real-Time Motion Controllers

Seth Cooper[1] Aaron Hertzmann[2] Zoran Popović[1]

[1]University of Washington [2]University of Toronto

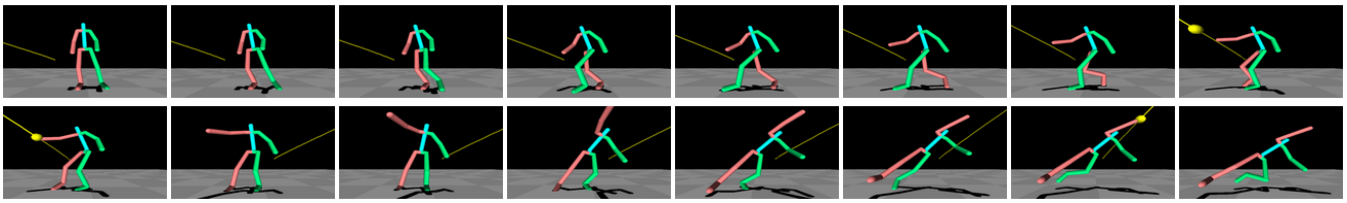

Figure 1: Catching controller: the character moves through space to catch two consecutive balls thrown from different directions. In addition to the current character state, the controller has a three-dimensional input space that specifies the incoming position and speed of the ball to be caught.

Abstract

This paper describes an approach to building real-time highly-controllable characters. A kinematic character controller is built on-the-fly during a capture session, and updated after each new motion clip is acquired. Active learning is used to identify which motion sequence the user should perform next, in order to improve the quality and responsiveness of the controller. Because motion clips are selected adaptively, we avoid the difficulty of manually determining which ones to capture, and can build complex controllers from scratch while significantly reducing the number of necessary motion samples.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: Motion Capture, Human Motion, Active Learning

1 Introduction

Human motion capture data provides an effective basis for creating new animations. For example, by interpolating and concatenating motions, realistic new animations can be generated in real-time in response to user control and other inputs. In this paper, we consider such an animation model that we refer to as a *motion controller*: a controller generates animation in real-time, based on user-specified *tasks* (e.g., a user might press forward on a game controller to specify the task "walk forward,"). Each task is parameterized by a *control vector* in a continuous space (e.g., "catch the ball flying from a specific direction and velocity"). In this paper, a controller is essentially a function from the combined space of states and tasks to the space of motions. Our controllers are kinematic: they produce the character's motion by interpolating motion capture, rather than producing motion dynamically through forces and torques.

Motion capture data is very time-consuming and expensive to acquire, and thus it is desirable to capture as little as possible. When building the simplest controllers, a designer can minimize the amount of data captured by carefully planning the data samples to be acquired. However, for non-trivial tasks, the space of possible clips is vast, and manual selection of samples quickly becomes intractable. For example, a controller for a human that can walk and dodge projectiles must be parameterized by the direction and speed of walking, the direction and speed of the projectiles. Because the task can be changed at any point during the animation, the controller also needs to be parameterized by all possible pose configurations of the character. For example, the controller must be able to dodge any projectile that appears while the character is walking, turning, or recovering from the previous dodge. Even this moderate set of tasks leads to a huge set of initial configurations and control vectors. Determining which motions to capture for this controller — so that it can produce good motions in this huge space of possible inputs — is a daunting task, and, in our experience, is too difficult to do manually. Uniform sampling of the input space would be vastly inefficient, requiring a huge number of samples to be captured and then stored in memory. Because the space is structured — that is, nearby motions in the space can often be interpolated to produce valid new motions — selection of which motions to capture should greatly reduce the number of samples needed. However, even in a small controller space, nonlinearities in the space will mean that careful sampling is required.

In this paper, we propose the use of active learning to address these problems. In particular, we build the motion controller on-the-fly during the data acquisition process. After each motion is captured, the system automatically identifies specific tasks that the controller performs poorly, based on a suite of error metrics. These tasks are presented to the user as candidates for additional data samples. The user chooses one of the tasks to perform, and the controller is updated with this new motion clip. In this way, the system continues to refine the controller until it is capable of performing any of the tasks from any state with any control vector, the time available for capture has elapsed, or else the number of data samples has been reached a predefined maximum. This process yields highly-controllable, real-time motion controllers with the realism of motion capture data, while requiring only a small amount of motion capture data. We demonstrate the feasibility of this approach by using our system to construct three example controllers. We validate the active learning approach by comparing one of these controllers to manually-constructed controllers.

We emphasize that, by design, our system does not automate all decisions, but, rather, computes aspects of the controller that would be difficult for a user to handle. The user is left with specific decisions which are hard to quantitatively evaluate. Making these decisions normally entails selecting from among a few alternatives

presented by the system.

In this paper, we refer to a generic "user" that runs the system. In practice, some roles may be performed by separate individuals. For example, a human operator might run the active learning software, but have an actor perform the actual motions; later, a separate non-specialist user (e.g., a game player) may control the motions with the learned controller.

Contributions. A key contribution of this work is the use of an active learning method for animation that produces compact controllers for complex tasks. We also develop a framework that includes user input at key points of the process, including providing motion capture samples, and in selecting which motions to capture from a few automatically-determined options. We also develop a cluster-based learning model for motion controllers.

