Motion Fields for Interactive Character Locomotion

Yongjoon Lee[1,2*] Kevin Wampler[1†] Gilbert Bernstein[1] Jovan Popović[1,3] Zoran Popović[1]
[1]University of Washington [2]Bungie [3]Adobe Systems

Abstract

We propose a novel representation of motion data and control that enables characters with both highly agile responses to user input and natural handling of arbitrary external disturbances. The representation organizes motion data as samples in a high dimensional generalization of a vector field we call a 'motion field'. Our runtime motion synthesis mechanism freely 'flows' in the motion field and is capable of creating novel and natural motions that are highly-responsive to the real time user input, and generally not explicitly specified in the data.

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

Keywords: animation, motion representation, data-driven animation

1 Introduction

Human motion is a highly-varied and continuous phenomenon: it quickly adapts to different tasks, responds to external disturbances, and in general is capable of continuing locomotion from almost any initial state. As video games increasingly demand that characters move and behave in realistic ways, it is important to bring these properties of natural human motion into the virtual world. Unfortunately this is easier said than done. For instance, despite many advances in character animation techniques, creating highly agile and realistic interactive locomotion controllers remains a common but difficult task.

We propose a new motion representation for interactive character animation, termed a *motion field* which provides two key abilities: the ability for a user to control the character in real time and the ability to operate in the fully-continuous configuration space of the character. Although there exist techniques which allow one or the other of these abilities, it is the combination of the two which allows for highly agile controllers which can respond to user commands in a short amount of time.

More specifically, a motion field is a mapping which associates each possible configuration of a character with a *set* of motions describing how the character is able to move from their current state. In order to generate an animation we select a single motion from this set, follow it for a single frame, and repeat from the character's resulting state. The motion of the character thus 'flows' through the state space according to the integration process, similar to a particle flowing through a force field. However, instead of a single fixed flow, a motion field allows multiple possible motions at each frame. By using reinforcement learning to choose between these possibilities at runtime the direction of the flow can be altered, allowing the character to respond optimally to user commands.

Because motion fields allow a range of actions at every frame, a character can immediately respond to new user commands rather than waiting for pre-determined transition points as in a motion graph. This allows motion field-based controllers to be significantly more agile than their graph-based counterparts. By further altering this flow with other external methods, such as inverse kinematics or physical simulation, we also can directly integrate these techniques into the motion synthesis and control process. Furthermore, since our approach requires very little structure in the motion capture data that it uses, minimal effort is needed to generate a new controller. The primary contribution of this work lies in the combining of a continuous state representation with an optimal control framework. We find that this approach provides many advantages for character animation.

2 Related Work

In the past ten years, the bag-of-clips data structures such as motions graphs have emerged as primary sources of realistic character controllers [Lee et al. 2002; Arikan and Forsyth 2002; Kovar et al. 2002]. These structures are inherently discrete with coarse transitioning abilities that provide great computational advantages. Unfortunately, this discretization also obscures continuous properties of motion. First, it is difficult to create graphs which allow very quick responses to changes of direction or unexpected disturbances since a change to the motion can only happen when a new edge is reached [Treuille et al. 2007; McCann and Pollard 2007]. Second, because the motions are restricted to the clips which constitute the graph it is difficult to couple these methods to physical simulators and other techniques which perturb the state away from states representable by the graph. More generally, it is very hard to use a graph-based controller when the character starts from an arbitrary state configuration [Zordan et al. 2005].

Although a number of methods have been proposed to alleviate some of the representational weaknesses of pure graph-based controllers, including parameterized motion graphs [Shin and Oh 2006; Heck and Gleicher 2007], increasing the numbers of possible transitions [Arikan et al. 2005; Yin et al. 2005; Zhao and Safonova 2008] and splicing rag doll dynamics in the graph structure [Zordan et al. 2005], the fundamental issue remains: unless the representation prescribes motion at every continuous state in a way that is controllable in real time, the movement of characters will remain restricted. Hence, even when the method anticipates some user inputs [McCann and Pollard 2007], the character may react too slowly, or transition too abruptly because there is no shorter path in the graph. Similarly, when methods anticipate some types of upper-body pushes [Yin et al. 2005; Arikan et al. 2005], the character may not react at all to hand pulls or lower-body pushes.

Another group of methods use nonparametric models to learn the dynamics of character motion in a fully continuous space [Wang et al. 2008; Ye and Liu 2010; Chai and Hodgins 2005]. These techniques are generally able to synthesize starting from any initial

*e-mail: yjlee@bungie.com
†e-mail: wampler@cs.washington.edu

ACM Reference Format
Lee, Y., Wampler, K., Bernstein, G., Popović, J., Popović, Z. 2010. Motion Fields for Interactive Character Animation. *ACM Trans. Graph. 29*, 6, Article 138 (December 2010), 8 pages. DOI = 10.1145/1866158.1866160 http://doi.acm.org/10.1145/1866158.1866160.

state, and lend themselves well to applying physical disturbances [Ye and Liu 2010] and estimating a character's pose from incomplete data [Chai and Hodgins 2005]. These models are used to estimate a single 'most likely' motion for the character to take at each possible state. This precludes the ability to optimally control the character. The primary difference between our work and these is that instead of building a model of the most probable single motion, we attempt to model the set of possible motions at each character state, and only select the single motion to use at runtime by using principles from optimal control theory. This allows us to interactively control the character while enjoying the benefits of a fully continuous state space. Our work combines the concepts of near-optimal character control present in graph-based methods with those of nonparametric motion estimation techniques.

