Motion Fields for Interactive Character Locomotion

By Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović

Abstract
We propose a novel representation of motion data and control of virtual characters that gives highly agile responses to user input and allows a natural handling of arbitrary external disturbances. In contrast to traditional approaches based on replaying segments of motion data directly, our representation organizes samples of motion data into a high-dimensional generalization of a vector field that we call a motion field. Our runtime motion synthesis mechanism freely flows through the motion field in response to user commands. The motions we create appear natural, are highly responsive to real-time user input, and are not explicitly specified in the data.

1. INTRODUCTION
Whenever a video game contains a character that walks or runs, it requires some method for interactively synthesizing this locomotion. This synthesis is more involved than it might at first appear, since it requires both the creation of visually accurate results and the ability to interactively control which motions are generated. The standard techniques for achieving this create realistic animation by directly (or nearly directly) replaying prerecorded clips of animation. They provide control by carefully specifying when it is possible to transition from playing one clip of animation to another. The synthesized motions are thus restricted to closely match the prerecorded animations.

True human motion, however, 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 that 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—thereby admitting any possible pose for a character as a valid state to animate from, rather than being restricted to poses that are near those in the prerecorded motion data. Although there exist techniques that allow one or the other of these abilities, it is the combination of the two that allows for highly agile controllers that can respond to user commands in a short amount of time.

More specifically, a motion field is a mapping that associates each possible configuration of a character with a set of motions describing how the character is able to move from this 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 intelligently 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 predetermined transition points between different animation clips as in motion graphs. 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.

2. RELATED WORK
In the past ten years, the primary approaches for animating interactive characters have been based on carefully replaying precaptured clips of animation. By detecting where transitions can be made from one clip of animation to another without requiring a visually obvious “jump” in the animation, these methods allow both realistic motion and real-time control. In order to better understand our motion fields approach described later, it is useful to have a geometric picture of how these types of methods operate. The full space of all the possible ways in which a character’s body can be posed contains dozens of dimensions, but one can picture it abstractly as a two-dimensional (2D) plane. Each point in this plane then describes a single static pose for the character, while an animation is described by a continuous path. If we imagine this 2D plane as something like a university quad, then the animation could be “traced
out” by following the path made while walking through this quad.

For approaches that rely on directly replaying clips of precaptured animation,\textsuperscript{1,6,7} one can imagine these recorded clips as forming a number of paved paths through the quad. Each path represents the animation data, and the points where paths meet or branch correspond to points where the motion can smoothly transition from one animation to another. Intuitively, these clip-based approaches correspond to the restriction that one always walks on the paved paths, thus representing the ways motions can be synthesized as a graph. Although this restriction means that any synthesized motions closely match the data, most of the space of possible motions is rendered off-limits. This has the disadvantage of making it difficult to quickly respond to changes in user commands or unexpected disturbances, since a change to the motion can only happen when a new edge is reached.\textsuperscript{11,16} Second, because the motions are restricted to the clips that 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.\textsuperscript{20} In essence, our approach of motion fields addresses these problems by allowing the animation to veer “off the paths,” and it uses a set of precaptured animations only as a rough guide in determining what motion to synthesize.

Although a number of methods have been proposed to alleviate some of the representational weaknesses of pure graph-based controllers, including parameterized motion graphs,\textsuperscript{5,11} increasing the numbers of possible transitions,\textsuperscript{2,19} and splicing rag doll dynamics in the graph structure,\textsuperscript{20} the fundamental issue remains: whether 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,\textsuperscript{11} 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,\textsuperscript{3} the character may not react at all to hand pulls or lower-body pushes.

Another group of methods uses nonparametric models to learn the dynamics of character motion in a fully continuous space.\textsuperscript{3,17,18} These techniques are generally able to synthesize starting from any initial state, and lend themselves well to applying physical disturbances\textsuperscript{16} and estimating a character’s pose from incomplete data.\textsuperscript{3} 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 (in that they do not directly incorporate physics except via the data on which they are built), 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, and the design of agile, lifelike, fully-dynamic characters remains 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 a 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.

