AUTOMATED METHODS FOR DATA-DRIVEN SYNTHESIS OF REALISTIC AND CONTROLLABLE HUMAN MOTION

by

Lucas Kovar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2004

© Copyright by Lucas Kovar 2004

All Rights Reserved

ACKNOWLEDGMENTS

Five years ago, having never taken a course in course in computer graphics, nor read any books on the subject, nor spoken with any researchers in the field, I walked into Michael Gleicher's office and asked to work with him. I do not recommend that others emulate this approach to finding a thesis advisor. Still, it worked out well for me, and Mike has by any measure been a fantastic advisor. He has always been available when I have needed advice or feedback, has looked scrupulously after my best interests, and has granted me considerable freedom to work as I see fit. Thanks, Mike. I would also like to thank the rest of my thesis committee — Stephen Chenney, Nicola Ferrier, Chuck Dyer, and Stephen Wright — for many useful discussions over the past several years, and I am particularly grateful to Stephen Chenney for his careful and patient critique of this dissertation.

This work would not have been possible without the help of many people. John Schreiner was involved with the development of the constraint enforcement algorithm presented in Chapter 3, and Fred Pighin is a co-author of the published version of Chapter 4. Alex Mohr and Hyun Joon Shin have provided sage programming advice, countless illuminating discussions, and pitiless assessments of animation quality. The University of Wisconsin's graphics group has been quite supportive, and Andrew Gardner, Andy Selle, David Dynerman, Matt Anderson, Eric McDaniel, and Rachel Heck have all helped me fix software bugs and understand the (poorly-documented) operation of various pieces of equipment. Motion capture data was generously donated by Demian Gordon, House of Moves, and the Ohio State University graphics group. Intel and Cisco have provided much-appreciated financial support. Finally, I thank Shaima Nasiri for her patience, understanding, and willingness to cook; my friends for making graduate school enjoyable; and my parents for their love and support.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

AUTOMATED METHODS FOR DATA-DRIVEN SYNTHESIS OF REALISTIC AND CONTROLLABLE HUMAN MOTION

Lucas Kovar

Under the supervision of Associate Professor Michael L. Gleicher

At the University of Wisconsin-Madison

Human motion is difficult to animate convincingly — not only is the motion itself intrinsically complicated, but human observers have a lifetime of familiarity with human movement, which makes it easy to detect even minor flaws in animated motion. To create high-fidelity animations of humans, there has been growing interest in motion capture, a technology that obtains strikingly realistic 3D recordings of the movement of a live performer. However, by itself motion capture offers little control to an animator, as it only allows one to play back what has been recorded. This dissertation shows how to use motion capture data to build generative models that can synthesize new, realistic motion while providing animators with high-level control over the properties of this motion. In contrast to previous work, this dissertation focuses on *automated* algorithms that make it feasible to work with the large data sets that are needed to construct expressive motion models. Two models in particular are considered. The first is the *motion graph*, which allows one to rearrange and seamlessly attach short motion segments into longer streams of movement. The second is *motion blending*, which creates motions "in between" a set of examples and can be used to create continuous and intuitively parameterized spaces of related actions. Automated methods are presented for building these models and for using them to generate realistic motion that satisfies high-level requirements.

Michael L. Gleicher

ABSTRACT

Human motion is difficult to animate convincingly — not only is the motion itself intrinsically complicated, but human observers have a lifetime of familiarity with human movement, which makes it easy to detect even minor flaws in animated motion. To create high-fidelity animations of humans, there has been growing interest in motion capture, a technology that obtains strikingly realistic 3D recordings of the movement of a live performer. However, by itself motion capture offers little control to an animator, as it only allows one to play back what has been recorded. This dissertation shows how to use motion capture data to build generative models that can synthesize new, realistic motion while providing animators with high-level control over the properties of this motion. In contrast to previous work, this dissertation focuses on *automated* algorithms that make it feasible to work with the large data sets that are needed to construct expressive motion models. Two models in particular are considered. The first is the *motion graph*, which allows one to rearrange and seamlessly attach short motion segments into longer streams of movement. The second is *motion blending*, which creates motions "in between" a set of examples and can be used to create continuous and intuitively parameterized spaces of related actions. Automated methods are presented for building these models and for using them to generate realistic motion that satisfies high-level requirements.

Chapter 1

Introduction

Animated humans are an important part of a diverse range of media, and they are commonplace in entertainment, training, and visualization applications. At present they are used as characters in games and for special effects in movies; they are part of simulations used by the military to prepare soldiers and by industry to instruct workers in using equipment; and they are used as visualization aids for medical analysis (studying an injured person's gait) and equipment design (determining if controls can be comfortably accessed). Moreover, there is every reason to believe that the demand for animated humans will grow in the future. Because much of our lives are spent observing and interacting with other people, animated humans are a natural and essential part of any visual medium designed to tell stories or simulate real world events.

While animated humans may take the form of cartoons or caricatures, realism is often desirable: more realistic characters can make entertainment applications more engaging and simulations more immersive. However, realistic human motion is quite difficult to animate. The human body is an extremely complicated structure that is capable of actions ranging from subtle and nuanced (such as a shrug) to highly dynamic (such as a back flip). At the same time, people are natural experts on human motion. Having seen other people move our entire lives, we are highly attuned to the subtleties of human movement. For example, a person's mood can be judged purely from body language, and people can identify acquaintances from a distance simply by observing how they walk. As a result, even casual viewers can readily identify inaccuracies in animated motion,

and animations must be of considerable fidelity if they are to be considered realistic by human observers.

The challenge of creating realistic motion is compounded by the fact that one typically needs not just one motion, but many motions. A synthetic actor in a movie, for example, must alter its motion if the director decides to change the script. An agent in a training simulation must adapt its actions to reflect the user's behavior. A character in a game must respond to user input and to changes in the environment. In each case one needs characters with a wide range of possible motions, and these motions must not just be realistic but also *controllable* in the sense that they can be tailored to high-level directions.

In principle, realistic motion can be animated by manually specifying a sequence of character poses, which is known as keyframing. However, this requires an investment of time and artistic talent that is prohibitive for most applications — just a few seconds of high-quality motion may take a skilled artist days to animate. A second possibility is to physically simulate the movement of the human body and compute joint motions that correspond to the desired action [23, 24, 25, 38, 39, 51, 58]. While this approach guarantees that motions are physically plausible (at least within the limitations of the underlying body model), motions that are physically plausible may nonetheless appear unnatural, awkward, or robotic. This is because important properties of real human motion, such as "personality", are neither described nor enforced by physical laws. More generally, the qualities that make motion look human are not readily expressed through sets of equations.

An increasingly popular alternative to keyframing and physical simulation is to record the movement of a live performer, a process known as *motion capture*. Example motion capture technologies include mechanical armatures, magnetic sensors, and specialized multi-camera systems; see Sturman's tutorial [85] for a brief history and Menache's book [61] for a more in-depth introduction. Motion capture data can be used to create high-fidelity animations of effectively any motion that a real person can perform, and it has become a standard tool in the movie and video game industries. However, because motion capture can only be used to reproduce a recorded movement, it offers little control over a character's actions. Imagine, for example, that we want to animate a character walking around in a virtual environment. The character must be able to travel at different

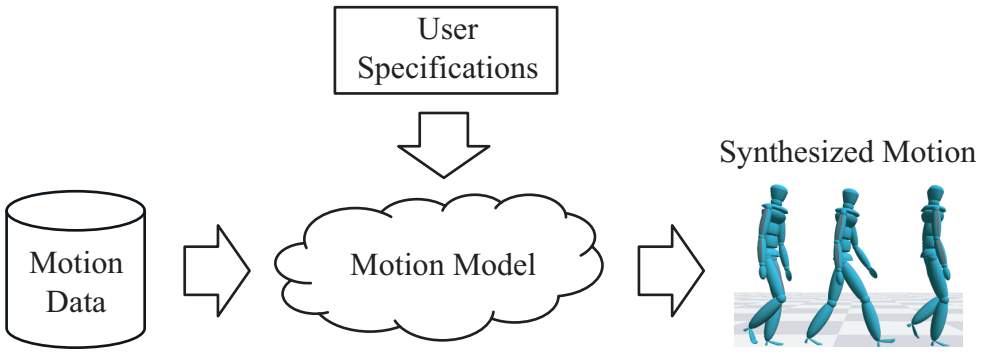

Figure 1.1: Schematic for data-driven motion synthesis.

speeds, follow both smoothly and sharply changing paths, and avoid obstacles. The data, however, only allows the character to perform fixed motions from a limited pool of options. Clearly, it is not possible to capture every variation of every possible way of walking that might be needed, nor is it possible to capture every sequence of steps that might be taken. Similarly, it is infeasible to adjust an animation by directly editing motion data. Just a few seconds of data involves over a thousand parameters, and these parameters are coupled in complicated ways. Manually adjusting data values is analogous to editing a video by changing the colors of individual pixels.

While it is impractical to edit motion data directly, the basic idea of using captured motions to make new ones is nonetheless sound, provided that the data is first converted to a more useable form. *Data-driven motion synthesis* uses example motions to build generative models that can synthesize new, realistic motions through a high-level user interface (Figure 1.1). The goal of data-driven synthesis is to preserve the quality of the original examples while providing intuitive control over a character's actions.

Although the capabilities of data-driven synthesis depend on the underlying motion model, in general this approach can only reliably produce motion that is reasonably similar to the examples. If one wants a flexible motion model that can animate characters with a wide range of movement, then one must start with a large data set. However, previous research has focused almost exclusively on small data sets consisting of a single motion or a handful of motions. This is mainly for practical reasons, as it is only in recent years that it has become economical to acquire larger amounts of data. Nonetheless, it is now common for industry-scale projects to incorporate data sets

containing hundreds or thousands of individual actions. While this makes it possible to animate characters with expressive ranges of motion, it also necessitates a much greater investment of time and effort, since existing methods for building motion models from raw data require considerable labor from the user.