Although our controllers are kinematic, dynamic controllers have been extensively explored as an alternative method of character animation. In principle, such controllers offer the best possibility for highly realistic interactive character animation. However, high-fidelity physically based character animation is harder to attain because physics alone does not tell us about the muscle forces needed to propel the characters. Despite a broad repertoire of demonstrated skills [Hodgins et al. 1995; Hodgins and Pollard 1997; Wooten and Hodgins 2000; Faloutsos et al. 2001; Yin et al. 2007; Coros et al. 2008b], nonparametric modeling of coarse-scale dynamics [Coros et al. 2009; Coros et al. 2008a], and use of motion capture [Laszlo et al. 1996; Sok et al. 2007; da Silva et al. 2008; Muico et al. 2009], agile, lifelike, fully-dynamic characters remain an open challenge.

3 Motion Fields

Interactive applications such as video games require characters that can react quickly to user commands and unexpected disturbances, all while maintaining believability in the generated animation. An ideal approach would fully model the complete space of natural human motion, describing every conceivable way that a character can move from a given state. Rather than confine motion to canned motion clips and transitions, such model would enable much greater flexibility and agility of motion through the continuous space of motion.

Although it is infeasible to completely model the entire space of natural character motion, we can use motion capture data as a local approximation. We propose a structure called a *motion field* that finds and uses motion capture data similar to the character's current motion at any point. By consulting similar motions to determine which future behaviors are plausible, we ensure that our synthesized animation remains natural: similar, but rarely identical to the motion capture data. This frees the character from simply replaying the motion data, allowing it to move freely through the general vicinity of the data. Furthermore, because there are always multiple motion data to consult, the character constantly has a variety of ways to make quick changes in motion.

3.1 Preliminary Definitions

Motion States We represent the states in which a character might be configured by the pose and the velocity of all of each of a character's joints. A *pose* $x = (x_{\text{root}}, p_0, p_1, \ldots, p_n)$ consists of a 3d root position vector x_{root}, a root orientation quaternion p_0 and joint orientation quaternions $p_1, \ldots p_n$. The root point is located at the pelvis. A *velocity* $v = (v_{\text{root}}, q_0, q_1, \ldots, q_n)$ consists of a 3d root displacement vector v_{root}, root displacement quaternion q_0, and joint displacement quaternions $q_1, \ldots, q_n$, all found via finite differences. Given two poses x and x', we can compute this finite difference as:

$$v = x' \ominus x = \left(x'_{root} - x_{root}, p'_0 p^{-1}, p'_1 p_1^{-1}, \ldots, p'_n p_n^{-1}\right)$$

By inverting the above difference, we can add a velocity v to a pose x to get a new displaced pose $x' = x \oplus v$. We can also interpolate multiple poses or velocities together ($\sum_{i=1}^{k} w_i x_i$ or $\sum_{i=1}^{k} w_i v_i$) using linear interpolation of vectors and unit quaternion interpolation[Park et al. 2002] on the respective components of a pose or velocity. We use $\ominus$ and $\oplus$ in analogy to vector addition and subtraction in Cartesian spaces, but with circles to remind the reader that we are working mostly with quaternions.

Finally, we define a *motion state* $m = (x, v)$ as a pose and an associated velocity, computed from a pair of successive poses x and x' with $m = (x, v) = (x, x' \ominus x)$. The set of all possible motion states forms a high dimensional continuous space, where every point represents the state of our character at a single instant in time. A path or trajectory through this space represents a continuous motion of our character. When discussing dynamic systems, this space is usually called the phase space. However, because our motion synthesis is kinematic, we use the phrase *motion space* instead to avoid confusion.

Motion Database Our approach takes as input a set of motion capture data and constructs a set of motion states $\{m_i\}_{i=1}^{n}$ termed a *motion database*. Each state m_i in this database is constructed from a pair of successive frames x_i and x_{i+1} by the aforementioned method of $m_i = (x_i, v_i) = (x_i, x_{i+1} \ominus x_i)$. We also compute and store the velocity of the *next* pair of frames, computed by $y_i = x_{i+2} \ominus x_{i+1}$. Generally, motions states, poses and velocities from the database will be subscripted (e.g. m_i, x_i, v_i, and y_i), while arbitrary states, poses and velocities appear without subscripts.

Similarity and neighborhoods Central to our definition of a motion field is the notion of the *similarity* between motion states. Given a motion state m, we compute a *neighborhood* $\mathcal{N}(m) = \{m_i\}_{i=1}^{k}$ of the k most similar motion states via a k-nearest neighbor query over the database [Mount and Arya 1997]. In our tests we use $k = 15$. We calculate the (dis-)similarity by:

$$d(m, m') = \sqrt{\begin{array}{rl} \beta_{\text{root}} ||v_{\text{root}} - v'_{\text{root}}||^2 & + \\ \beta_0 ||q_0(\hat{u}) - q'_0(\hat{u})||^2 & + \\ \sum_{i=1}^{n} \beta_i ||p_i(\hat{u}) - p'_i(\hat{u})||^2 & + \\ \sum_{i=1}^{n} \beta_i ||(q_i p_i)(\hat{u}) - (q'_i p'_i)(\hat{u})||^2 & + \end{array}} \quad (1)$$

where $\hat{u}$ is some arbitrary unit length vector; $p(\hat{u})$ means the rotation of $\hat{u}$ by p; and the weights $\beta_{\text{root}}, \beta_0, \beta_1, \ldots, \beta_n$ are tunable scalar parameters. In our experiments, we set β_i as bone lengths of the body at the joint i in meters, β_{root} and β_0 are set to 0.5. Intuitively, setting β_i to the length of its associated bone de-emphasizes the impact of small bones such as the fingers. Note that we factor out root world position and root yaw orientation (but not their respective velocities).