2 Related Work

A common theme in computer animation research is to create new motion from existing motion capture data. Most methods create animation off-line, for example, by interpolating a similar set of motions according to user-specified control parameters [Witkin and Popović 1995; Kovar and Gleicher 2004; Mukai and Kuriyama 2005; Rose et al. 1998; Wiley and Hahn 1997], by optimizing motion according to probabilistic time-series models [Brand and Hertzmann 2000; Li et al. 2002], by concatenating and blending example sequences [Arikan et al. 2003; Kovar et al. 2002; Torresani et al. 2007], or by combining model- and data-driven techniques [Yamane et al. 2004; Zordan et al. 2005; Liu and Popović 2002; Liu et al. 2005]. These methods generate motion off-line, whereas we consider the problem of real-time synthesis. It is worth noting that many of these methods typically require large motion databases; the need for such databases could be mitigated by our active learning approach.

A number of real-time animation systems build on motion capture data as well. One approach is to directly play and transition between clips from a motion database [Gleicher et al. 2003; Lee et al. 2002; Lee et al. 2006]; precomputation can be used to allow real-time planning of which clips to use [Lau and Kuffner 2006; Lee and Lee 2006]. Reitsma and Pollard [2004] present a method for evaluating the possible motions generated by such approaches. A few authors have described methods that generate new poses in response to real-time input. Our motion controller model is most similar to methods that transition between interpolated sequences. Park et al. [2004] and Kwon and Shin [2005] combine interpolation of motions with a graph structure to generate new locomotion sequences. Shin and Oh [2006] perform interpolation on graph edges for simple models of locomotion and other repetitive motions. In previous work, it is assumed that a corpus of motion data is available in advance, or that a user will manually select which motions to capture. In this paper, we show how the use of adaptive selection of motion sequences allows the creation of controllers with greater complexity, while allowing fine-scale parameterized control and capturing relatively few motions overall.

Data acquisition is difficult, expensive, and/or time-consuming for problems in many disciplines. Consequently, automatic selection of test cases has been extensively studied. In statistics, optimal experimental design methods seek the most informative test points to estimate unknown nonlinear functions [Atkinson and Donev 1992; Santner et al. 2003]. The simplest methods determine all test points in advance, e.g., via space-filling functions, or by optimizing an objective function. However, it is often not possible to determine in advance which regions of input space will need the most data. Active learning methods select test data sequentially: after each data point is acquired, the next test point is chosen to maximize an objective function [Cohn et al. 1994]. Active learning has been studied most extensively for classification problems (e.g., [Lindenbaum et al. 2004]). In this paper, we present an active learning algorithm for motion controllers. Our approach is distinct from existing active learning methods in two ways: first, instead of choosing the next sample to capture, our system identifies a set of candidates, from which a user chooses a sample to improve; second, we assume that a metric of correctness is provided by which candidates may be chosen.

3 Motion Controllers

Before describing our active learning framework — which is the main contribution of this paper — we will briefly discuss our model for motion controllers.

In our framework, a kinematic controller $\mathscr{C} : \mathbb{S} \times \mathbb{T} \times \mathbb{U} \rightarrow \mathbb{M}$ generates a motion clip $\mathbf{m} \in \mathbb{M}$ that starts at character state $\mathbf{s} \in \mathbb{S}$ and performs task $\mathbf{t} \in \mathbb{T}$ parameterized by the control vector $\mathbf{u} \in \mathbb{U}$. $\mathbb{S}$ defines a set of all permissible character states, $\mathbb{T}$ is a discrete set of controller tasks, $\mathbb{U}$ defines the operational range of the control inputs for task $\mathbf{t}$, and $\mathbb{M}$ defines the space of output motions. A single controller can learn and integrate several tasks, each specified by a value of the argument $\mathbf{t}$. For example, a catching controller consists of two tasks: one that catches the ball coming at given speed and direction, and the idle controller that is invoked when there is no ball to be caught. We take advantage of the fact that the controller produces continuous motion clips rather individual states, and solve for the new motion $\mathbf{m}$ only when the task parameters $\mathbf{u}$ change or when the current motion $\mathbf{m}$ finishes.

We represent a motion clip $\mathbf{m} \in \mathbb{M}$ as a sequence of poses. In order to compare, blend, and visualize motions in a translation- and rotation- invariant way, we decouple each pose from its translation in the ground plane and the rotation of its hips about the up axis. We represent changes in position and rotation relative to the previous pose's local frame. We represent the *state* $\mathbf{s}$ of the character as a vector containing the poses in the next ten frames of the currently-playing clip. The use of multiple frames for state facilitates finding smooth motion blends. We determine the distance between two states with a method inspired by Kovar et al. [2002], by evaluating the distance between point clouds attached to each corresponding pose.

We consider data-driven controllers which create a new clip $\mathbf{m}$ by interpolating example motion capture clips $\{\mathbf{m}_i\}$ associated with task parameters $\{\mathbf{u}_i\}$. However, not all sampled clips can be blended together in a meaningful way. Consider a tennis controller: it would not make sense to blend forehand and backhand stroke motion samples. For this reason, each controller consists of groups of blendable clips which we refer to as "clusters." Each cluster $\mathbf{C}_j$ contains a set of blendable motion samples $\{(\mathbf{m}_{k,j}, \mathbf{u}_{k,j})\}$ that share a common start state $\mathbf{s}_j$, and a continuous blend map $\mathbf{w} = \mathbf{b}_j(\mathbf{u})$ which produces motion blend weights $\mathbf{w}$.