The thesis of this dissertation is that highly automated methods can be used to build generative motion models that offer high-level control and preserve motion realism. Automation is crucial for working with the larger data sets needed for flexible generative models. However, automation also makes it more difficult to guarantee controllability and realism, because a user will no longer be directly overseeing every aspect of a model's construction. To provide control, we develop models that allow users to create motion just by specifying high-level properties, rather than finer details. For example, a walk might be generated by specifying the path that is to be traversed, rather than the location of each individual footstep, and a reach might be created by supplying the location of the target object, as opposed to a detailed trajectory that the hand is to follow. This allows users to direct motion without having to understand the details of the underlying model. To preserve the realism of the captured motions, our models explicitly limit the degree to which generated motion can deviate from the original data, and the user can control this limit to make an appropriate tradeoff between motion quality and model flexibility. Our models also guarantee that synthesized motion preserves the smoothness (C_1 continuity) of the captured motions. This is important because sharp or jittery changes to motion are almost always unrealistic, since real people cannot change their pose instantaneously. Finally, our models infer and enforce kinematic constraints on synthesized motions, which helps ensure that important interactions with the environment are maintained.

Different motion models provide different means of controlling motion. We focus on two motion models that offer complementary forms of control (Figure 1.2): *motion graphs*, which control the sequencing of multiple actions, and *motion blends*, which control the properties of individual actions. A *motion graph* is a directed graph where edges correspond to motion clips and nodes indicate choice points where different clips can seamlessly connect. Motion graphs allow arbitrarily long motions to be constructed from shorter pieces, and they can be used both

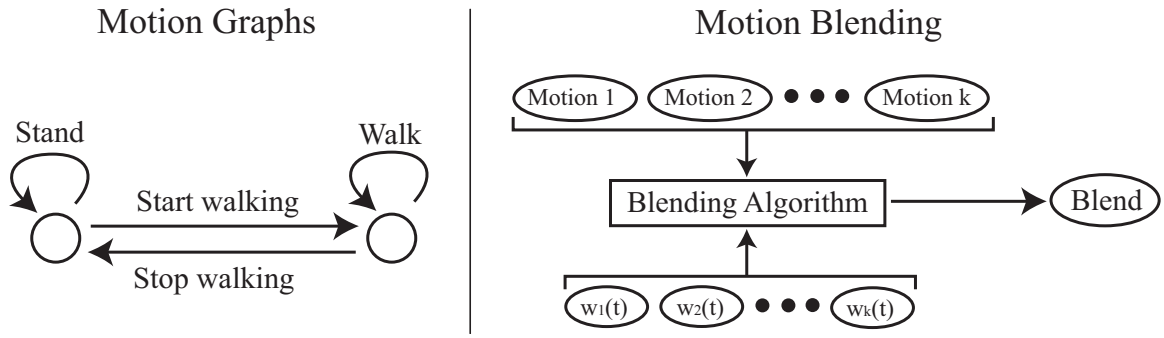

Figure 1.2: Schematic representations of motion graphs and motion blending.

for local control (choosing the next action based on current circumstances) and global control (creating a sequence of actions with desired properties). *Motion blending* combines input motions according to time-varying weights, creating new motions "in-between" the examples. By choosing appropriate weight functions, one can smoothly transition between motions, interpolate motions, and exert continuous control over a motion. Also, given multiple variations of the same action, blending can be used to construct a *parameterized motion*, which is a continuous family of related actions parameterized on high-level features. For example, a parameterized punch might allow an animator to create a specific punch simply by specifying its speed and target location. We have chosen to focus on graph-based and blending-based synthesis because these models are by far the most commonly used, and they appear frequently in previous research [14, 63, 68, 69, 76, 78, 96] and have proven successful in industry [62, 88]. The extension of our methods to other models (in particular, hybrid graph/blending-based models [76]) is beyond the scope of this dissertation, and is left for future work.

This dissertation provides automated methods for building graph-based and blending-based motion models from sets of example motions. The automation not only significantly reduces the time and effort needed to build a model, but also makes it feasible to work with the large data sets needed for expressive models. When combined with motion capture technology as a source of large numbers of high-quality examples, our methods provide a general mechanism for quickly creating realistic motions that meet specific requirements, whether they be provided by a human or by an artificial intelligence module. Four main technical contributions are made:

1. **Efficient and smooth enforcement of kinematic end effector constraints (Chapter 3).** We introduce a constraint enforcement algorithm that can be used to automatically adjust motions synthesized through motion graphs and motion blends so they preserve interactions with the environment, effectively expanding the range of motion that these models can successfully produce.

2. **Automated motion graphs (Chapter 4).** We present a method for automatically building motion graphs from raw motion data, and we further present an automated search framework for extracting motions from these graphs that satisfy high-level constraints.

3. **Improved automatic motion blending (Chapter 5).** We develop an automated blending algorithm based on a novel data structure called a *registration curve*. Registration curves encapsulate relationships between the input motions that currently must be specified manually [68, 76, 78], plus additional information that is important for creating blends. This allows us not only to automate previous blending algorithms, but also to expand the range of motions that can be successfully blended.

4. **Automated construction of parameterized motions (Chapter 6).** To help users collect different variations of the same action, we present a search algorithm for extracting sets of logically similar motion segments from a data set. We also introduce a technique for building accurate parameterizations that is more efficient and stable than existing methods.

Figure 1.3 depicts how these chapters relate to one another. While not directly concerned with motion graphs or motion blends, the constraint enforcement algorithm of Chapter 3 is a vital component of our strategy for preserving motion quality during data-driven synthesis, and hence it is an essential part of this dissertation. Graph-based synthesis is considered in Chapter 4, which may be read independently of the remaining two technical chapters. Both Chapter 5 and Chapter 6 address blending-based synthesis, with Chapter 5 discussing the automation of blending itself and Chapter 6 focusing on parameterized motion as an important application of blending.

Although we will use practical applications as motivation, the methods in this dissertation are not restricted to any particular domain (e.g., movies versus games). However, different applications

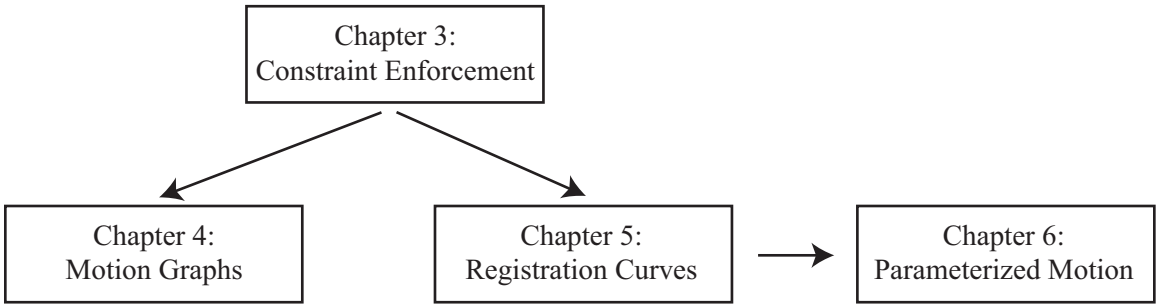

Figure 1.3: Logical relationships among Chapters 3–6.

will typically have very different requirements in a motion model. For example, a lead movie character that fills most of the screen must adhere to very high quality standards, but may need a only a modest range of motion in order to fine-tune a scripted set of actions. In contrast, characters buried within a crowd of thousands can typically accept lower fidelity motion, but they must also have the flexibility to adapt to complex, changing surroundings. To accommodate these different needs, our motion models are equipped with a small number of thresholds that, roughly speaking, determine how much the original example motions may be altered during synthesis, allowing a user to determine an appropriate tradeoff between model flexibility and motion quality.

The remainder of this chapter presents some preliminary technical material and a brief overview of the methods that will be developed in subsequent chapters.

1.1 Preliminaries

1.1.1 Motion Representation

For our purposes, the human body is represented as an articulated figure, or *skeleton*, consisting of a set of ball-and-socket joints arranged in a tree-like hierarchy (Figure 1.4). Each joint has exactly one parent and is associated with a coordinate system defined relative to that of its parent. The exception is the joint at the top of the hierarchy, called the *root*, which has no parent and is associated with a coordinate system defined relative to a (fixed) world coordinate system. A skeleton is constructed by specifying the number and connectivity of its joints plus the offset $\mathbf{o}_i$ of each joint other than the root, where a joint's offset specifies the location of the origin of its

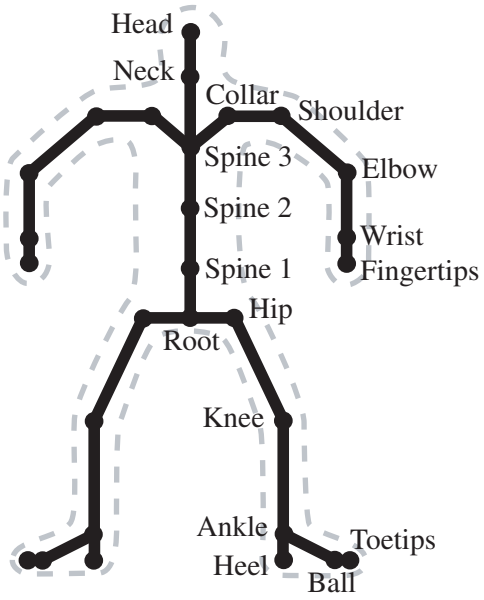

Figure 1.4: Our skeleton model.

coordinate system relative to its parent's coordinate system. The skeleton is typically assumed to have rigid bones, which means that joint offsets are fixed (Chapter 3 relaxes this assumption). All of the experiments performed in this dissertation use the skeleton shown in Figure 1.4. Note that this model can only represent "full-body" motion, since structures such as fingers, toes, and the face are not included. While these details could be added through additional joints, in practice facial and hand capture are performed independently and then integrated with full-body motion as a postprocess.