Similarity Weights Since we allow the character to deviate from motion states in the database, we frequently have to interpolate data from our neighborhood $\mathcal{N}(m)$. We call the weights $[w_0, \ldots, w_k]$ used for such interpolation *similarity weights* since they measure similarity to the current state m:

$$w_i = \frac{1}{\eta} \frac{1}{d(m, m_i)^2} \quad (2)$$

where m_i is the ith neighbor of m and $\eta = \sum_i \frac{1}{d(m,m_i)^2}$ is a normalization factor to ensure the weights sum to 1.

3.2 Motion Synthesis

Actions The value of a *motion field* $\mathcal{A}$ at a motion state m is a set of control actions $\mathcal{A}(m)$ determining which states the character can transition to in a single frame's time. Each of these actions $a \in \mathcal{A}(m)$ specifies a convex combination of neighbors $a = [a_1, \ldots, a_k]$ (with $\sum a_i = 1$ and $a_i > 0$). Given one particular action $a \in \mathcal{A}(m)$, we then determine the next state m' using a transition or *integration function* $m' = (x', v') = \mathcal{I}(x, v, a) = \mathcal{I}(m, a)$ Letting i range over the neighborhood $\mathcal{N}(m)$, we use the function

$$\mathcal{I}(m,a) \quad = \quad \left(x \oplus \sum a_i v_i, \sum a_i y_i\right) \tag{3}$$

Unfortunately, this function frequently causes to our character's state to drift off into regions where we have little data about how the character should move, leading to unrealistic motions. To correct for this problem, we use a small drift correction term that constantly tugs our character towards the closest known motion state $\bar{m} = (\bar{x}, \bar{v})$ in the database. The strength of this tug is controlled by a parameter $\delta = 0.1$

$$\mathcal{I}(m,a) \quad = \quad \left(x \oplus v', y'\right) \tag{4}$$

$$v' \quad = \quad (1-\delta)\left(\sum_{i=1}^{k} a_i v_i\right) \oplus \delta((\bar{x} \oplus \bar{v}) \ominus x) \tag{5}$$

$$y' \quad = \quad (1-\delta)\left(\sum_{i=1}^{k} a_i y_i\right) \oplus \delta \bar{y} \tag{6}$$

Passive Action Selection Given a choice of action we now know how to generate motion, but which action should we pick? This question is primarily the subject of section 4. However, we can quickly implement a simple solution using the similarity weights (eq. 2) as our choice of action. This choice results in the character meandering through the data, generating streams of realistic (albeit undirected) human motion.

Foot-Skate Cleanup The result of motion synthesis might contain foot-skating artifacts. We remove these artifacts by applying inverse kinematics on the contact foot. To do this, we first annotate foot contacts in the motion data. For every motion state m_i in the database, we store whether the left foot is in contact $l_{\text{contact}}(m_i) = 1$ or not $l_{\text{contact}}(m_i) = 0$, and likewise for the right foot $r_{\text{contact}}(m_i)$. Then at runtime, we determine whether or not a foot is in contact at an arbitrary motion state m by taking a weighted vote across the neighborhood $\mathcal{N}(m)$.

$$l_{\text{contact}}(m) = \sum_{m_i \in \mathcal{N}(m)} w_i l_{\text{contact}}(m_i) \tag{7}$$

(w_i are the similarity weights of the neighbors: Equation (2)) If $l_{\text{contact}}(m) \geq 0.5$ we say the left foot is in contact. When the foot leaves contact, we blend out of the inverse kinematics solution that holds the foot in place during contact, within 0.2 seconds.

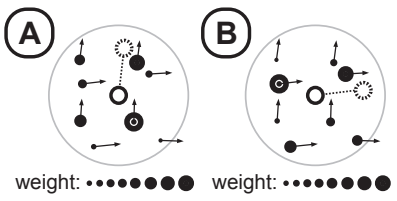


Figure 1: Control using action weights. *By reweighting the neighbors (black dots) of our current state (white dot), we can control motion synthesis to direct our character towards different next states (dashed dots).*

4 Control

As described in section 3, at each possible state of the character a motion field there is a *set* of actions which the character can choose from in order to determine their motion over the next frame. In general, which particular action from this set it is best to choose depends on the user's current commands. Deciding on which action to choose in each state in response to a user's commands is thus key in enabling real time interactive locomotion controllers.

4.1 Markov Decision Processes Control

A Markov decision process is a mathematical structure formalizing the concept of making decisions in light of both their immediate and long-term results. An MDP consists of four parts: (1) a state space, (2) actions to perform at each state, (3) a means of determining the state transition produced by an action, and (4) rewards for occupying desired states and performing desired actions. By expressing character animation tasks in this framework, we make our characters aware of long term consequences of their actions. This is useful even in graph-based controllers, but vital for motion field controllers because we are acting every frame rather than every clip. For further background on MDP-based control see [Sutton and Barto 1998], or [Treuille et al. 2007] and [Lo and Zwicker 2008] for their use in graph-based locomotion controllers.

States Simply representing the state of a character as a motion state m is insufficient for interactive control, because we must also represent how well the character is achieving its user-specified task. We therefore add a vector of *task parameters* θ_T to keep track of how well the task is being performed, forming joint *task states* $s = (m, \theta_T)$. For instance in our direction following task θ_T records a single number: the angular deviation from the desired heading. By altering this value, the user controls the character's direction.