Given a state $\mathbf{s}$, a task $\mathbf{t}$, and a control vector $\mathbf{u}$, applying a controller entails two steps. First, we find the $\mathbf{t}$-specific clusters with $\mathbf{s}_j$ closest to $\mathbf{s}$; if there is more than one, we use the one which has a $\mathbf{u}_{k,j}$ closest to $\mathbf{u}$. Second, the output motion is generated by computing the blend weights $\mathbf{w} = \mathbf{b}_j(\mathbf{u})$, and then interpolating with these weights: the new clip is $\sum_k \mathbf{w}_k \mathbf{m}_{k,j}$. Interpolation is performed in a time-aligned, translation- and rotation-invariant manner similar to [Kovar and Gleicher 2003]. The new motion is then blended in with the currently-playing motion. However, if the current character state is very dissimilar to the beginning of $\mathbf{m}$, then the new controller motion is ignored, and the current motion continues until the subsequent states $\mathbf{s}$ produce successful controller motion.

In our examples, we also use a modified version of the controller that employs an inverse kinematics (IK) step to further satisfy end-effector constraints (e.g., catching a ball with the right hand). An IK controller $\mathscr{C}_{IK} : \mathbb{S} \times \mathbb{T} \times \mathbb{U} \rightarrow \mathbb{M}$ is defined in terms of the simple blend controller as $\mathscr{C}_{IK}(\mathbf{s}, \mathbf{t}, \mathbf{u}) = IK(\mathscr{C}(\mathbf{s}, \mathbf{t}, \mathbf{u}), \mathbf{u})$. The function IK transforms the motion to satisfy kinematic constraints defined

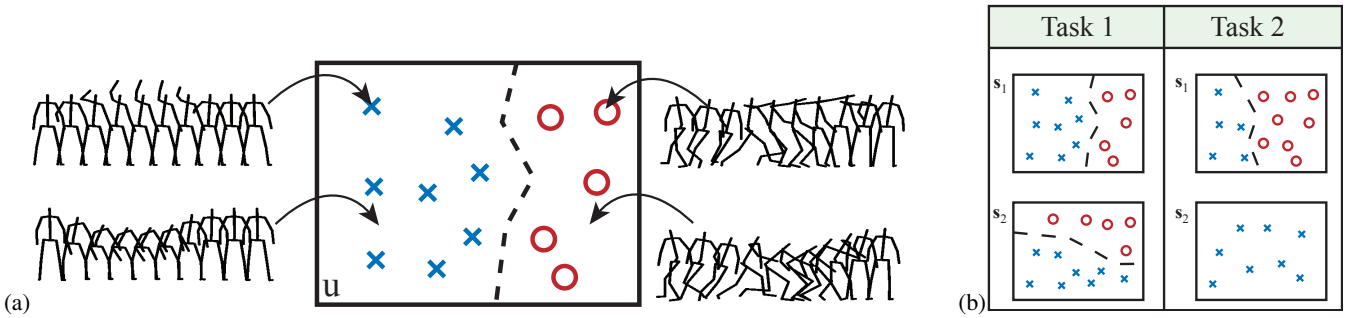

Figure 2: (a) Illustration of a catching controller for a single start state and a 2D controller space $\mathbb{U}$. The space has been partitioned into two clusters, with each cross and circle representing the control vector for an example motion. The motions in the left cluster correspond to catching in place, whereas the motions on the right correspond to taking a step during the catch. The dashed line represents the boundary between clusters implicit in the nearest-neighbor assignment. (b) A motion controller consists of a collection of controllers, each for a specific task and start state.

by **u**. The joint angles for the relevant end-effector (i.e., shoulder and elbow for the hand) are first optimized to satisfy the kinematic constraint in that time instant. The neighboring frames are linearly blended to produce continuous motion. The use of IK enables the controller to achieve a much greater variety of motions from a few samples and thus enables the active learning to use significantly fewer samples overall. In general, the more powerful the controller, the fewer samples active learning requires for full coverage of the controller.

A new clip **m** is usually played from the beginning. However, in some cases, it may be useful for a motion clip to start in the middle. In particular, this occurs when the new motion, if started from the same state as the current motion, would be generated by the same cluster. This implies that the new motion would be generated by blending the same examples as the current motion. In this case, the new motion can be started at the time corresponding to the current time index of the current clip, since all motions belonging to the same cluster can be blended at any portion of the clip. This process is not appropriate for all tasks (e.g. , one cannot change trajectory in midair) and whether or not to allow this is indicated on a per-task basis.

The clustered motion-blend model used in our experiments was motivated by similar models used in the game industry. In real game controllers, specific clusters and the samples within each clusters are all determined in an ad-hoc manual process. This manual process invariably produces suboptimal controllers and multiple trips to the motion capture lab. More importantly, the manual process does not scale well for control problems with a large number of input dimensions, such as those described in the results section.

We point out that the active learning framework introduced in the following sections does not depend on the specific details of the controller specification. In addition, the data samples need not be captured; they can easily come from alternate sources, such as animator-created motion clips.