A *motion* $\mathbf{M}(t)$ is defined as a multidimensional function that specifies the configuration of each of the skeleton's n joints at each point in time:

$$\mathbf{M}(t) = (\mathbf{p}(t), \mathbf{q_1}(t), \ldots, \mathbf{q_n}(t)), \tag{1.1}$$

where $\mathbf{p}$ is the position of the root in the global coordinate system and $\mathbf{q_i}$ is the orientation of the i^{th} joint relative to its parent's coordinate system (for an introduction to coordinate transformations, refer to any standard graphics text, such as Foley et al. [26]). Orientations are represented as unit quaternions [84]; see Grassia [34] for a comparison with other representations, such as Euler angles or orthonormal matrices. Motion data provides a discrete set of skeletal poses, or *frames*,

$\mathbf{M}(t_1), \ldots, \mathbf{M}(t_k)$ that corresponds to a regular sampling of the underlying motion $\mathbf{M}(t)$. We generate root positions in between samples through linear interpolation, and intermediate joint orientations are computed through spherical linear interpolation [84].

While motion data ultimately takes the form of a series of skeletal poses, in practice there is considerable amount of processing needed to obtain this representation from the raw measurements of a motion capture system. In particular, since the human body is not truly a rigid skeleton with simple ball-and-socket joints, the raw measurements must be approximated with a best-fit sequence of skeletal postures. Techniques for performing this fit can be found in the research literature [11, 67, 103] and are standard parts of commercial motion capture processing software. We assume that the conversion of raw measurements to skeletal movement has already taken place.

1.1.2 Constraints

A *constraint* is any property that must be exhibited by a motion. A constraint might state physical limitations, such as a constraint that restricts how far a joint can rotate or prohibits interpenetration of different body parts, or it can relate to properties important to the meaning of a motion, such as a constraint that requires the hands of a clapping character to make contact at certain times. Abstractly, constraints may be thought of as meta-data that attach labels to ranges of frames; these labels might be applied to a single frame, a range of frames, or the entire motion. We assume that motions are already annotated with relevant constraint information. Although automated methods exist for generating these annotations in special cases (e.g., when the character must be in contact with an object in the environment [10, 52]), our experience suggests that manual annotation is preferable if accuracy is desired. We view this as simply another step in the process of converting raw measurements into a useable form, much like fitting skeletal motion to the actual movement of the performer, and we have found that in the context of the full motion capture processing pipeline it is not especially taxing.

1.2 Overview

1.2.1 End Effector Cleanup

Motion graphs and motion blending create new motions through simple procedures that can fail to preserve important features of the original data. One feature that is particularly easy to lose are kinematic constraints on end effectors. These constraints restrict the positions and/or orientations of a hand or foot during specified time intervals, and they typically represent interactions between the character and the environment. For example, one especially common kind of constraint is a *footplant*, which requires part of a foot to remain stationary on the ground. Footplants exist in most circumstances where a character uses its legs to bear weight.

Chapter 3 introduces a new algorithm for adjusting a motion so it satisfies kinematic end effector constraints. While techniques exist for enforcing more general kinds of constraints, they rely on iterative numerical methods that can introduce distracting visual discontinuities into a motion. Our algorithm, in contrast, exclusively uses analytical methods requiring a fixed amount of per-frame computation, and it enforces constraints exactly without introducing discontinuities. This smoothness guarantee is achieved by allowing limbs to change length (which has not been considered in previous work on constraint enforcement) and by explicitly ensuring that changes to individual joint parameters remain continuous when constraints turn on and off. When combined with methods for automatically inferring constraints on motions generated through motion graphs and motion blending (Chapters 4 and 5), our algorithm can automatically correct constraint violations and thereby extend the range of realistic motion that can be produced through these models.

1.2.2 Motion Graphs

Motion graphs (Figure 1.2) have historically been built by hand: an artist determines which segments of captured motion can be connected and uses software tools to adjust them so they can join seamlessly. This can be a tedious and time-consuming process. Chapter 4 shows how a motion graph can be automatically constructed by identifying places in a data set where motions are locally similar and then synthesizing special transition motions at these points. Similarity

is determined through a novel point-based distance metric that factors out irrelevant differences stemming from each motion's choice of global coordinate system. Constraints from the original data are propagated into transitions and enforced using the methods of Chapter 3, allowing simple linear blending methods to be used without sacrificing properties like crisp footplants.

While our approach allows motion graphs to be built with little effort, these motion graphs are generally quite complicated, containing many individual clips with no clear organization. Rather than having a user directly interact with such a graph, Chapter 4 introduces a general search framework for finding a sequence of edges that optimizes a user-supplied objective function and obeys restrictions on what kinds of motion are legal. To demonstrate the potential of this approach, we apply this framework to the problem of creating motion that traverses arbitrary paths.

1.2.3 Motion Blending

Most blending algorithms (Figure 1.2) are not automatic in the sense that they require the user to add special annotations to the input motions. Although automatic algorithms exist, they are quite fragile and can only be used in rather special circumstances. In particular, they have three principal limitations:

1. **Timing differences are not accounted for.** Logically corresponding events in two motions usually occur at different absolute times; for example, the time between successive heelstrikes is shorter in a hurried walk than in a relaxed walk. Existing automatic blending methods ignore timing differences, causing them to combine frames from structurally unrelated parts of the input motions. The effect of this depends on the input motions, but is usually quite bizarre — a blend of the different walk styles, for instance, would not be a new walk but an odd movement where the legs skip across the ground with only cursory contact.

2. **Only very similar root trajectories can be blended.** Existing methods (both manual and automatic) combine root trajectories through simple averaging. If the input root trajectories are not already quite similar, this can cause the blended motion to unnaturally slow down,

produce extreme footsliding artifacts, and create large and discontinuous changes in root orientation.

3. **Kinematic constraints are not applied to blends.** Existing blending algorithms combine motions through relatively simple procedures that fail to preserve kinematic constraints. While these constraints can be enforced as a postprocess (using, e.g., the methods of Chapter 3), the constraints must first be known. Existing automatic algorithms are unable to determine what constraints should be applied to a blend, preventing the application of constraint satisfaction algorithms.

Chapter 5 provides an automatic blending algorithm that avoids these problems. The heart of this algorithm is the construction of a data structure called a registration curve, which encapsulates relationships among the timing, local coordinate frame, and constraint state of arbitrarily many input motions. This information can be used to avoid the problems discussed above without resorting to user intervention. Chapter 5 shows how to automatically build registration curves and use them to create blends, and it demonstrates registration curves in common blending applications such as transitioning, interpolation, and continuous control.

1.2.4 Parameterized Motion

Given a set of example motions representing different variations of the same kind of action, motion blending can interpolate/extrapolate these examples to produce new motions with different properties. For example, given a set of punches that target different locations, interpolation could be used to create punches that target new locations. In principle, a user could then control the action simply by adjusting interpolation weights. However, this is an awkward form of control because interpolation weights typically have no simple connection to high-level motion properties. To correct this, a map can be built between relevant motion features (such as the target location of a punch) and interpolation weights, resulting in a *parameterized motion* that can be controlled directly through these features (Figure 1.5) [76, 96].

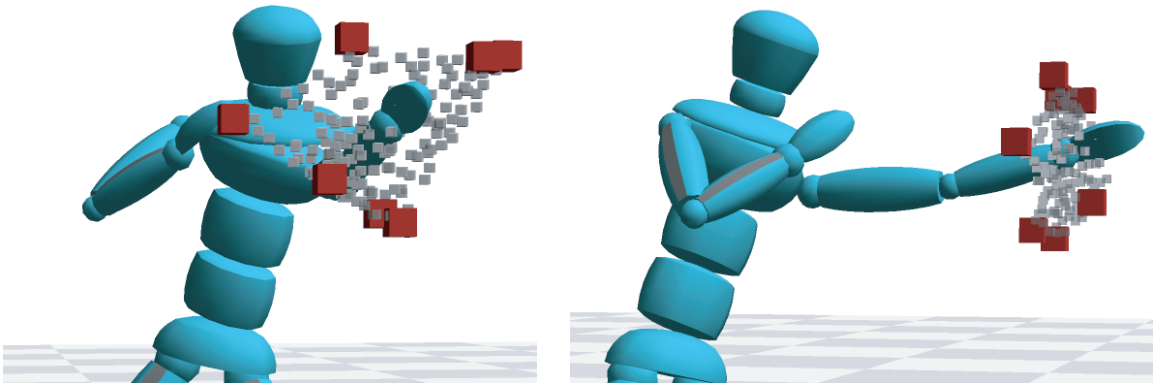

Figure 1.5: A parameterized punch. Red cubes show target locations of captured punches and grey cubes indicate the range of target locations that can be reached by interpolating the examples.