In this section we will describe how we define a motion field from a set of example animation clips. For the moment we will ignore the question of how a motion field can be used to interactively control a character, instead focusing on synthesizing noninteraction animations. We achieve this by organizing a set of example animation clips into a motion field to describe a single “flow”—allowing an animation to be generated starting from any possible character pose. In Section 4 we will then describe how this technique can be extended to allow for the character to be interactively controlled in real time.

#### 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 a character’s joints. A pose $x = (x_{\text{root}}, p_0, p_1, \ldots, p_n)$ consists of a 3D root position vector $x_{\text{root}}$, a root orientation quaternion $p_0$, and joint rotation 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 $v_{\text{root}}$, 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
\[ \mathbf{v} = \mathbf{x}' \oplus \mathbf{x} = (x'_{\text{root}} - x_{\text{root}}, p'_{0}, p'_{1}, \ldots, p'_{n}) \]

By inverting the above difference, we can add a velocity \( \mathbf{v} \) to a pose \( \mathbf{x} \) to get a new displaced pose \( \mathbf{x}' = \mathbf{x} \oplus \mathbf{v} \). We can also interpolate multiple poses or velocities together (\( \sum_{i=1}^{k} w_{i} \mathbf{x}_{i} \) or \( \sum_{i=1}^{k} w_{i} \mathbf{v}_{i} \)) using linear interpolation of vectors and unit quaternion interpolation on the respective components of a pose or velocity. We use \( \oplus, \odot, \) and \( \sum \) 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 \( m = (x, v) = (x, x' \odot x) \). 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 \( m_{i} = (x_{i}, v_{i}) = (x_{i}, x_{i+1} \odot x_{i}) \). We also compute and store the velocity of the next pair of frames, computed by \( \mathbf{y}_{i} = x_{i+1} \odot x_{i} \). Generally, motion states, poses, and velocities from the database will be subscripted (e.g., \( m_{i}, \mathbf{x}_{i}, \mathbf{v}_{i}, \) and \( \mathbf{y}_{i} \)), 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 \( m_{i} \), we compute a neighborhood \( \mathcal{N}(m_{i}) = \{m_{j}\}_{j=1}^{n} \) of the \( k \) most similar motion states via a \( k \)-nearest neighbor query over the database. In our tests we use \( k = 15 \). We calculate the (dis-)similarity by

\[
\begin{align*}
\text{sim} & = \frac{1}{\eta \cdot d(m, m')^2} \\
\text{dist} & = \frac{1}{\eta \cdot d(m, m')^2}
\end{align*}
\]

where \( m \) is the \( i \)-th neighbor of \( m \) and \( \eta = \sum_{i=1}^{d(m, m')^2} \) 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 \( a_{i} > 0 \)). By increasing the values of some of the \( a_{i} \) weights over others, the direction of an action is biased to better match the directions of the associated neighbors (Figure 1). 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

\[
I(m, a) = (x \odot \sum_{i} a_{i} \mathbf{v}_{i} \odot \sum_{i} a_{i} \mathbf{y}_{i})
\]

Unfortunately, this function frequently causes 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 toward the closest known motion state \( \mathbf{m} = (\mathbf{x}, \mathbf{v}) \) in the database. The strength of this tug is controlled by a parameter \( \delta = 0.1 \):

\[
\mathbf{I}(m, a) = (x \odot v', y')
\]

\[
v' = (1 - \delta) \left( \sum_{i} a_{i} \mathbf{v}_{i} \right) \oplus \delta (\mathbf{x} \odot \mathbf{v} \odot x)
\]

\[
y' = (1 - \delta) \left( \sum_{i} a_{i} \mathbf{y}_{i} \right) \oplus \delta \mathbf{y}
\]

**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 (Equation (2)) as our choice of action. This choice results in the character meandering through the data, generating streams of realistic (albeit undirected) human motion.

4. CONTROL

As described in Section 3, at each possible state of the character, a motion field gives a set of actions that the character

---

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 toward different next states (dashed dots). (a) Weights resulting in an upward motion. (b) Weights resulting in a motion to the right.
can choose from in order to determine their motion over the next frame. In general, which particular action from this set should be chosen 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

From a user’s perspective, the easiest way to control a virtual character is through high-level commands such as “turn to the right” or “walk backwards.” Ultimately, these high-level commands must be boiled down to the low-level actions needed to execute them. In the case of motion fields, these low-level actions are chosen at each frame. Because these low-level choices occur over such a short time scale, it is not always directly obvious which choices should be made in order to satisfy a user’s high-level command. In order to allow for easy user control, it is therefore necessary to connect the single-frame actions used in a motion field with their long-term consequences.