4 Active Learning

We now describe how motion controllers are built interactively. Although there are multiple steps to the process, the basic idea is simple: identify the regions of the control space that cannot be performed well, and improve them. The process requires the user to first define the control problem by enumerating the set of tasks $\{\mathbf{t}_k\}$ and to specify the operational range of each control vector **u**. Each control vector must have a finite valid domain $\mathbf{u} \in \mathbb{U}_k$ (e.g., bounds constraints) in order to limit the range of allowable tasks[1]. For example, **u** might specify the desired walking speed, and $\mathbb{U}_k$ might be the range of possible walking speeds. A key assumption of our approach is that the motion controller does not need to be able to start from any initial state **s**, but only from states that might be generated by the controller. Hence, we only consider possible starting poses that might arise from another clip generated by the controller. In order to "seed" the active learning, the user provides some initial state **s**.

[1]The input space $\mathbb{U}$ for the controller is defined as a union of the task-specific control spaces: $\mathbb{U} = \bigcup_k \mathbb{U}_k$.

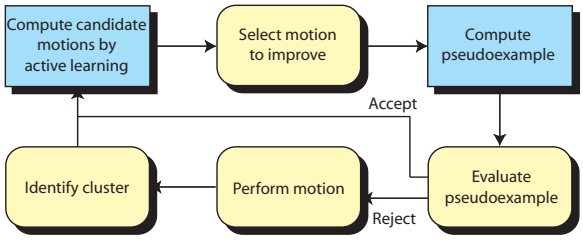

Figure 3: Flowchart of a motion capture session using active learning. Blue rectangles are automatic processes and yellow rounded rectangles require user input.

The active learning process then proceeds in the following outer loop:

1. The system identifies a set of controller inputs $(\mathbf{s}_i, \mathbf{t}_i, \mathbf{u}_i)$ that the motion controller cannot handle well. The user selects one to be improved.
2. The system generates an improved "pseudoexample" motion for this task. If the pseudoexample is approved by the user, the motion controller is updated. This step can often save the user from having to perform the motion.
3. If the pseudoexample is rejected, the user performs the task, and the new motion clip is used to update the motion controller.

We have set up our system in a motion capture lab. The system's display is projected on the wall of the lab for the user to see. The interface is mouse-based so the user can interact with the system using a gyro mouse while capturing motions. We now describe these steps in detail.

4.1 Selecting candidate tasks

The goal of this step is to identify control problems that the motion controller cannot handle well. A candidate problem is fully specified by a start state **s**, a task **t**, and control vector **u**. Because the evaluation of motions can be difficult to do purely numerically, we do not have an explicit mathematical function that can precisely identify which motions are most in need of improvement. Instead, we find multiple candidates according to different *motion metrics*, each of which provides a different way of evaluating the motion generated by a controller, and we let the user determine the candidate to improve. We use the following metrics:

- **Task error.** For each task, the user specifies one or more metrics to measure how well a motion **m** performs task **t** with inputs **u**. For example, if the task requires the character's hand to be at position **u** at a specific time, the metric might measure the Euclidean distance between the character's hand and **u** at that time. Multiple error metrics may be provided for a task, with the goal that they all should be satisfied. It is likely that some regions of control space are simply harder to solve than others, and direct application of this metric will oversample these spaces. To address this, we compute the task error as the difference of the task-performing metric when compared to the nearest existing example.
- **Sum of task errors.** When more than one task metric is provided by the user, a weighted sum of these metrics is also used as a separate metric.
- **Transition error.** In order to evaluate blending between clips, we measure the distance between a current state and the start state **s** of a motion **m**.
- **Distance from previous examples.** In order to encourage exploration of the control space, we use a metric that measures the distance of a control vector from the nearest existing example control vector.

To find the worst-performing regions of controller input space, we need to sample the very large controller input space. We reduce the amount of samples required by using different sampling processes for different motion metrics.

To generate candidates using each task error metric, we search for motions that perform their tasks poorly. For each cluster start state, random control vectors **u** are generated for its task by uniform random sampling in $\mathbb{U}_k$. We keep the **u**'s which give the worst result for each motion metric. Each of these points is refined by Nelder-Mead search [1965], maximizing the badness of their respective metrics. Distance metric candidates are generated similarly, replacing task error with nearest-neighbor distance in $\mathbb{U}_k$. We use the optimized **u**, along with the **t** and **s** which generated them, as candidates.

To generate candidates using transition error, we search for states that will blend poorly with existing clusters. The average motion clip is generated for each cluster by averaging the examples in that cluster. We then find the state **s** in this motion that is the furthest from any existing cluster's start state. We use these states[2], along with **t** and **u** set to the center of the domain $\mathbb{U}_k$, as candidates.

Once all candidates have been generated, the user is then shown the candidates for each metric, sorted by score, along with information about them, such as their error and corresponding **u**. The user then chooses one of the candidates to be improved. Providing the user with several options has several important purposes. First, the user is able to qualitatively judge which sample would be best to improve. This would be very difficult to do with the purely automatic evaluation metrics. Second, the user is able to have some control over the order samples are collected in. For example, it can be more convenient for the user to collect several consecutive samples from the same start state. Third, viewing the candidates gives the user a good sense of the performance of the controller. As more samples are provided, the quality of the candidates improves. Once all of the candidates are considered acceptable, the user has some confidence that the controller is finished. However, there is no formal guarantee as to how the controller will perform in new tasks. In our experience, although many candidates are generated, the user usually finds one worth refining after watching only a few examples.