Existing methods for constructing parameterized motions are designed for small data sets consisting exactly of separate example motions that evenly sample a predefined range of variation. In a large data set, however, the necessary examples will in general be buried within longer streams of motion. Currently users must manually scan through the data and extract the segments of interest. This task is complicated by the fact that they must not just find *some* pieces of motion that contain the desired examples, but must rather carefully crop out a set of actions that begin and end in analogous skeletal postures — for example, a parameterized walk must be built from examples that all begin and end at the same point in the locomotion cycle. Also, in a large data set the available range of variation for a given action is generally not known in advance, which is problematic since existing parameterization techniques implicitly assume that the accessible region of parameter space has a simple, known boundary.

Chapter 6 shows how to automate the extraction of examples through a novel database search algorithm that efficiently locates motion segments logically similar to (but perhaps numerically distinct from) a query motion. It also extends existing work on parameterization by providing a more efficient algorithm that ensures accuracy and guarantees that only reasonable blends can ever be created (that is, it explicitly limits the allowable amount of extrapolation from the data). Our methods are demonstrated on a 37,000-frame data set (about ten minutes of motion sampled at 60Hz) containing a variety of different kinds of motion.

Chapter 2

Related Work

This chapter reviews research related to human motion synthesis. We first present a general overview of common strategies, and then for each of the specific topics covered in this dissertation we provide a detailed review of previous work.

2.1 Motion Synthesis Methods

Existing motion synthesis methods can be divided into three categories: manual synthesis, physically-based synthesis, and data-driven synthesis. In this section we discuss specific methods that fall within these categories and summarize some strengths and weaknesses of each approach.

2.1.1 Manual Synthesis

The oldest and most basic synthesis method is to manually specify a character's degrees of freedom (DOFs) at individual points in time, which are called keyframes. DOFs in between keyframes are then computed through simple interpolation methods, such as cubic spline interpolation. Since this interpolation rarely produces realistic results unless the keyframes are quite dense, the artist must manually construct a substantial fraction of the character poses that appear in the motion. This is arduous and time consuming, not only because many poses must be created (animations typically run at 24 frames per second), but also because it is challenging to craft these poses such that they yield a convincing motion when played in sequence. On the other hand, keyframing

provides complete control over every detail of a character's movement, and it remains popular in applications (such as cartoons) where motion need not be realistic as long as it is expressive.

An alternative to keyframing is to handcraft algorithms that procedurally replicate particular motions. This allows an artist to create entire motions at once, rather than one pose at a time, and motions can be altered on the fly to allow for online, interactive character animation. Perlin [69] and Perlin and Goldberg [70] have demonstrated that a variety of motions can be generated with simple and computationally efficient algorithms. However, these algorithms can be quite difficult to design, a new algorithm must be developed whenever a new kind of motion is to be animated, and the generated motion rarely attains the same level of realism as motion capture.

2.1.2 Physically-Based Synthesis

Since real human movement is governed by the laws of physics, physical simulation is a natural strategy for animating motion. Given the mass distribution for each body part and the torques generated by each joint, Newton's laws provide a system of ordinary differential equations that can be integrated to yield joint trajectories. However, while average mass distributions can be obtained from the biomechanics literature [97], finding joint torques that yield a particular motion is considerably more difficult. Hodgins et al. [39] proposed handcrafting joint controllers based on finite state machines and proportional-derivative servos. They demonstrated this approach on several motions, including running, bicycling, and vaulting. Similar methods were employed by Wooten and Hodgins [99, 100] to generate gymnastic motions like flipping or tumbling and by Faloutsos et al. [24] to generate motions for preserving balance and recovering from a fall. While these efforts have successfully produced certain kinds of motion, controller design is a difficult task, and a given controller can only yield a narrow range of motion. On the other hand, once a controller has successfully been created, one can attempt to adapt it to new circumstances. For cyclic motions like walking, Laszlo et al. [51] applied limit cycle control to adjust the underlying joint controller so as to yield a desired variation of the original motion. To adapt a controller to a new body, Hodgins and Pollard [38] computed new controller parameters by first applying approximate scaling strategies and then tuning the results with simulated annealing. Finally, Faloutsos et

al. [23] used support vector machines to learn the conditions under which controllers for different actions can be composed, providing a meta-controller that switches control strategies to transition between actions.

Rather than explicitly constructing joint controllers, some previous work has employed constrained optimization to convert a small number of user-provided keyframes into a full motion that obeys physical laws. Liu and Popović [58] considered the case of ballistic motions, such as leaping or hopping. Given a small number of keyframed poses, a rough motion was first computed via spline interpolation. An optimization was then performed to find a minimal deviation from this initial motion that obeyed calculated trajectories for the linear and angular momentum of/about the center of mass. When the character was airborne, these trajectories were derived from Newton's laws, and when the character was on the ground, they were governed by a biomechanically-inspired model of momentum transfer. To make optimization more efficient, Fang and Pollard [25] demonstrated that physical constraints expressed in terms of aggregate force and torque (rather than the force and torque at each joint) can be differentiated in time linear in the number of character DOFs, whereas in general derivative computation is of quadratic complexity.

Although physically based motion synthesis can automatically generate physically realistic motions, a motion that obeys the laws of physics can appear devoid of personality. Neff and Fiume [65] proposed creating more natural looking motion through an antagonistic joint control model, where opposing muscle forces varied the amount of tension and relaxation. Finding appropriate tension parameters, however, can be a nontrivial task. To create more realistic motions through physically-based optimization, Safonova et al. [80] applied principal components analysis to captured examples of a motion to extract a set of basis poses, which were then used to restrict the space of possible skeletal postures. While this can improve convergence and make individual poses appear more natural, the *sequence* of poses generated by the optimization still does not approach the fidelity of captured motion. More generally, at present physical simulation can only provide a relatively narrow range of realistic motion, and for this reason we use motion capture as our source of example motions. However, our methods would work just as well when applied to physically simulated or hand-animated motion.

2.1.3 Data-Driven Synthesis

Motion capture technology provides a source of highly realistic example motions that can serve as raw material for a variety of data-driven synthesis algorithms, including those introduced in this dissertation. In this section we will discuss previous work in data-driven synthesis that is *not* directly related to motion graphs, motion blending, or parameterized motion. These topics will be deferred until Sections 2.3–2.5. Also, we note that while historically the availability of motion capture data has made data-driven synthesis practical, example motions could also potentially come from keyframe animation, physical simulation, or even another data-driven synthesis algorithm. In this respect the techniques mentioned here are complimentary to our own, as they could readily be applied to motions generated with our methods.

A simple way of creating motion is to apply signal processing operations independently to each degree of freedom of an example motion. Bruderlin and Williams [15] identified several potentially useful operations, including multiresolution filtering, waveshaping, and adding smooth displacement maps. Witkin and Popovic [98] introduced a variant of displacement mapping called motion warping, and Gleicher [32] adapted displacement mapping to provide a simple interface for interactively editing the path traversed by a character. These algorithms are fast and easy to implement, but because they operate on each DOF independently, they are not well suited for adjustments that involve coordinated movement. For example, moving the hand to a particular location can require simultaneous adjustments to the elbow, shoulder, and spine. Gleicher [30, 31], Lee and Shin [54], and Shin et al. [83] coherently adjusted multiple DOFs by using optimization to minimally adjust a motion such that user-specified constraints (e.g., hand position at certain times) were satisfied.

Several researchers have proposed editing operators designed to alter a motion's aesthetic properties. Unuma et al. [90] combined cyclic motions by linearly combining the Fourier coefficients of each DOF, and they reported that in some cases abstract properties such as a walk's "briskness" could be controlled. Given two examples of the same action, one performed neutrally and one with a particular emotion (e.g., angrily), Amaya et al. [4] represented the difference as a warp applied

to the timing and amplitudes of the neutral motion. These warps could then be applied to a different neutral motion, with the goal of adding similar emotional content. Chi et al. [19] applied Laban Movement Analysis to control the expressive content of gestures. Neff and Fiume [66] presented operations for adjusting a motion's succession (how movement spreads from the hip to the extremities), amplitude (total joint range of motion), and extent (proximity of hands and arms to body).

The preceding algorithms all adjust only kinematic properties, but other algorithms are designed to preserve physical correctness. Tak et al. [47, 86] used optimization to ensure that the body's zero moment point remained inside the support polygon, which is a requirement of physical validity. Popović and Witkin [71] presented an editing framework based on physically-based optimization that improved efficiency by first fitting the original motion onto a hand-tailored simplified model that still captured essential dynamical features. Abe et al. [1] employed an optimization framework similar to [58] to adjust captured ballistic motions (such as jumping up and spiking a volleyball) while obeying restrictions on angular and linear momentum. Zordan and Hodgins [102] mapped motion capture data onto a physical simulation by using proportional-derivative servos to produce joint torques that tracked the measured joint angles. A motion could then be edited either by applying new external forces (e.g., to simulate being hit) or by kinematically adjusting the original motion and then tracking the result as closely as possible in the simulation.

2.1.4 Comparison of Motion Synthesis Methods

Manual, physically-based, and data-driven synthesis each have advantages and disadvantages. Manual synthesis provides the greatest degree of control over motion, as it is constrained neither by visual realism nor by physical accuracy. At the same time, manual synthesis requires an extraordinary investment of time and talent, and realistic human motion in particular can only be created with considerable effort from the most skilled of artists. Physically-based synthesis eliminates a great deal of manual labor by, roughly speaking, using physical laws to automatically fill in the details of a character's motion. However, while this approach guarantees that motions are physically accurate (within the limitations of the rather crude approximation that the human body

is a rigid skeleton), physical accuracy does not imply visual realism. For highly dynamic motions, such as a standing broad jump, the laws physics sufficiently constrain the range of possible motion that physically-based synthesis yields convincing results. For more nuanced motions, such as walking, picking up a glass, or smelling a flower, physics provides only a loose set of constraints, and physically-based synthesis yields motions that look awkward and unnatural. Moreover, physically-based synthesis is computationally expensive: current algorithms [25, 58, 80] on modern hardware typically require several minutes to produce a few seconds of motion. Data-driven synthesis, in contrast, can yield highly realistic animations of effectively any motion that a real person can perform, and most algorithms (including those is this dissertation) can produce motion in real time. However, data-driven synthesis requires access to relatively expensive special-purpose motion capture hardware. Also, while in principle data driven synthesis could be applied to any creature whose movement can be captured, in practice it is primarily applicable to humans. Manual and physically-based synthesis, in contrast, can be readily applied to a wide range of body structures.