Efficiently planning for the long-term consequences of short-term actions is a standard problem in artificial intelligence. In particular, the domain of artificial intelligence known as reinforcement learning provides tools that can be naturally applied to controlling a character animated with a motion field. In order to apply these tools from reinforcement learning, we formulate the motion field control problem as a Markov decision process (MDP).

An MDP 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, or Treuille and Lo and Zwicker 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 \( \theta \) to keep track of how well the task is being performed, forming joint task states \( s = (m, \theta) \). For instance, our direction-following task \( \theta \) 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 \( s = (m, \theta) \), a character in a motion field has a set of actions \( A(m) \) to choose from in order to determine how they will move over the next frame (Section 3). There are infinitely many different actions in \( A(m) \), 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 \( A(s) \) from \( A(m) \). 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.

\[
\left\{ q_i \right\} = \left\{ \alpha = (w_{i1}, \ldots, w_{i_{k2}}, 1, w_{i_{k2}}, \ldots, w_{i_{k1}}) \right\}
\]

In other words, to derive action \( \alpha \), simply set \( w_{i} \) to 1 and renormalize. This scheme samples actions that are not too different from the passive action at \( m \) so as to avoid jerkiness in the motion while giving the character enough flexibility to move toward nearby motion states.

Transitions. Additionally, we must extend the definition of the integration function \( I \) (Equation (4)) to address task parameters: \( I(s, a) = I(m, \theta, a) = I(m, a, \theta') \). How to update task parameters is normally obvious. For instance, in the direction-following task, where \( \theta \) is the character’s deviation from the desired direction, we simply adjust \( \theta \) 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 a high reward \( R(s, a) \) for maintaining a small deviation from the desired heading and a lower reward for large deviations. See Section 5 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 naive approach to this problem would be to pick the action that yields the largest immediate reward—the greedy policy:

\[
\pi_{c}(s) = \arg \max_{a \in A(s)} R(s, a)
\]

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

Somehow, we need to consider the effect of the current action choice on the character’s ability to accrue future rewards. A lookahead policy \( \pi \), does just this by considering the cumulative reward over future task states:

\[
\pi_{k}(s) = \arg \max_{a \in A(m)} \left[ R(s, a) + \max_{a' \in A(s')} \gamma R(s', a') \right]
\]

with \( s_{i} = I(s, a) \) and \( s_{i} = I(s_{i-1}, a_{i-1}) \). \( \gamma \) is called the discount factor and controls how much the character focuses on short-term \( (\gamma \rightarrow 0) \) versus long-term \( (\gamma \rightarrow 1) \) reward.

As mentioned earlier, 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 function $V(s)$, a scalar-valued function representing the expected cumulative future reward received for acting optimally starting from task state $s$:

$$V(s) = \max_{a \in A(s)} \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$  \hspace{1cm} (10)

We will describe shortly how we represent and precompute the value function, but for the moment notice that we can now rewrite Equation (9) by replacing the infinite future search with a value function lookup:

$$\pi_s(s) = \arg \max_{a \in A(s)} \left[ R(s, a) + V(I(s, a)) \right]$$  \hspace{1cm} (11)

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

**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 5 for details of the sampling). This sampling gives us high resolution near the motion database states, 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. Fitted value iteration operates by first noting that Equation (11) 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:

$$V(s_i) = R(s_i, \pi(s_i)) + V(I(s_i, s_i))$$  \hspace{1cm} (12)

where $\pi(s_i)$ is as defined in Equation (11) and $V(I(s_i, s_i))$ is computed via interpolation. We can solve for $V(s_i)$ at each sample state by iteratively applying Equations (11) and (12). We begin with an all-zero value function $V(s_i) = 0$ for each sample $s_i$. Then at each $s_i$, Equation (11) is used to compute $\pi(s)$ after which we use Equation (12) 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.