[2]We also consider the "seed" start state, as well as the ending states of clusters which are distant enough from any start state.

Figure 4: The interactive setup. The user is preparing to perform a walking motion. The projection display shows the desired task and initial state.

4.2 Determining weights

Before proceeding, we will discuss the details of our implementation of the per-cluster blend function, **b**.

We implement blend functions using a scheme similar to Radial Basis Functions (RBFs), based on the method proposed by Rose et al. [1998; 2001]. We use a different form of linear basis, and replace their residual bases R_j with bases that are non-uniform along each dimension:

$$R_j(\mathbf{u}) = \prod_\ell h\left(\frac{|\mathbf{u}_\ell - \mathbf{u}'_{j,\ell}|}{\alpha_{j,\ell}}\right) \tag{1}$$

where ℓ indexes over the elements of a control vector, and h is a cubic profile curve[3], and the α's are scale factors. We believe that other non-isotropic representations could perform equally well. The blend function is then

$$b_i(\mathbf{u}) = a_i + \mathbf{c}_i^T\mathbf{u} + \sum_{j=1}^{N} r_{j,i}R_j(\mathbf{u}) \tag{2}$$

After weights are produced by this function, they are clamped to the range $[0,1]$ and then normalized to sum to 1.

Each time a new example data point $(\mathbf{u}_j, \mathbf{w}_j)$ is provided, we solve for the linear weights a_i and $\mathbf{c}_i$ using least squares. We then initialize all elements of α_j to the distance to the nearest neighbor of $\mathbf{u}_j$, and solve for **r** as in Rose *et al.* We then update each α_j by performing a small optimization across each dimension ℓ. We place a number of evaluation samples along dimension ℓ in the neighborhood of $\mathbf{u}_j$. Then, for regularly sampled values in the range $[1..3]$, we scale $\alpha_{j,\ell}$, solve for **r**, and evaluate the sum of the task error metrics at the evaluation samples; the best scale is kept. We then update α_j by these scales and solve for **r** one last time.

4.3 Generating a pseudoexample

The system first attempts to improve performance at the selected task by generating a new motion called a "pseudoexample" [Sloan et al. 2001]. A pseudoexample is a new training motion defined as a linear combination of existing motions. It is capable of reshaping **b** without requiring an additional motion sample. A pseudoexample can be represented directly in terms of the weights **w** used to generate the motion. These weights are chosen to minimize the sum of the task error metrics for this task. To get an initial guess for the correct cluster and weights for the pseudoexample, we iterate over all clusters that start at the given **s**, and all the motions in the cluster as well as the currently predicted motion at **u**. Of all these motions, the one which performs best on the task error metric is selected as

[3]Specifically, $h(x) = (1-x)^3 + 3x(1-x)^2$ for $0 \le x \le 1$ and $h(x) = 0$ otherwise.

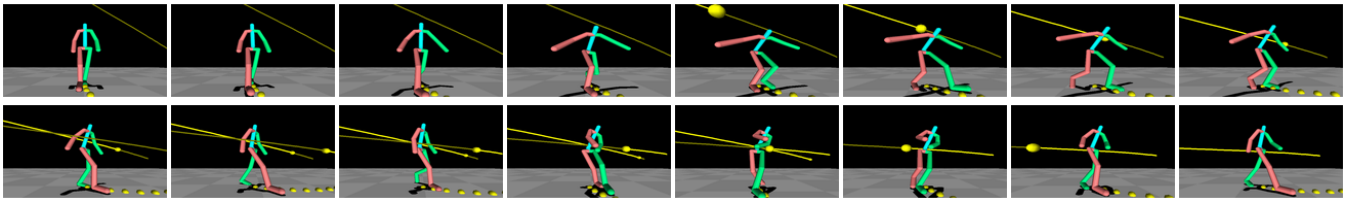

Figure 5: Dodging controller: the character maintains a walking direction while avoiding two consecutive projectiles. In addition to the current character state, the controller has a four-dimensional input space that specifies the incoming position and speed of the ball to be dodged and angle to turn while walking.

providing the cluster and weights for the pseudoexample. We then optimize the weights within the cluster according to the sum of task errors metric using Nelder-Mead [1965]. The resulting weights, together with the control vector, constitute the pseudoexample $(\mathbf{u}, \mathbf{w})$.

The clip generated by these weights and its evaluated task error is then shown to the user. If the user approves the pseudoexample, it is permanently added to this cluster's RBF; otherwise, it is discarded.

4.4 Performing a new motion

The user is presented with the appropriate starting state and a task-specific visualization (e.g., showing the trajectory and initial position of a ball to be caught). Since the motion is determined relative to the character, the visualizations translate and rotate with the user's movement prior to beginning the capture. The user starts moving and eventually gets to (or near) the starting state. Our system automatically determines when the initial state conditions are met, and records the rest of the captured motion. This aspect of the interface significantly simplifies the task of performing the required motion sample. Once captured, the user can review the motion and its task error. If the motion is not satisfactory, the user must repeat the motion. In our experiments, the vast majority of tasks could be performed in three or fewer trials. We found that difficult to perform tasks typically required only a few more trials, usually no more than seven in total, although a few rare cases required up to twelve in total.