2.2 Enforcing Kinematic End Effector Constraints

Enforcing kinematic constraints on end effectors is a special case of the *inverse kinematics* (IK) problem, which is to adjust an articulated figure so one or more of its joints are at specified positions and/or orientations. The use of IK in animation dates back to some of the earliest systems [48, 29]. Fundamentally, IK is about finding the solution to a system of nonlinear equations, and IK algorithms may be divided into two classes: numerical and analytical. Numerical algorithms typically arrive at a solution by linearizing the problem about an initial skeletal configuration, moving towards the goal configuration, and repeating until convergence; see Welman [95] for a survey of computational methods. These algorithms can handle sophisticated constraints on complex skeletons, but are considerably more computationally expensive than analytical algorithms. Also, since the equations are ill-conditioned in certain character configurations [60], it is difficult to guarantee that numerical methods will yield smooth changes to a character's DOFs, even when smooth

changes are made to end effector trajectories. Analytical algorithms, in contrast, find efficient closed-form solutions to the IK equations and typically provide smoothness guarantees. However, there is no general method for constructing an analytical IK solver for a given skeletal topology, and in general these solvers only exist for relatively simple topologies with far fewer DOFs than a full-body skeleton.

This dissertation introduces a new analytical IK algorithm that can be used to place wrists and ankles in specified positions and orientations. Our algorithm incorporates a specialized IK solver designed for individual limbs. This single-limb solver is similar to one developed by Tolani et al. [89], except we take measures to avoid "popping" artifacts that can occur when a limb is near full extension. Previous full-body IK algorithms have also treated individual limbs independently [54, 83].

In general, a motion will contain multiple distinct constraints that each last for multiple frames and may or may not overlap in time. Ideally, these constraints will be satisfied by adding a smoothly varying sequence of per-frame adjustments. Performing IK independently on each frame [76] will not in general yield a smooth result, since constraints can discontinuously switch on and off. Monzani et al. [64] smoothly dissipated adjustments into unconstrained frames, but they employed a general numerical IK solver and did not ensure that smooth changes would be made when the number of constraints affecting a character changed (e.g., on one frame both ankles might be constrained, and on the next frame just one ankle might be constrained). We ensure that only smooth changes are made regardless of constraint timing. Gleicher [30, 31] enforced smoothness by representing the total adjustment made to each DOF with a cubic spline and optimizing the knot values. Since the optimization was performed over the entire motion, this process was offline and could scale poorly to long motions, and the restriction that adjustments be represented as a spline means that in general constraints could only be approximately enforced. Our method uses a fixed amount of computation per frame, requires a limited amount of lookahead, and satisfies constraints exactly. Lee and Shin [54] satisfied kinematic constraints with a hierarchical sequence of adjustments. At each stage a hybrid numerical/analytical IK solver was applied independently to each frame and a spline was fit to the resulting displacements, with the knot spacing growing

smaller at later stages of the algorithm. This required IK to be performed on each frame multiple times, whereas our algorithm executes a single per-frame IK calculation and uses a purely analytic IK algorithm.

Our work is related to the task of fitting a skeleton to measured marker positions, such as would be derived from an optical motion capture system. Each marker is assumed to be rigidly attached to a single joint, and sufficiently many markers exist so all of a joint's DOFs can be reconstructed from the markers attached to it. However, some error will be introduced since the human body is not truly a rigid skeleton, and these errors will accumulate if the orientation of each joint in a kinematic chain is independently reconstructed in sequence. This can cause reconstructed end effector positions to deviate substantially from their measured locations. Several methods exist for solving this problem. Bodenheimer et al. [11] used an offline optimization-based numerical IK algorithm (related to that of Zhao and Badler [101]) wherein the user could control the tradeoff between accurate joint orientations and accurate joint positions. Choi and Ko [20] presented a similar online algorithm based on inverse rate control. Shin et al. [83] used a simplified IK solver to perform online skeletal reconstruction where the importance of matching measured end effector positions was tied to their proximity to important objects in the environment (e.g., a doorknob). All of these algorithms rely on having measured end-effector positions, which are not available when motions are generated through data-driven synthesis methods.

2.3 Motion Graphs

The video game industry has long built motion graphs (sometimes called "move trees") for the purpose of character control, although it is only relatively recently that this practice has been published [62]. These motion graphs have historically been constructed manually in the sense that a user explicitly decided which motion clips could be connected. Early research involving motion graphs also built the graphs manually [69, 76]. We show how a motion graph can be automatically constructed by identifying places where motions are locally similar.

Automated graph-based motion synthesis has been the subject of a considerable amount of previous work. Several researchers have built graph-based statistical models wherein each node is a simple generator of poses and edges represent transitions between different kinds of poses [13, 28, 14, 57]. Nodes were formed by clustering poses from the original data, and transition probabilities at edges were computed based on the sequence of poses in the input motions. While statistical models provide a natural means of adding variability to a motion, they also place looser guarantees on motion quality since the true statistics of human motion are unknown (e.g., pose distributions at a given instant are not as simple as a gaussian). Also, these efforts have not focused on controlling the properties of generated motion. Molina-Tanco and Hilton [63] constructed a hidden Markov model wherein the observables corresponded to clusters of motions segments (generated with a k-means algorithm), the hidden states were individual motion segments, and transition probabilities took on one of two values depending on whether the segments of motion were contiguous in the original data. A user could require a motion to start and end in two particular poses, and a dynamic programming algorithm found a maximum-likelihood sequence of connecting motion segments. This work is similar to our own in that generated motion consisted of spliced segments of captured motions, as opposed to the output of a probabilistic model. However, we use a similarity-based cost metric for transitions that can take on a continuum of values, rather than one of two values, and we consider more general constraints on motion synthesis. Pullen and Bregler [72] "stylized" simple keyframed animations by breaking them into smaller pieces and replacing each piece with captured motion. To control the process, a user selected skeletal parameters particularly important to the motion and specified a characteristic time scale. The system then found clips of motion data which matched keyframed parameters at the appropriate frequency band. Our method does not require a keyframed example motion, although one could be supplied as part of an objective function.

Graph-based synthesis has also been discussed in research involving low-DOF characters [49] and motion generated from procedural methods [69], physical simulation [23, 41], and video [82, 81]. These different motion representations require different methods for identifying and creating transitions.

Our work was performed concurrently with (and independently of) that of Lee et al. [52] and Arikan and Forsyth [5], both of whom also automatically identified transition locations based upon a distance metric and then used search algorithms on the resulting graph to extract motions that satisfied user-defined constraints. Aside from technical distinctions in how distances were defined and how transitions were created, the primary differences with our work were in the applications. We consider constraints on the path followed by the character throughout the duration of the motion, with the style of the motion (represented with an annotation, such as "ballet") optionally restricted on different parts of the path. Arikan and Forsyth [5] allowed constraints on the pose, position, and orientation of the character on the first and last frames of animation, with no restrictions on intermediate frames. Lee et al. [52] provided path constraints without style restrictions, generated motion that best fit a video sequence, and presented an interface where a user could directly select what motion was to take place next.

The generation of transitions is an important part of building motion graphs. Our work uses a simple linear blending method similar to that of Perlin [69], except that we propogate kinematic constraints from the original data into the transitions. Other transition methods are possible, such as displacement maps [5, 52, 72], the torque-minimization algorithm of Rose et al. [77], or the blending method introduced in Chapter 5.

There has been much recent interest in graph-based motion synthesis, and a number of extensions have been made to our work and that of Lee et al. [52] and Arikan and Forsyth [5]. To help users build motion graphs with a simpler and more understandable structure, Gleicher et al. [33] developed a system for automatically detecting skeletal poses that appear frequently in a data set. Multi-way transitions were created around these poses, forming high-connectivity graphs with a small number of nodes. Arikan et al. [6] presented an approximate dynamic programming algorithm for extracting motions that satisfy annotation constraints. For the special case of motion with a clear rhythmic structure (such as dancing), Kim et al. [46] automatically segmented motion data using beat analysis, clustered these motion segments with a k-means algorithm, and then built a Markov model representing transition probabilities between different clusters. New motions could then be synthesized that preserved the original data's rhythmic structure while satisfying

constraints on the path traversed by the character. Reitsma and Pollard [75] empirically evaluated the ability of a particular motion graph to perform navigation tasks (i.e., directing a character to a particular position and orientation) by embedding the graph in the environment of interest and calculating 1) the degree to which different portions of the environment could be connected with a motion and 2) the ability of synthesized motions to follow the shortest path between two points. Given a data set containing two coupled motions and a new "control" motion, Hsu et al [40] used an approximate dynamic programming algorithm to construct the complementary motion. For example, given data of two people dancing and a new motion for the leader, an appropriate motion for the follower could be generated automatically. Finally, Lee and Lee [53] controlled motion synthesis in real time by using dynamic programming to precompute a lookup table indicating the optimal transition to take from any node given one of a finite set of goal configurations.