Actions At each task state $s = (m, \theta_T)$ a character in a motion field has a set of actions $\mathcal{A}(m)$ to choose from in order to determine how they will move over the next frame (section 3). There are infinitely many different actions in $\mathcal{A}(m)$, but many of the techniques used to solve MDPs require a *finite* set of actions at each state. In order to satisfy this requirement for our MDP controller, we sample a finite set of actions $\mathcal{A}(s)$ from $\mathcal{A}(m)$. Given a motion state m, we generate k actions by modifying the similarity weights (Equation (2)). Each action is designed to prefer one neighbor over the others.

$$\{\frac{a_i}{\|a_i\|} | a_i = (w_0, \cdots, w_{i-1}, 1, w_{i+1}, \cdots, w_{k-1})\} \tag{8}$$

In words, to derive action a_i simply set w_i to 1 and renormalize. This scheme samples actions which are not too different from the passive action at m so as to as to avoid jerkiness in the motion, while giving the character enough flexibility to move towards nearby motion states.

Transitions Additionally, we must extend the definition of the integration function $\mathcal{I}$ (Equation (4)) to address task parameters: $\mathcal{I}_s(s, a) = \mathcal{I}_s(m, \theta_T, a) = (\mathcal{I}(m, a), \theta'_T)$. How to update task parameters is normally obvious. For instance in the direction following task, where θ_T is the characters deviation from the desired direction, we simply adjust θ_T by the angle the character turned.

Rewards In order to make our character perform the desired task we offer rewards. Formally, a *reward function* specifies a real number $R(s, a)$ quantifying the reward received for performing the action a at state s. For instance, in our direction following task we

give high a high reward $R(s,a)$ for maintaining a small deviation from the desired heading and a lower reward for large deviations. See section Section 6 for the specific task parameters and reward functions we use in our demos.

4.2 Reinforcement Learning

The goal of reinforcement learning is to find "the best" rule or *policy* for choosing which action to perform at any given state. A naïve approach to this problem would be to pick the action which yields the largest immediate reward: the *greedy policy*.

$$\pi_G(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} R(s,a) \tag{9}$$

Although simple, this policy is myopic, ignoring the future ramifications of each action choice. We already know that greedy graph-based controllers perform poorly [Treuille et al. 2007]. Motion fields are even worse. Even for the simple task of changing direction, we need a much longer horizon than $\frac{1}{30}$th of a second to anticipate and execute a turn.

Somehow, we need to consider the affect of the current action choice on the character's ability to accrue future rewards. A *lookahead policy* π_L does just this by considering the cumulative reward over future task states:

$$\pi_L(s) = \underset{a \in \mathcal{A}(m)}{\operatorname{argmax}} \left[R(s,a) + \max_{\{a_t\}} \sum_{t=1}^{\infty} \gamma^t R(s_t, a_t) \right] \tag{10}$$

with $s_1 = \mathcal{I}_s(s,a)$ and $s_t = \mathcal{I}_s(s_{t-1}, a_{t-1})$. γ is called the *discount factor* and controls how much the character focuses on short term ($\gamma \to 0$) versus long term ($\gamma \to 1$) reward.

As written, computing the lookahead policy involves solving for not only the optimal next action, but also an *infinite* sequence of optimal future actions. Despite this apparent impracticality, a standard trick allows us to efficiently solve for the correct next action. The trick begins by defining a *value funciton* $V(s)$, a scalar-valued function representing the expected cumulative future reward received for acting optimally starting from task state s.

$$V(s) = \max_{a \in \mathcal{A}(m)} \sum_{t=0}^{\infty} \gamma R(s_t, a_t) \tag{11}$$

We will describe shortly how we represent and precompute the value function, but for the moment notice that we can now rewrite equation 10 by replacing the infinite future search with a value function lookup:

$$\pi_L(s) = \underset{a \in \mathcal{A}(m)}{\operatorname{argmax}} \left[R(s,a) + V(\mathcal{I}_s(s,a)) \right] \tag{12}$$

Now the lookahead policy is only marginally more expensive to compute than the greedy policy.

4.2.1 Value Function Representation and Learning

Since there are infinitely many possible task states, we cannot represent the value function exactly. Instead we approximate it by storing values at a finite number of task states s_i and interpolating to estimate the value at other points (Figure 2). We choose these task state samples by taking the Cartesian product of the database motion states m_i and a uniform grid sampling across the problem's task parameters. See Section 6 for details of the sampling. This sampling gives us high resolution near the motion database states,

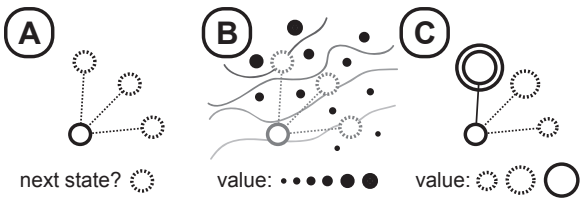


Figure 2: Action search using a value function. *(A) At every state, we have several possible actions (dashed lines) and their next states (dashed circles). (B) We interpolate the value function stored at database states (black points) to determine the value of each next state. (C) We select the highest value action to perform.*

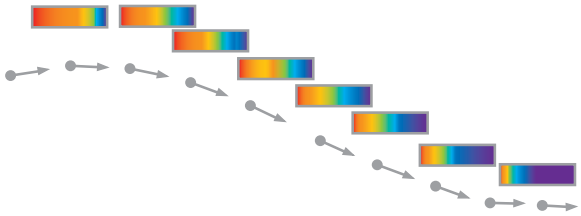


Figure 3: Uncompressed value function. *The value functions V_{m_i} are stored at every motion state m_i.*

which is where the character generally stays. In order to calculate the value $V(s)$ of a task state not in the database, we interpolate over neighboring motion states using the similarity weights and over the task parameters multilinearly.