To determine where the desired motion sample ends, we seek local minima of state distance in the captured clip, comparing against the beginnings and ends of the existing clusters, as well as considering the actual end of the recorded motion. We sort these by their distance, and let the user select the appropriate ending. This typically involves nothing more than the user confirming the first choice.

The system now needs to determine the appropriate cluster for the new motion. The motion is timewarped to all clusters with similar start and end states, and the alignment results are shown to the user, sorted in increasing order of alignment error. The user can then select which cluster to assign the clip to. In our experiments, the first choice is frequently correct. If the user decides that no cluster is appropriate, a new cluster is created. This approach removes the need for manually determining the clusters at any point during controller synthesis. Clusters are created only when the user-supplied samples are sufficiently different from all existing clusters.

The cluster is then updated with the new clip, and a new $(\mathbf{u}, \mathbf{w})$ for that clip is added to the cluster's RBF. The active learning process is repeated by selecting a new candidate tasks. The active learning process can have many termination criteria, including the time spent in the lab, number of samples allotted to the controller, as well as the overall controller quality and controllability. The user can estimate the measure the quality and controllability of the current controller by evaluating the quality of the candidate tasks: if all "poor-performing" candidates appear to be of high-quality, the controller has probably reached the point where no additional samples can significantly improve the quality and controller coverage.

5 Results

We have constructed three examples to demonstrate our framework. The controllers are demonstrated in the accompanying video. Each of these controllers was constructed interactively in a short time in the motion capture lab. We capture and synthesize motions at between 20 and 25 frames per second.

The computation time required is negligible: each active learning step took between 5 and 30 seconds, depending on the size of the controller involved. The majority of the time was spent in capture, especially since it normally takes two or three tries to perform a motion with the correct start state that matches the task.

Our first example is a parameterized walk. The control space $\mathbb{U}$ is three-dimensional, controlling stride length (the distance between the heels at maximum spacing, ranging from $0.2-0.9$ meters), speed (ranging from $15-25$ centimeters per frame), and turning angle (change in horizontal orientation per stride, ranging from $-\pi/4$ to $\pi/4$ radians). There is one task error metric for each of these features (e.g., stride length), measured as the the squared distance between the desired feature and the maximum feature in the motion. In order to determine strides, the metric assumes that a sample contains a complete walk cycle beginning with the right foot forward. This task allows motions to start from the middle.

The controller for this task produced only 1 cluster, using 12 examples totaling 843 frames, and 10 pseudoexamples. We limited lab time to roughly half an hour, including all active learning and user interaction. This controller achieved 89% coverage (discussed in section 6).

Our second example combines a parameterized walk with projectile dodging. This example has two tasks. When there is nothing to dodge, the control space is one-dimensional, controlling turning angle. When walking and dodging, the control space is four-dimensional, controlling turning angle as well as the incoming height, incoming angle, and distance in front of the character of the incoming projectile.

The walk turning parameter as a ratio of radians per meter traveled, ranging from $-\pi/3$ to $\pi/3$. The task metric measures the absolute value of the difference between this ratio and a the target value. The walk task allows motions to start from the middle.

The parameters of the incoming projectile are specified relative to the character's local coordinate system. They are the incoming height, from $0.25-2.0$ meters, the distance in front of the character to be aimed at, from $0.25-1.0$ meters, and the incoming angle, from $-\pi/2$ to $\pi/2$ radians. These parameters specify a trajectory for the projectile. The error metric is the inverse of the distance between the projectile and any point on the character. The dodge task also includes the same turning control parameter as used for walking, with the same error metric as well.

The synthesized controller contains 10 clusters, using total of 30 examples totaling 1121 frames, and 19 pseudoexamples. We limited lab time to roughly an hour, and achieved 76% coverage of the dodge task.

Our third example is catching a ball. This example has two tasks. When not catching, there is a zero-dimensional stand task. When catching, the control space is three-dimensional, controlling the incoming position on the plane in front of the character as well as

speed.

The parameters of the incoming ball are specified relative to the character's local coordinate system. They are the incoming height, from 0.5 to 2.0 meters, the distance to the right of the character, from -2.0 to 2.0 meters, and the incoming speed, from 0.1 to 0.2 meters per frame. These parameters specify a trajectory for the ball. The task error metric is the squared distance between the ball and the character's right hand.

For this example, we removed rotation invariance from all quantities, in order to build a model in which the character is always facing a specific direction while waiting for the next ball. We have also allowed the user to load in several motions of diving to the side in place of performing these motions.

Also, in this example, we use the IK controller so that we have can have greater reachability of the resulting motion. The resulting controller used 12 clusters, using 33 examples totaling 1826 frames, and 23 pseudoexamples. The data, not including the diving motions[4], was captured in the lab session limited to roughly an hour. We achieved 57% coverage of the catch task, normalized by the manual controllers discussed in section 6.

It is worth noting that we also tried creating the controller with a simpler cluster model that did not include IK, and found that the performance was significantly poorer, due to the nonlinearity of the underlying control space. In general, the more powerful the controller model, the more active learning can take advantage of it and require less samples to produce an effective controller.