2.4 Motion Blending

Motion blending has been an integral part of several research efforts. One of the earliest is the real-time procedural animation system of Perlin [69]. To animate a character, a user manually constructed a set of base motions and then used blending operations to transition between motions and to create new ones via interpolation. The language for constructing motions contained built-in timing and constraint models, simplifying the related blending issues. Several systems have used multitarget interpolation to create parameterized motions [96, 78], and the Verbs and Adverbs system of Rose et al. [76] combined this with linear blend transitions to create a motion graph containing parameterized motions. Multiple research groups have used general blending (that is, blending with arbitrary weight functions) to provide continuous control of locomotion [35, 7, 68]. Registration curves support all of these operations (interpolation, transitioning, and continuous control) and could be used as a back end in these systems for computing blends.

2.4.1 Timing

Motions with different timing may be *timewarped* such that logically related events occur simultaneously; we refer to this mapping between corresponding frame indices as a *timewarp curve*. We build timewarp curves automatically and for arbitrarily many motions. Timewarping has also been an important part of previous work on motion blending. Some systems required the user to mark sparse "key times" in the input motions to acquire time correspondences [76, 68]; for example, for a set of punches these might be the initiation, apex, and retraction of each punch. Our technique requires no user intervention. Ashraf and Wong [7] semi-automatically labelled motions with sparse timing information, such as zero-crossings of the second derivative of user-specified joint angles. Timewarp curves were generated through a greedy algorithm that matched identical labels (e.g., the same joint angle had experienced a zero-crossing). Our algorithm generates dense correspondences and uses a dynamic programming technique with stronger optimality properties. The method of Bruderlin and Williams [15] is the most similar to our own, as it also is based on dynamic programming. However, their algorithm does not yield one-to-one timewarp curves (and hence does not provide an invertible mapping between frames), and it is only designed for two input motions. We extend this method to produce timewarp curves that provide one-to-one mappings for example sets containing an arbitrary number of motions.

2.4.2 Root Blending

Most previous work has used linear interpolation to compute blended root parameters, which can cause the blended root trajectory to collapse if the input trajectories are not quite similar. A notable exception is the work of Park et al. [68], where the root was positioned and oriented according to a user-specified path (Gleicher [32] used a similar method to interactively edit individual motions). Blending was then used to determine individual skeleton poses appropriate for the local speed and curvature of the root trajectory. We blend root parameters directly, so there is no need to specify the root path. Also, Park et al.'s method was based on the assumption that the input motions were sufficiently smooth that the root path could be well approximated by a circular

arc. Our method is applicable to motions with sharply varying path curvatures (see, for example, Figure 5.17 of Chapter 5).

2.4.3 Automatic Constraint Annotation

Bindiganavale and Badler [10] automatically detected contact constraints between end effectors and other objects by identifying when acceleration zero-crossings coincided with close proximity to items of interest. This method is suitable only for finding constraints that actually occurred in a captured motion, whereas for blends we are interested in the constraints that *should* be satisfied. Rose et al. [76] determined constraints on a blend by manually specifying corresponding constraints in the input motions and blended the boundaries over which these constraints were active. Our method is similar, except corresponding constraints are identified automatically. Ashraf and Wong [8] copied the constraints from the input motion with the currently highest blending weight. While this approach works well for the special case of transitions, it can produce artifacts with other blending operations such as interpolation. In particular, when the blend weights are nearly equal a small change in the weights may produce large changes in the constraints. Our technique ensures that constraints change smoothly if the blend weights change smoothly.

2.5 Parameterized Motion

Wiley and Hahn [96] and the Verbs and Adverbs system of Rose et al. [76] pioneered the technique of building parameterized motions from interpolations of captured examples, and the Verbs and Adverbs system in particular introduced the now-popular approach of applying scattered data interpolation methods to approximate the mapping between motion parameters and interpolation weights. This dissertation builds upon this work by introducing a new search algorithm for extracting the initial example motions from a data set and by providing a more efficient and robust method for building very general kinds of parameterizations. The remainder of this section expands on previous research relating to searching motion data sets and parameterizing the space of interpolated motions.

2.5.1 Searching for Example Motions

Searching a motion data set for segments similar to an example motion is related to the time sequence retrieval problem, which has been studied by the database community for over a decade. Given a distance metric and a query time sequence, the task is to search a database for time sequences whose distance to the query is either below a threshold ϵ or among the k smallest. Most proposed solutions follow the GEMINI framework proposed by Faloutsos et al. [22]. First, a low-dimensional approximation is extracted from each time series in the database. Example approximations include the first few coefficients of a Fourier [2] or wavelet [18] transform, the average values in adjacent windows [43], and bounding boxes [92]. Next, a distance metric is defined over this approximation that underestimates the true distance between the time series. Finally, the approximated signals are stored in a spatial data structure such as an R-tree [36].

This approach provides efficient pruning of portions of the database that are distant from the query while keeping dimensionality low enough that spatial access methods remain viable [12]. However, it also determines similarity through direct numerical comparison, comparing a computed distance against a user-supplied threshold. This makes it impossible to distinguish unrelated motion segments from variations of the query that are numerically quite different — the only discriminant used in the search process is the distance measure, and both kinds of motion are distant from the query. In particular, finding increasingly different variations of the query requires using higher and higher distance thresholds, leading to a continually growing collection of spurious results that must be manually discarded by the user. Our strategy is instead to find close motions and then use them as new queries, allowing one to ultimately find motions distant from the query without resorting to higher thresholds. This strategy is inspired by manifold learning algorithms that use local neighborhoods of points to infer the structure of a low-dimensional manifold embedded in a high-dimensional space [79, 87]. Jenkins and Matarić [42] used a similar strategy to extract motion primitives from human motion data, although their focus was on controlling robot motion, rather than producing high-fidelity animation. A second important problem with the GEMINI framework is that in general it does not provide precise start and end frames for each match, but rather returns a larger range of frames that surrounds the "true" match. This is because perturbing

the boundaries of a motion segment will usually produce only a small change in distance to the query, and unless the initial threshold is quite tight these perturbations will also be returned by the search algorithm. Our search method effectively returns "local minimum" matches that are optimal under perturbation, preventing the user from having to tune boundary frames manually or hunt through nearly identical results. A final limitation of existing search algorithms is that they assume the distance between individual data elements (i.e., skeletal poses) can be computed with an L_p norm. Frame distance metrics that represent orientations with quaternions [52] fail this criterion. We allow arbitrary distance metrics to be used for individual frames.

Content-based database retrieval has appeared in a number of graphics contexts, including images [17], video [91], and 3D models [27]. To search motion capture data sets, Cardle et al. [16] and Keogh et al. [44] used variants of the GEMINI framework. Liu et al. [59] automatically extracted keyframes for each motion in a database and used these keyframes to construct a hierarchical tree of clusters of motions, with deeper levels of the tree corresponding to joints deeper in the skeletal hierarchy. To process a query, the closest leaf cluster was found and its motions were directly compared against the query. This algorithm also uses a direct numerical comparison to determine similarity, and it is designed to compare entire motions against a query, whereas our algorithm is also able to compare subsections of motions.

The problem of searching for motions is related to that of generating descriptive labels. Arikan et al. [6] used support vector machines to automate the process of annotating a motion data set. Other researchers have developed automated methods for clustering and segmenting [63, 46, 9] motion data based on behavioral similarity, which may also be viewed as mechanisms for generating labels. While none of these techniques are designed to provide precise boundaries between different actions, even a coarse set of annotations can make database search more efficient and robust (see Section 6.2 of Chapter 6).

2.5.2 Parameterizing Interpolations

Many methods for building parameterized motions are inaccurate in the sense that they do not ensure that the actual parameters of synthesized motions are the same as the desired parameters.

This is reasonable for qualitative properties like "happiness", where strict accuracy is neither quantifiable nor necessary [90, 76]. Similarly, strict accuracy is unnecessary if motion interpolation is not being used to directly control parameters of interest. For example, while Park et al. [68] used motion interpolation to create locomotion with specified speed and curvature, the actual path of the root was determined through a user-specified trajectory. Nonetheless, in many cases accurate parameterizations are essential. Rose et al. [78] improved the accuracy of scattered data interpolation by adding additional samples to parameter space. Specifically, they identified sets of target parameters for which the accuracy was particularly poor and used gradient descent to find interpolation weights that yielded those parameters. We offer an alternative solution based on more directly sampling the space of interpolations. The general strategy of sampling the space of interpolations was previously used by Wiley and Hahn [96] to obtain regular samplings of parameter space and by Zordan and Hodgins [102] to generate dense sets of example motions as an aid for inverse kinematics tasks, although these efforts were not focused on improving the accuracy of parameterization. Also, we expand on this previous work by showing how to restrict interpolation weights to reasonable values and how to sample in a way that scales well to large numbers of examples.

Previous work on parameterized motions has performed scattered data interpolation by computing a best-fit linear map between interpolation weights and motion parameters and then adding radial basis functions centered on each example [76, 68, 78]. This can produce interpolation weights containing large negative weights [3], and if the user requests parameters far from the examples, interpolation weights are based purely on the linear approximation and hence are effectively arbitrary. Also, the run time of this algorithm is $O(n)$ for n example motions. We propose instead using k-nearest-neighbors interpolation, as suggested by Allen et al. [3]. This allows us to explicitly constrain interpolation weights to reasonable values, project points outside the accessible region of parameter space back onto it, and compute interpolations in time that is nearly independent of the number of example motions.