Given an MDP derived from a motion field and a task specification, we solve for an approximate value function in this form using *fitted value iteration* [Ernst et al. 2005]. Fitted value iteration operates by first noting that equation 12 can be used to write the definition of the value function in a recursive form. We express the value at a task state sample s_i recursively in terms of the value at other task state samples:

$$V(s_i) = R(s_i, \pi_L(s_i)) + V(\mathcal{I}_s(s_i, \pi_L(s_i))) \tag{13}$$

where $\pi_L(s_i)$ is as defined in equation 12 and $V(\mathcal{I}_s(s_i,a))$ is computed via interpolation. We can solve for $V(s_i)$ at each sample state by iteratively applying equations 12 and 13. We begin with an all-zero value function $V_0(s_i) = 0$ for each sample s_i. Then at each s_i equation 12 is used to compute $\pi_L(s)$ after which we use equation 13 to determine an updated value at s_i. After all the s_i samples have been processed in this manner, we have an updated approximation of the value function. We repeat this process until convergence and use the last iteration as the final value function.

4.2.2 Temporal Value Function Compression

Unlike graph-based approaches, motion fields let characters be in constant transition between many sources of data. Consequently, we need access to the value function at all motion states, rather than only at transitions between clips. This fact leads to a large memory footprint relative to graphs. We offset this weakness with compression.

For the purpose of compression, we want to think of our value function as a collection of value functions of task parameters. Without compression, we store one of these value sub-functions $V_{m_i}(\theta_T) = V(m_i, \theta_T)$ at every database motion state m_i (see Figure 3). Here, we observe that our motion states were originally

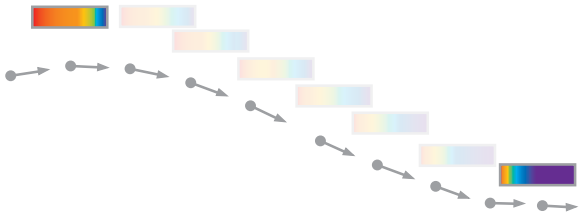


Figure 4: Value function with temporal compression. *The value functions at intermediate motion states are interpolated by the neighboring 'anchor' motion states that have explicitly stored value functions.*

obtained from continuous streams of motion data. At 30Hz temporally adjacent motion states and their value functions are frequently similar; we expect that V_{m_t} changes smoothly over "consecutive" motion states m_t relative to the original clip time. Exploiting this idea, we only store value functions at every N-th motion state, and interpolate the value functions for other database motion states (See Figure 4). We call these database states storing value functions 'anchor' motion states. We compute the value function at the ith motion state between two anchors m_0 and m_N as

$$V_{m_i}(\theta_T) = \frac{N-i}{N} V_{m_0}(\theta_T) + \frac{i}{N} V_{m_N}(\theta_T) \tag{14}$$

We can learn a temporally compressed value function with a trivially modified form of the algorithm given in section 4.2.1. Instead of iterating over all task states, we only iterate over those states associated with anchor motion states.

This technique allows the tradeoff between the agility of a motion field-based controller and its memory requirements. Performing little or no temporal interpolation yields very agile controllers at the cost of additional memory, while controllers with significant temporal compression tend to be less agile. In our experiments we found that motion field controllers with temporal compression are approximately as agile as graph-based controllers when restricted to use an equivalent amount of memory, and significantly more agile when using moderately more memory (see Section 6).

5 Response to Perturbation

Because each motion state consists of a pose and a velocity, the space of motion states the character can occupy is identical to the phase space of the character treated as a dynamic system. This identification allows us to easily apply arbitrary physical or non-physical perturbations and adjustments. For example, we can incorporate a dynamics engine or inverse kinematics. Furthermore, we do not have to rely on target poses or trajectory tracking in order to define a recovery motion. Recovery occurs automatically and simultaneously with the perturbation as a by-product of our motion synthesis and control algorithm.

To illustrate the integration of perturbations into our synthesis algorithm, we describe a simple technique which provides pseudo-physical interaction with the ability to apply forces to any part of the body. This approach blends the results obtained by a physical simulator with the results of our motion synthesis technique. This blend occurs over a window of k update steps, beginning when a set of forces is first applied. (We set $k = 20$ i.e. $2/3$ of a second in our implementation.) During this blending phase, we use a modified integration formula (Equation (4)):

$$\mathcal{I}_D(x, v, a) = \frac{i}{k}\mathcal{I}(x, v, a) + \frac{k-i}{k} D(x, 0, i) \tag{15}$$

where $D(x, 0, i)$ is the state after i steps of dynamic simulation starting at pose x with initial velocity 0. $\mathcal{I}_D$ can be used in conjunction with both passive and controlled motion fields.

In our implementation we use Open Dynamics Engine (ODE, [Smith 2010]) to calculate $D(x, 0, i)$. At each of the next k frames after a force is applied we set the state of the character in ODE to x with zero initial velocity. We then apply any perturbation forces and simulate the resulting dynamics for i frames with gravity disabled. This setup (with zero initial velocity and no gravity) has the useful property that in the absence of any perturbation forces the character's pose x goes unaltered. In order to better mimic the way in which an actual person would "tip" about their feet when pushed we also pin any contacting feet to the ground with ball joints during this simulation. When a new force is applied during an ongoing blend, we simply terminate the old blending process early and begin again with the new force. As a result, velocities do not transfer correctly between multiple quick pushes. However, in many cases this is not visually apparent, even when multiple large forces are applied in quick succession. In addition, because $\mathcal{I}(x, v, a)$ does not handle velocity in the same manner as a dynamical system, our perturbation method is not physically accurate, but rather a heuristic which gives plausible-looking results. Nevertheless, it is useful as an illustration of how perturbations can be easily integrated into the synthesis process.