6 Evaluation

We have performed a numerical evaluation of the active learning method proposed using the catching controller. We compare the active learning approach with manual sample selection. We have had four animation researchers (two associated with this project and two not) plan a motion capture session for a catching controller like the one in our results. They were asked to define the blend-space clusters, and determine a compact number of samples to best cover the controller space. All manual designs required more samples than the learned controller, but produced controllers that, visually and quantitatively, were significantly inferior to our learned controller. In Table 1, we show the number of samples and the percentage of input space coverage of each controller. For this comparison, we only consider the catch task, ignoring the stand task.

The coverage is defined as the percentage of inputs for which the controller successfully matches the start state and completes the task (for catching the ball, this means the hand is within a small distance of the ball). The percentage is computed by randomly sampling state and task, similar to the scheme in Section 4.1. For this comparison, we look at the percentage of samples controllable by a given controller out of the samples controllable by any of the controllers, to remove samples physically impossible to catch. The manual controller with roughly 80% more samples produced significantly less coverage, while the controller with 20% more samples covers roughly half the space of learned controller. The total mocap studio time was roughly the same. Of course, coverage is not a perfect measure of a controller, since it does not measure realism of the motion.

The difference in the quality of controllers is also apparent in the accompanying video: with the manual controllers, the balls are often caught in very strange and unrealistic ways. Of course, the evaluation depends on the skill level of people who chose the motions,

[4]The dive catch required significant motion cleanup and was also painful to perform. In order to facilitate prototyping and testing, we captured several representative dives in advance. Then, when the active learning system requested a dive, one of these pre-captured motions was provided (if appropriate) instead of performing a new dive. However, this meant that the controller is limited to the dives in this example set, and cannot, e.g., dive while running, since this was not included in the examples.

Method	Samples	Coverage
Active Learning	30	57%
Manual 1	36	38%
Manual 2	44	56%
Manual 3	54	48%
Manual 4	41	48%

Table 1: Comparison of active learning and manual methods of sample selection. Active learning captured less samples during a one-hour period, but achieved better coverage of the control space than manually-planned motions.

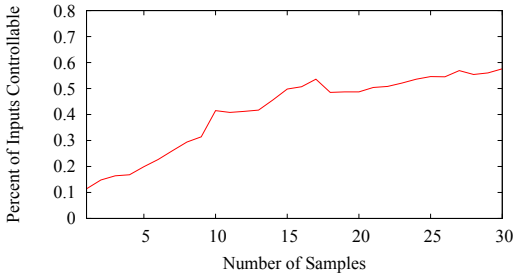

Figure 6: This graph shows the percentage of inputs controllable as motion samples are added to the controller during a capture session. Coverage may decrease when a new cluster is introduced, because at that point, the controller may choose to use this new cluster because its start state is a better match than an existing one. Coverage decreases temporarily until additional samples in the new cluster improve the input coverage.

but we believe that it is nonetheless indicative of the difficulty of the problem and the attractiveness of the active learning approach. We also show a chart demonstrating the improvement of the controller as each sample is added in Figure 6. We use the same measure of coverage as above.

7 Discussion and Future Work

We have introduced an active learning framework for creating real-time motion controllers. By adaptively determining which motions to add to the model, the system creates finely-controllable motion models with a reduced number of data clips and little time spent in the motion capture studio. In addition, by always presenting the worst-performing state samples, the user has a continuous gauge of the quality of the resulting controller. The active learning framework both automatically determines the parameters of each individual cluster and determines the necessary number and relationship between different clusters, dynamically determining the controller structure as more samples appear.

Although our system focuses on one specific model for motion synthesis, we believe that the general approach of adaptive model-building will be useful for many types of animation models — any situation in which minimizing the number of motion samples is important can potentially benefit from active learning. We have designed the system to be fast and flexible, so that relatively little time is spent waiting for the next candidate to be selected. Hence, we employed a number of heuristic, incremental learning steps, and our models are not generally "optimal" in a global sense.

Our system does not make formal guarantees of being able to perform every task from every start state. Some of these tasks may be impossible (e.g., starting a new task from midair); others may not have been captured in the limited time available in a motion capture session. The failure modes of the controller during synthesis include not having a cluster with the appropriate start state, not determining the best cluster to use, and not determining the best weights within a cluster. There is a tradeoff between lab time, number of samples, and coverage that the user must evaluate. In our examples we show that it is possible to get a high amount of coverage with a low number of samples and short lab time.

Some difficulties arise during the capture process. It is necessary for the user to look at the screen as well as mentally project their

motion onto the on-screen visualization. We believe that these difficulties can be overcome through the use of an augmented or virtual reality system.

We have no guarantee of scalability, but we believe that this system will scale well to handle many different tasks performed sequentially, creating highly flexible characters. However, our controller examples do not fully test the effectiveness of our approach in very high dimensions, and this work certainly does not solve the fundamental problem of "curse of dimensionality" for data-driven controllers. We believe that active learning will generalize to controllers with higher dimensions. However, for controllers with 10 or more task inputs together with 37 DOF characters state space, even though active learning will drastically reduce the number of samples, the number of required samples would still be impractical. In very high dimensions, it becomes an imperative to use sophisticated motion models that accurately represent nonlinear dynamics, since the expressive power of such models greatly reduces the number of required samples.

Acknowledgments

The authors would like to thank Gary Yngve for his help with the video, and the anonymous reviewers for their comments. This work was supported by the UW Animation Research Labs, NSF grants CCR-0092970, EIA-0321235, Electronic Arts, Sony, Microsoft Research, NSERC, CFI, and the Alfred P. Sloan Foundation.