**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, each defined over the space of task parameters. Without compression, we store one of these subfunctions $V_{m_i}(\theta_r) = V(m_i, \theta_r)$ at every database motion state $m_i$ (see Figure 3). Here, we observe that our motion states were originally obtained from continuous streams of motion data. At 30 Hz, temporally adjacent motion states and their value functions are frequently similar; we expect that $V_{m_i}$ changes smoothly over “consecutive” motion states $m_i$ 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 $i$th motion state between two anchors $m_u$ and $m_v$ as

$$V_{m_i}(\theta_r) = \frac{N - i}{N} V_{m_u}(\theta_r) + \frac{i}{N} V_{m_v}(\theta_r)$$  \hspace{1cm} (13)

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.

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

**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.
This technique allows the trade-off 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 5).

5. 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.

5.1. Agile responses to user control

**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 5). The reward $R_{\text{direction}}$ for the direction task and the reward $R_{\text{line}}$ for the line-following task are respectively defined as

\[
R_{\text{direction}}(m, \theta, a) = -|\theta| \quad (14)
\]

\[
R_{\text{line}}(m, \theta, d, a) = -|\theta| - 0.05|d| \quad (15)
\]

**Motion data setup.** We used 142 seconds of motion data containing leisurely paced locomotion and quick responses to direction and line changes. We selected the source motion data with minimum care except to roughly cover the space of possible motion. The only manual preprocessing 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 $\theta$. For the line-following task, we take a Cartesian cross product sampling between 18 uniform $\theta$ samples and 13 uniform $d$ samples spanning $-2.0$ to 2.0 meters. 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 (13). 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 a cluster of computers can easily address these time and memory burdens.

**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 using the same motion data, tasks, and reward functions. In order to maximize agility, we allowed a wide range of up to $\pm 45$ degrees of directional warping on clips, and gave minimal importance to the physicality cost (see Lee et al. for details). Figure 6 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 Table 1. We ran a similar experiment for the line-following task. We uniformly sampled user direction changes as well as line

![Figure 5. Task parameters.](image)

![Figure 6. Response time.](image)

![Table 1. Response times in seconds for the direction task until converging within 5 degrees of desired direction](table)

<table>
<thead>
<tr>
<th>Representation</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph-based</td>
<td>0.31</td>
<td>0.94</td>
<td>2.36</td>
</tr>
<tr>
<td>Motion field</td>
<td>0.21</td>
<td>0.40</td>
<td>1.01</td>
</tr>
<tr>
<td>Motion field ×10</td>
<td>0.21</td>
<td>0.43</td>
<td>1.23</td>
</tr>
<tr>
<td>Motion field ×20</td>
<td>0.25</td>
<td>0.66</td>
<td>1.19</td>
</tr>
<tr>
<td>Motion field ×30</td>
<td>0.38</td>
<td>0.78</td>
<td>1.93</td>
</tr>
</tbody>
</table>

Motion field ×10 denotes ten-fold temporally compressed value function on the motion field ($N = 10$). Motion field ×20 and ×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.
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).

**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 10×, 20×, and 30× 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 increase for graph-based value functions.

The approximate nearest neighborhood (ANN) queries represent most of the computational cost. The runtime performance depends on the sample action size $k$ (Equation (7)), 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 can reduce the cost of the $n$ subsequent calls, because the next states tend to be quite close to each other at 30 Hz.

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.

### 5.2. 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 nonphysical 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 test the integration of perturbations into our synthesis algorithm, we integrated pseudo-physical interaction with motion field driven synthesis. This was done by using a physics simulator to determine how the character would respond to a push or a pull, and blending the results of this simulation into the flow of the motion field. We tested the perturbations using both passive and controlled motion fields 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 that 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.

### 6. DISCUSSION

This paper introduces a new representation for character motion and control that allows real-time-controlled motion to flow through the continuous configuration space of the character poses. This flow alters in response to real-time 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 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 (e.g., due to a perturbation) or when extreme responsiveness is required.

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, more recent research has used Gaussian process latent variable
models to perform well even when constructed with relatively little motion data. We also 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.

More generally, we feel that motion fields provide a valuable starting point for motion representations with which 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.

Acknowledgments
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

Gilbert Bernstein and Zoran Popović (gilbert, zoran)@cs.washington.edu, University of Washington, Seattle.