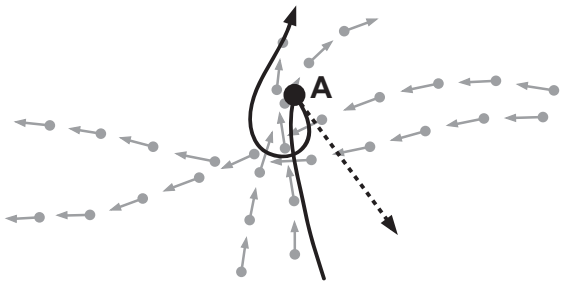


Figure 5: Responding to external perturbation. *When external force (dashed vector) is applied at state* ***A*** *causing a discontinuous change of behavior, the system can immediately find a new path around the motion fields to naturally recover from the impact.*

6 Experiments

This section presents analysis on two important properties of motion fields – agility in responding to user directive changes and ability to respond to dynamic perturbation.

6.1 Agile Responses to User Control

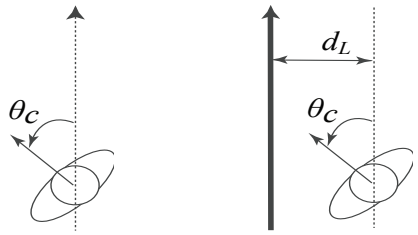
6.1.1 Experiment Setup

We created value functions for two example tasks: following an arbitrary user-specified direction and staying on a straight line while following the user direction. (See Figure 6). The reward $R_{\text{direction}}$ for the direction task and the reward R_{line} for the line following task are respectively defined as

$$R_{\text{direction}}(m, \theta_c, a) = -|\theta_c| \tag{16}$$

$$R_{\text{line}}(m, \theta_c, d_L, a) = -|\theta_c| - 0.05|d_L|. \tag{17}$$

Motion Data Setup We used 142 seconds of motion data containing leisurely-paced locomotion and quick responses to direction



(a) Direction (b) Line Following

Figure 6: ***Task Parameters.*** *For the direction following task (a), the difference in angle θ_c of the desired direction from the character facing direction is used. For the line following task (b), distance to the desired line d_L is also considered with θ_c.*

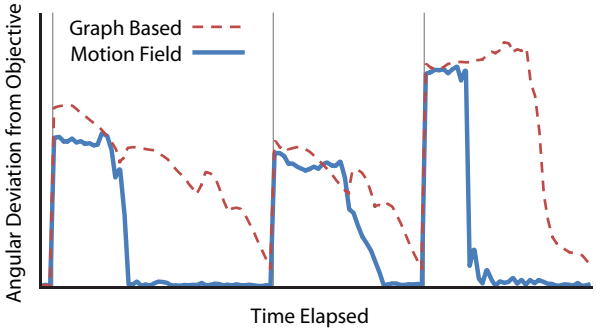


Figure 7: ***Response Time.*** *Direction adjustment over time with three consecutive direction changes within 4.23 seconds. The motion field control adjusts in a significantly shorter time period than the graph-based control.*

and line changes. We selected the source motion data with minimum care except to roughly cover the space of possible motion. The only manual pre-processing was foot contact annotation.

Value Function Computation We use value iteration to calculate the value function. For the direction task, we store values for 18 uniformly sampled directions θ_c. For the line following task, we take a Cartesian cross product sampling between 18 uniform θ_c samples and 13 uniform d_L samples spanning -2.0m to 2.0m. We set the discount factor to $\gamma = 0.99$. For each task, we also created 'temporally compressed' versions of the value functions, where we set $N = 1, 10, 20, 30$ in equation 14. Using value iteration to solve for the value function takes within 2 minutes if there is sufficient memory to cache the actions and transitions, and 3 hours otherwise. Distributing the value iteration updates over compute clusters can easily address these time and memory burdens.

6.1.2 Response Timing Analysis

Graph-Based Control vs Motion Field Control In order to compare how quickly the character can adjust to abruptly changing directives, we created a graph-based task controller [Lee et al. 2009] using the same motion data, tasks and reward functions. In order to maximize agility, we allowed a wide range of up to ± 45 degrees of directional warping on clips, and gave minimal importance to the physicality cost. (See Lee et al. [2009] for details.) Figure 7 shows typical responses to changing user directions. For both tasks, the motion fields demonstrated much quicker convergence to new goals, as shown in the accompanying video and the Table 1.

Effect of Value Function Compression We recorded response times using the compressed value functions on uniformly sampled user direction changes. With increasing degree of compression the system still reliably achieved user goals, but gradually lost agility in the initial response (See Table 1). We ran a similar experiment for the line following task. We uniformly sampled user direction changes as well as line displacement changes. Then we measured the time until the character converged to within 5 degrees from the desired direction and 0.1 meters from the desired tracking line. We observed similar losses of agility (See Table 2).

Representation	Minimum	Average	Maximum
Graph-based	0.31	0.94	2.36
Motion Field	0.21	0.40	1.01
Motion Field $\times 10$	0.21	0.49	1.23
Motion Field $\times 20$	0.25	0.66	1.19
Motion Field $\times 30$	0.38	0.78	1.93

Table 1: *Response times in seconds for the direction task until converging within 5 degrees of desired direction. Motion Field $\times 10$ denotes ten-fold temporally compressed value function on the motion field ($N = 10$). Motion Field $\times 20$ and $\times 30$ are defined similarly. A motion field with thirty-fold temporal compression has agility similar to graph-based control, while even a twenty-fold compression is significantly more responsive than the graph-based alternative.*

Representation	Minimum	Average	Maximum
Graph-based	0.47	1.30	2.19
Motion Field	0.30	0.57	1.26
Motion Field $\times 10$	0.30	0.68	1.42
Motion Field $\times 20$	0.42	0.91	2.51
Motion Field $\times 30$	0.55	1.45	3.56

Table 2: *Response times in seconds for the line following task until converging within 5 degrees of desired direction and 0.1 meters from the desired tracking line. In this two-dimensional control example, the twenty-fold compression is still more responsive than the graph-based control.*

6.1.3 Storage Requirement and Computational Load

The uncompressed value function for the direction-following task is stored in 320KB. The compressed value functions required 35KB, 19KB, and 13KB for 10x, 20x, and 30x cases respectively. This compares to the storage required for the graph-based method of 14KB. We believe this is reasonable and allows flexible trade off between storage and agility. For more complex tasks, the size increase of the value functions are in line with the size increased for graph-based value functions.