Chapter 3

End Effector Cleanup

3.1 Introduction

Real human motion often contains interactions between end effectors (hands and feet) and parts of the environment, and each of these interactions may be thought of as a kinematic constraint that requires the relevant end effector and the object to be in contact. For example, one of the most common kinematic constraints is a *footplant*, which requires part of a foot to remain stationary on the ground. Footplants are ubiquitous because they are present in almost any motion where a character uses its legs to support its weight. Regardless of the accuracy of the original motion capture data, constraints such as footplants can easily be violated. This is because the raw data is rarely used directly, but rather is first fit onto a rigid skeleton and then altered to satisfy the particular demands of the animation. Indeed, the data-driven synthesis techniques developed in Chapters 4–6 produce motions that in general violate kinematic end effector constraints. Even modest constraint violations can be quite distracting because they break the logical connection between the character's actions and its surroundings — for example, when a footplant is violated, the character appears to unnaturally penetrate or slide across the ground, an artifact known as *footskate*.

This chapter presents a new algorithm for enforcing kinematic constraints on end effectors. This algorithm is a crucial component of the motion models developed in the following chapters, as it provides a mechanism for automatically enforcing these constraints in synthesized motion. Our algorithm is appropriate for constraints that require part of an end effector to be stationary

(such as when standing on the ground or leaning on a table), to reach a particular point in space (such as when pushing a button or kicking a ball), or to maintain contact with an object for a prolonged duration (such as when carrying a box). It exactly enforces constraints while ensuring that no discontinuous changes are made to the motion, and it exclusively uses analytic methods involving a fixed amount of calculation for each frame. It is also simple and easy to implement, as it is based entirely on straightforward geometric manipulations. Finally, our algorithm processes an individual frame using only information from a fixed neighborhood of surrounding frames, which makes it useful for online applications that can accept a small time delay ($\frac{1}{4}$s–1s), such as animating background characters in a game or individuals in a crowd simulation.

While previous work [30, 31, 54, 20, 83] has assumed that skeletons are perfectly rigid, we allow bones to change length. We argue that rigid skeletons force one to make a precarious and unnecessary tradeoff between satisfying constraints exactly and avoiding sharp adjustments to joint parameters. While more specific reasons will be covered in detail later, our decision is unusual enough that we would like to explain up front why we believe it to be reasonable. First, since end effector cleanup is intended as a postprocess, editing operations which require fixed bone lengths are still usable. Second, support for variable bone lengths exists in a variety of standard data formats (such as BVH, BVA, Acclaim, and HTR [50]) and animation packages (such as Poser, Maya, and 3D Studio Max). Third, variable bone lengths can be seamlessly incorporated into traditional algorithms for specifying how a skeleton drives a character's mesh [56]. Lastly, recent perceptual experiments by Harrison et al. [37] suggest that the amount of non-rigidity that we require in practice is difficult to detect even when viewers are specifically looking for changes in limb length.

In the remainder of this chapter, we provide the details of our algorithm (Section 3.2), present some results (Section 3.3), and conclude with a brief discussion (Section 3.4).

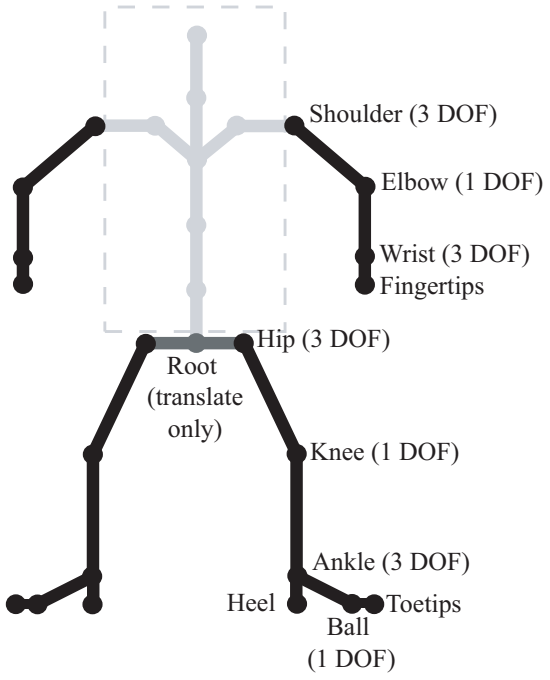

Figure 3.1: Skeleton model for end effector cleanup.

3.2 An End Effector Cleanup Method

3.2.1 Algorithm Overview

For the purpose of end effector cleanup, we model the body as shown in Figure 3.1. The orientations of the spine joints and the root are not altered, although we allow changes to the root position. While allowing the spine to bend would provide our solver with more flexibility, this would also require iterative optimization procedures [54] that lack the speed and robustness advantages of the analytic algorithm presented here. Each limb is treated as a 3 DOF joint (the shoulder/hip) followed by a 1 DOF joint (the elbow/knee) and a 3 DOF end effector (the wrist/ankle); extra degrees of freedom exhibited by the elbow/knee are left unchanged. The elbow and knee are prohibited from bending backwards, but no orientation limits are enforced for the other joints. The two segments of each limb are allowed to vary in length, but the end effectors are assumed to be rigid (e.g., the offsets of the heel and ball do not change).

While a user may directly provide target positions and orientations for an end effector, it is often convenient to supply more general restrictions on end effector configuration. We discuss in detail the case of plant constraints on end effector tips. These are constraints that require either the heel, ball, wrist, or fingertips to remain in a fixed position during some time interval. In general the plant position is unspecified, although it may be constrained to lie on a planar surface. Given constraints of this form, our system determines appropriate positions for the end effector tips (Section 3.2.2) and corresponding end effectors configurations (Section 3.2.3). Note that plant constraints are a natural generalization of footplant constraints, for which the end effector tip is the heel or ball of a foot and the planar surface is the ground. Constraints on the very tips of the toes are not considered, although we do ensure that the toes don't penetrate the ground as a result of other constraints being satisfied[1]. Other kinds of restrictions on end effector configuration, such as ones used to avoid object penetration, are also not discussed. If such constraints exist, then we assume that an outside process will provide our algorithm with appropriate end effector positions and orientations.

The basic idea behind our algorithm is simple. Frames of motion are processed sequentially, so both fixed-duration motions and continuous streams of motion can be handled. To place end effectors in given positions and orientations, an analytic inverse kinematics (IK) algorithm is used to adjust individual limbs. This IK algorithm creates smooth changes as long the target end effector configurations vary smoothly, and in particular damps unnaturally fast changes that can occur when a limb is near full extension. To ensure that smooth changes are made across constraint boundaries, our algorithm incorporates information from the surrounding neighborhood of frames when adjusting a particular frame.

Our approach is divided into five steps. The first two steps assume that constraints are specified as plant constraints on end effector tips; if target end effector positions and orientations are supplied directly, then these steps may be skipped.

1. Determine a position for each plant constraint.

[1]Toetip constraints are quite rare in practice. For example, a "tiptoeing" motion really involves the weight being supported on the toes *and* the end of the ball, and hence the correct result can be obtained simply by planting the ball.

2. For each constrained frame, compute global positions and orientations for the end effectors that satisfy the constraint positions found in the previous step. Intuitively, the end effector is "chopped off" and arranged so the plant constraints are satisfied. Care is taken to ensure that target end effector configurations change continuously when constraints switch on and off.

3. Calculate where the root should be placed so the target end effector positions can be reached by extending or contracting the limbs. This goal is balanced with ensuring that the adjustments made to the root are smooth, so in certain cases the target end effector positions may be slightly out of reach. If so, the residual distance is covered in the following step by stretching the limbs.

4. Using the root position found in the previous step, adjust each limb as necessary so target end effector configurations are reached. As much of the adjustment as possible is accomplished by modifying joint orientations, but when a limb is near full extension its length may be adjusted to avoid sharp changes to the knee/elbow.

5. The previous steps have ensured that smooth adjustments are made as long as at least one constraint exists on an end effector. However, there may still be discontinuities when an end effector switches from zero constraints to one or more constraints or vice-versa. This is avoided by smoothly dissipating adjustments into unconstrained frames.

Each step relies on results from the previous step over a neighborhood of nearby frames (Figure 3.2). Larger neighborhoods provide smoother results, but for online applications this comes at the expense of greater delay.

The remainder of this section describes each step of the algorithm and then concludes with a discussion regarding efficient implementation.

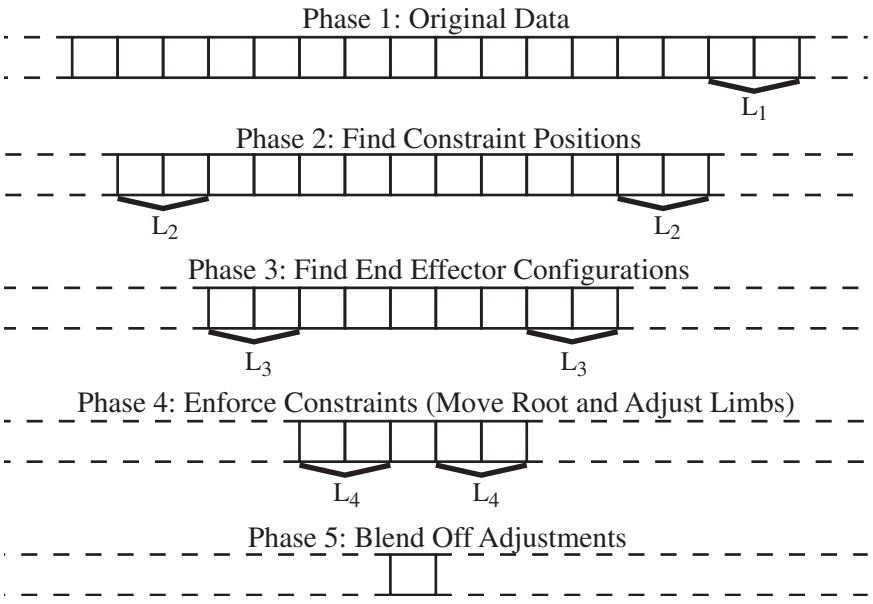

Figure 3.2: The phases of the constraint enforcement algorithm. First, plant positions are determined. Next, end effector configurations are found that satisfy these constraints. The root position and limb parameters are then adjusted to place the end effectors in these configurations. Finally, changes are dissipated into unconstrained frames. Each phase requires results from the previous phase over a neighborhood of surrounding frames; for example, in order to find the ankle configuration at frame $\mathbf{M_i}$, the constraint positions for the heel and/or ball must be known from $\mathbf{M_{i-L_2}}$ to $\mathbf{M_{i+L_2}}$.