References

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2003. Motion synthesis from annotations. *ACM Transactions on Graphics 22*, 3 (July), 402–408.

ATKINSON, A. C., AND DONEV, A. N. 1992. *Optimum Experimental Designs*. Oxford University Press.

BRAND, M., AND HERTZMANN, A. 2000. Style machines. *Proceedings of SIGGRAPH 2000* (July), 183–192.

COHN, D., ATLAS, L., AND LADNER, R. 1994. Improving Generalization with Active Learning. *Machine Learning 5*, 2, 201–221.

GLEICHER, M., SHIN, H. J., KOVAR, L., AND JEPSEN, A. 2003. Snap-Together Motion: Assembling Run-Time Animations. In *Proc. I3D*.

KOVAR, L., AND GLEICHER, M. 2003. Flexible Automatic Motion Blending with Registration Curves. In *Proc. SCA*.

KOVAR, L., AND GLEICHER, M. 2004. Automated Extraction and Parameterization of Motions in Large Data Sets. *ACM Trans. on Graphics* (Aug.).

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion Graphs. *ACM Trans. on Graphics 21*, 3 (July), 473–482. (Proceedings of ACM SIGGRAPH 2002).

KWON, T., AND SHIN, S. Y. 2005. Motion Modeling for On-Line Locomotion Synthesis. In *Proc. SCA*.

LAU, M., AND KUFFNER, J. 2006. Precomputed Search Trees: Planning for Interactive Goal-Driven Animation. In *Proc. SCA*.

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

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive Control of Avatars Animated With Human Motion Data. *ACM Trans. on Graphics 21*, 3 (July), 491–500.

LEE, K. H., CHOI, M. G., AND LEE, J. 2006. Motion patches: building blocks for virtual environments annotated with motion data. *ACM Transactions on Graphics 25*, 3 (July), 898–906.

LI, Y., WANG, T., AND SHUM, H.-Y. 2002. Motion Texture: A Two-Level Statistical Model for Character Motion Synthesis. *ACM Trans. on Graphics 21*, 3 (July), 465–472.

LINDENBAUM, M., MARKOVITCH, S., AND RUSAKOV, D. 2004. Selective sampling for nearest neighbor classifiers. *Mach. Learn. 54*, 2, 125–152.

LIU, C. K., AND POPOVIĆ, Z. 2002. Synthesis of Complex Dynamic Character Motion from Simple Animations. *ACM Trans. on Graphics 21*, 3 (July), 408–416.

LIU, C. K., HERTZMANN, A., AND POPOVIĆ, Z. 2005. Learning Physics-Based Motion Style with Nonlinear Inverse Optimization. *ACM Transactions on Graphics 24*, 3 (Aug.), 1071–1081.

MUKAI, T., AND KURIYAMA, S. 2005. Geostatistical motion interpolation. *ACM Trans. on Graphics 24*, 3 (Aug.), 1062–1070.

NELDER, J. A., AND MEAD, R. 1965. A simplex method for function minimization. *Computer Journal 7*, 4, 308–313.

PARK, S. I., SHIN, H. J., KIM, T. H., AND SHIN, S. Y. 2004. On-line motion blending for real-time locomotion generation. *Comp. Anim. Virtual Worlds 15*, 125–138.

REITSMA, P. S. A., AND POLLARD, N. S. 2004. Evaluating motion graphs for character navigation. In *Proc. SCA*.

ROSE, C., COHEN, M. F., AND BODENHEIMER, B. 1998. Verbs and Adverbs: Multidimensional Motion Interpolation. *IEEE Computer Graphics & Applications 18*, 5, 32–40.

ROSE III, C. F., SLOAN, P.-P. J., AND COHEN, M. F. 2001. Artist-Directed Inverse-Kinematics Using Radial Basis Function Interpolation. *Computer Graphics Forum 20*, 3, 239–250.

SANTNER, T. J., WILLIAMS, B. J., AND NOTZ, W. I. 2003. *The Design and Analysis of Computer Experiments*. Springer.

SHIN, H. J., AND OH, H. S. 2006. Fat Graphs: Constructing an interactive character with continuous controls. In *Proc. SCA*.

SLOAN, P.-P. J., ROSE III, C. F., AND COHEN, M. F. 2001. Shape by Example. In *Proc. I3D*.

TORRESANI, L., HACKNEY, P., AND BREGLER, C. 2007. Learning Motion Style Synthesis from Perceptual Observations. In *Proc. NIPS 19*.

WILEY, D. J., AND HAHN, J. K. 1997. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Applications 17*, 6 (Nov./Dec.), 39–45.

WITKIN, A., AND POPOVIĆ, Z. 1995. Motion Warping. *Proc. SIGGRAPH 95* (Aug.), 105–108.

YAMANE, K., KUFFNER, J. J., AND HODGINS, J. K. 2004. Synthesizing animations of human manipulation tasks. *ACM Trans. on Graphics 23*, 3 (Aug.), 532–539.

ZORDAN, V. B., MAJKOWSKA, A., CHIU, B., AND FAST, M. 2005. Dynamic Response for Motion Capture Animation. *ACM Transactions on Graphics 24*, 3 (Aug.).