The approximate nearest neighborhood (ANN) [Mount and Arya 1997] queries represent most of the computational cost. The runtime performance depends on the sample action size k (Equation (8)), as we make $(k + 1)$ ANN calls to find the optimal action: one ANN call to find the neighbors of the current state, and then k more ANN calls to find the neighbors of the next states to evaluate value by interpolation. We believe localized neighborhood search as in PatchMatch [Barnes et al. 2009] can reduce the cost of the n subsequent calls, because the next states tend to be quite close to each other at 30Hz.

The same ANN overhead applies at learning time. A naive learning implementation takes hours to learn a value function for a large database or a high dimensional task. By caching the result of the ANN calls on the fixed motion samples, we can dramatically speed up learning time to just a couple minutes.

6.2 Perturbation

Using the algorithm described in section 5 we integrated pseudo-physical interaction with motion field driven synthesis, using both passive and controlled motion fields. We tested the perturbations on the following four datasets:

1. 18 walks, including sideways, backwards, and a crouch.
2. dataset 1 plus 7 walking pushes and 7 standing pushes.
3. 5 walks, 6 arm pulls on standing, 6 arm pulls on walking, 7 torso pushes on standing, and 7 torso pushes on walking.
4. 14 walks and turns.

The character responds realistically to small or moderate disturbances, even in datasets 1 and 4 which only contain non-pushed motion capture. In datasets 2 and 3 with pushed data, we observe a wider variety of realistic responses, and better handling of larger disturbances. Forces applied to different parts of the character's body generally result in appropriate reactions from the character, even in the presence of user control.

We have, however, observed some cases where forces produced unrealistic motion. This occurs when the character is pushed into a state far from data with a reasonable response. This can be addressed by including more data for pushed motion.

7 Limitations

Just as with any other data-driven method, our method is limited by the data it is given. So long as the character remains close to the data the synthesized motion appears very realistic. When the character is far from the data, realism and physical plausibility of the motion declines. Although always limited by the presence of data, we expect that the range of plausible motion can be extended by an incorporation of concepts from physical dynamics (inertia, gravity, etc.) into the integration process.

We have successfully generated controllers for two-dimensional near-optimal control problems using a moderate-sized motion database. In order for this technique to scale to much larger sets of motion data and all possible tasks, the current time and space performance of the algorithm needs to be improved. Although we have presented a technique which allows the storage requirements of our method to be reduced (section 4.2.2) at high levels of compression the controller's agility degrades to that of graph-based controllers. As we find the value functions for locomotion tasks are generally smooth both in space and time, we expect that more advanced compression techniques can effectively enable motion flows on more complicated control tasks on massive data sets. One particularly interesting possibility would be to apply a motion field-based analogue of [Lee et al. 2009] which would adaptively select a compact representation while preserving the controller's behavior.

We have chosen k-NN rather than a radius search to define $\mathcal{N}(m)$ because it leads to more predictable runtime performance. Even in cases with significant redundant data, so long as just one of the k neighbors goes in a desired direction, the optimal control-based action selection will choose it. None the less, although we have not observed it, it is possible that in highly redundant data sets $\mathcal{N}(m)$ won't provide a sufficient variety of actions. Intelligently selecting which motion states to include in the motion database is will likely be necessary to use our technique with large unprocessed motion-capture corpuses.

Because we make heavy use of k-nearest neighbors lookups and interpolation, our method is more expensive at runtime than graph-based approaches. None the less, we have found that even our unoptimized implementation runs at approximately 200 frames per second. Further efficiency improvements—such as incremental nearest neighbor searches—are an interesting avenue of research. This would allow for large crowds of characters to be animated as well as enable motion fields which have very large sets of actions at each state.

One final current limitation of motion fields lies in the lack of well-understood tools to analyze and edit them. This is in contrast to motion graphs, which can rely on an extremely well-understood set of algorithms for manipulating graphs developed over many decades. For instance, motion graphs are usually pruned to ensure that they are strongly connected; starting from any state, a character can reach any other state. Although we have not found this pruning to be necessary in our controllers, this is the sort of task for which we do not yet have good tools in the context of motion fields. We think that the development of such tools would be useful in authoring new controllers, and would potentially have applications in areas outside of character animation.

8 Conclusion

This paper introduces a new representation for character motion and control that allows realtime-controlled motion to flow through the continuous configuration space of character poses. This flow can altered in response to realtime user-supplied tasks. Due to its continuous nature, it addresses some of the key issues inherent to the discrete nature of graph-like representations, including agility and responsiveness, the ability to start from an arbitrary pose, and response to perturbations. Furthermore, the representation requires no preprocessing of data or determining where to connect clips of captured data. This makes our approach both flexible, easy to implement, and easy to use. We believe that structureless techniques such as the one we propose will provide a valuable tool in enabling the highly responsive and interactive characters required to create believable virtual characters.

Although the motion field representation can be used by itself, we think it can easily integrate with graph-based approaches. Since motion fields make very few requirements of their underlying data, they can directly augment graph-based representations. In this way, one could reap the benefits of graphs (computational efficiency, ease of analysis, etc.) when the motion can safely be restricted to lie on the graph, but retain the ability to handle cases where the motion leaves the graph (for instance due to a perturbation), or when extreme responsiveness is requried. Finally, we are interested in techniques to more deeply incorporate ideas from dynamics into the distance metric and integration process, creating characters who behave in physically plausible ways, in a wide range of situations, using relatively little underlying data.

More generally, we feel that motion fields provide a valuable starting point for motion representations which wish to move beyond a rigidly structured notion of state. We believe that structureless motion techniques—such as ours—have the potential to significantly improve the realism and responsiveness of virtual characters, and that their applicability to animation problems will continue to improve as better distance metrics, integration techniques, and more efficient search and representation methods are developed.

Acknowledgements

This work was supported by the UW Animation Research Labs, Weil Family Endowed Graduate Fellowship, UW Center for Game Science, Microsoft, Intel, Adobe, and Pixar.

References

ARIKAN, O., AND FORSYTH, D. A. 2002. Interactive motion generation from examples. *ACM Transactions on Graphics (ACM SIGGRAPH 2002) 21*, 3, 483–490.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Pushing people around. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 59–66.

BARNES, C., SHECHTMAN, E., FINKELSTEIN, A., AND GOLDMAN, D. B. 2009. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH) 28*, 3 (Aug.).

CHAI, J., AND HODGINS, J. K. 2005. Performance animation from low-dimensional control signals. *ACM Transactions on Graphics 24*, 686–696.

COROS, S., BEAUDOIN, P., YIN, K. K., AND VAN DE PANN, M. 2008. Synthesis of constrained walking skills. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–9.

COROS, S., BEAUDOIN, P., YIN, K., AND VAN DE PANNE, M. 2008. Synthesis of constrained walking skills. *ACM Transactions on Graphics 27*, 5, 113:1–113:9.

COROS, S., BEAUDOIN, P., AND VAN DE PANNE, M. 2009. Robust task-based control policies for physics-based characters. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, ACM, New York, NY, USA, 1–9.

DA SILVA, M., ABE, Y., AND POPOVIĆ, J. 2008. Interactive simulation of stylized human locomotion. *ACM Transactions on Graphics 27*, 3, 82:1–82:10.

ERNST, D., GEURTS, P., WEHENKEL, L., AND LITTMAN, L. 2005. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research 6*, 503–556.

FALOUTSOS, P., VAN DE PANNE, M., AND TERZOPOULOS, D. 2001. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH 2001*, Annual Conference Series, 251–260.

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

HODGINS, J. K., AND POLLARD, N. S. 1997. Adapting simulated behaviors for new characters. In *Proceedings of SIGGRAPH 97*, Annual Conference Series, 153–162.

HODGINS, J. K., WOOTEN, W. L., BROGAN, D. C., AND O'BRIEN, J. F. 1995. Animating human athletics. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, 71–78.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. *ACM Transactions on Graphics 21*, 3 (July), 473–482.

LASZLO, J. F., VAN DE PANNE, M., AND FIUME, E. L. 1996. Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of SIGGRAPH 96*, Annual Conference Series, 155–162.

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 Transactions on Graphics 21*, 3 (July), 491–500.

LEE, Y., LEE, S. J., AND POPOVIĆ, Z. 2009. Compact character controllers. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, ACM, New York, NY, USA, 1–8.

LO, W.-Y., AND ZWICKER, M. 2008. Real-time planning for parameterized human motion. In *SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics symposium on Computer animation*.

MCCANN, J., AND POLLARD, N. 2007. Responsive characters from motion fragments. *ACM Transactions on Graphics (SIGGRAPH 2007) 26*, 3 (July).

MOUNT, D., AND ARYA, S. 1997. Ann: A library for approximate nearest neighbor searching.

MUICO, U., LEE, Y., POPOVIĆ, J., AND POPOVIĆ, Z. 2009. Contact-aware nonlinear control of dynamic characters. *ACM Transactions on Graphics 28*, 3, 81:1–81:9.

PARK, S. I., SHIN, H. J., AND SHIN, S. Y. 2002. On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 105–111.

SHIN, H. J., AND OH, H. S. 2006. Fat graphs: constructing an interactive character with continuous controls. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 291–298.

SMITH, R., 2010. Open dynamics engine, May. `http://ode.org/ode.html`.

SOK, K. W., KIM, M., AND LEE, J. 2007. Simulating biped behaviors from human motion data. *ACM Transactions on Graphics 26*, 3, 107:1–107:9.

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

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

WANG, J. M., FLEET, D. J., AND HERTZMANN, A. 2008. Gaussian process dynamical models for human motion. *IEEE Trans. Pattern Anal. Mach. Intell. 30*, 2, 283–298.

WOOTEN, W. L., AND HODGINS, J. K. 2000. Simulating leaping, tumbling, landing and balancing humans. *International Conference on Robotics and Automation (ICRA)*, 656–662.

YE, Y., AND LIU, K. 2010. Synthesis of responsive motion using a dynamic model. *Computer Graphics Forum (Eurographics Proceedings) 29*, 2.

YIN, K., PAI, D. K., AND VAN DE PANNE, M. 2005. Data-driven interactive balancing behaviors. In *Pacific Graphics*, (short paper).

YIN, K., LOKEN, K., AND VAN DE PANNE, M. 2007. Simbicon: simple biped locomotion control. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, 105.

ZHAO, L., AND SAFONOVA, A. 2008. Achieving good connectivity in motion graphs. In *Proceedings of the 2008 ACM/Eurographics Symposium on Computer Animation*, 127–136.

ZORDAN, V. B., MAJKOWSKA, A., CHIU, B., AND FAST, M. 2005. Dynamic response for motion capture animation. *ACM Trans. Graph. 24*, 3, 697–701.