3.2.2 Finding Plant Constraint Positions

Each frame may have up to eight plant constraints, one for each heel, ball, wrist, and fingertips. In some cases the location of a plant will be known a priori; for example, the character may be required to step on a pedal. If this is not the case, then plant locations must be computed.

Each end effector has two positions in its local coordinate system that may be planted. For an ankle, these positions correspond to the heel and the ball; for a wrist, they correspond to the wrist itself and the fingertips. Consider the problem of finding a position on frame $\mathbf{M_i}$ for a plant constraint on joint J_1 that spans n_{J_1} frames. Let J_2 be the other joint for that end effector, and let $\mathbf{o_1}$ and $\mathbf{o_2}$ be, respectively, the offset of J_1 and J_2 in the end effector's coordinate system (Figure 3.3). Since the motion is processed sequentially, the positions of all constraints existing on previous frames are already known, so it is sufficient to handle the case where the constraint starts on $\mathbf{M}_i$.

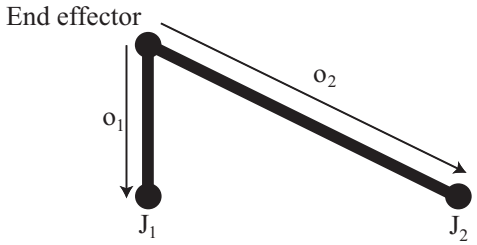

Figure 3.3: Diagram of an end effector. If the end effector is a hand and J_1 is the wrist, then $\|\mathbf{o_1}\| = 0$.

First, assume that J_2 either is not constrained on $\mathbf{M_i}$ or has a plant constraint for which a location has not yet been determined. The plant position $\mathbf{c_1}$ for J_1 is then found by averaging the global position of J_1 over the next $min(L_1, n_{J_1})$ frames and, if the plant is required to be in a plane, projecting the result onto this plane. L_1 is a user-defined constant.

Now assume that J_2 also has a plant constraint on $\mathbf{M_i}$ and that the position $\mathbf{c_2}$ of this plant is known. Since the hands and feet are treated as rigid objects, J_1 must be planted a distance $\|\mathbf{o}_1 - \mathbf{o_2}\|$ from J_2. Our strategy is to place J_1 in a position consistent with the average orientation of the end effector during the frames when both J_1 and J_2 are constrained. Let m be either the number of frames when both J_1 and J_2 are constrained or a user-defined constant L_1, whichever is smaller. Also, let $\mathbf{Q_j}$ be the global orientation of the end effector on frame $\mathbf{M_j}$, where we use capital letters to distinguish global orientations from local orientations. An average orientation $\overline{\mathbf{Q}}$ is defined by computing

$$\overline{\mathbf{Q}} = \frac{1}{1+m} \sum_{j=i}^{i+m} \mathbf{Q}_j \tag{3.1}$$

and then normalizing the result so $\overline{\mathbf{Q}}$ is a unit quaternion. $\mathbf{Q_j}$ is replaced with its antipode whenever the vector dot product of $\mathbf{Q_j}$ and $\mathbf{Q_i}$ is negative, which ensures that all of the $\mathbf{Q_j}$ are on the same hemisphere of the unit quaternion sphere [55]. We experimented with alternative (and slower) methods for averaging orientations based on the logarithmic map [68], but the differences were negligible. The plant location for J_1 is set to

$$\mathbf{c_1} = \mathbf{c_2} + \overline{\mathbf{Q}}(\mathbf{o_1} - \mathbf{o_2}). \tag{3.2}$$

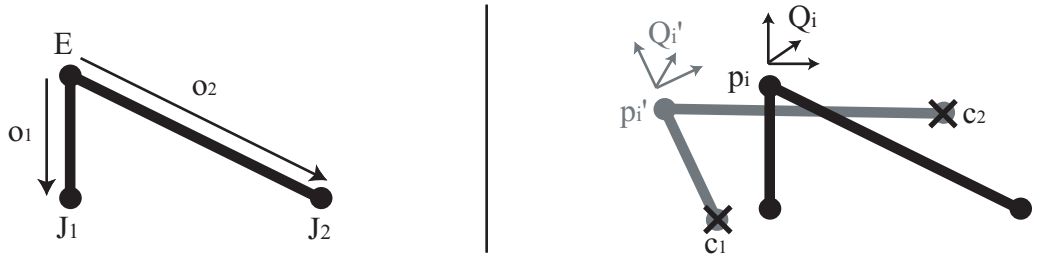

Figure 3.4: Left: Diagram of an end effector. Note that $\mathbf{o_1}$ and $\mathbf{o_2}$ are fixed. **Right:** The configuration $(\mathbf{p_i}, \mathbf{Q_i})$ of E on frame $\mathbf{M_i}$ is changed to $(\mathbf{p'_i}, \mathbf{Q'_i})$ so J_1 and J_2 are placed, respectively, at $\mathbf{c_1}$ and $\mathbf{c_2}$.

If $\mathbf{c_1}$ is constrained to lie in a plane, then it is rotated about $\mathbf{c_2}$ by the smallest rotation needed to place it in that plane.

3.2.3 Finding Target End Effector Configurations

Once locations for plant constraints are known, the next step is to find end effector configurations that satisfy these constraints. For example, if the heel and ball of the left foot are both planted, then the ankle must be set in a position and orientation that places these joints at their plant locations. We now discuss the problem of finding end effector configurations on frame $\mathbf{M}_i$ given plant positions for all constraints that are active from frames $\mathbf{M}_{i-L_2}$ to $\mathbf{M}_{i+L_2}$, where L_2 is a user-defined constant.

A foot is called *doubly constrained* if both the heel and ball are planted, *singly constrained* if only one of these joints are planted, and *unconstrained* otherwise. Similar definitions are used for hands. We now consider each of these scenarios for a given end effector E.

- E **is doubly constrained.** Let J_1 and J_2 be the joints that are to be planted, $\mathbf{o}_j$ be J_j's offset in $E's$ local coordinate system, and $\mathbf{c_j}$ be J_j's plant location (Figure 3.4). It is assumed that $\|\mathbf{o_2} - \mathbf{o_1}\| = \|\mathbf{c_2} - \mathbf{c_1}\|$. Also, let $\mathbf{p_i}$ and $\mathbf{Q_i}$ be the original global position and orientation of E on $\mathbf{M}_i$, and let $\mathbf{p'_i}$ and $\mathbf{Q'_i}$ be the (as yet undetermined) adjusted global position and orientation. When J_1 and J_2 are at their constraint locations, E is still free to rotate about the vector $\mathbf{c_2} - \mathbf{c_1}$. This degree of freedom is called the *roll*. If desired, the roll could be set explicitly, perhaps to orient a foot flat on the ground. Alternatively, the roll can be chosen

to minimize the difference between $\mathbf{Q_i}$ and $\mathbf{Q'_i}$. This can be done by finding the smallest rotation that, when applied to $\mathbf{Q_i}(\mathbf{o_2} - \mathbf{o_1})$, orients it in the same direction as $\mathbf{c_2} - \mathbf{c_1}$, where the magnitude of a rotation $\mathbf{q}$ is defined as the magnitude of its logarithmic map ($\|log(\mathbf{q})\|$). Define $\mathbf{rot}(\mathbf{a}, \mathbf{b})$ as a function that takes two vectors $\mathbf{a}$ and $\mathbf{b}$ and returns a rotation of $\cos^{-1}\left(\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}\right)$ degrees about $\frac{\mathbf{a}\times\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$. The closest orientation to $\mathbf{Q_i}$ that satisfies the plant constraints is then

$$\mathbf{Q'_i} = \mathbf{rot}\left(\mathbf{Q_i}\left(\mathbf{o_2} - \mathbf{o_1}\right), \mathbf{c_2} - \mathbf{c_1}\right)\mathbf{Q_i}. \tag{3.3}$$

With this orientation, the target global position of E is

$$\mathbf{p'_i} = \mathbf{c_1} - \mathbf{Q'_i o_1}. \tag{3.4}$$

- E **is singly constrained.** Let J_1 be the joint that is to be planted, with $\mathbf{o_1}$ and $\mathbf{c_1}$ defined as before. Given *any* orientation $\mathbf{Q'_i}$ for E, the plant constraint can be satisfied by setting $\mathbf{p'_i}$ as in Equation 3.4. Hence E could simply retain its original orientation and be translated by $\mathbf{c_1} - \mathbf{Q_i o_1}$. However, a discontinuity may exist if E is doubly constrained on $\mathbf{M_{i-1}}$ or $\mathbf{M_{i+1}}$, since in general E would have to be rotated to meet both plant constraints.

  To avoid this discontinuity, rotations applied to E on doubly constrained frames are blended off into nearby singly constrained frames. Let $\alpha(t)$ be a C^1 function such that $\alpha(0) = 1$, $\alpha(1) = 0$, $\frac{d\alpha}{dt}(0) = \frac{d\alpha}{dt}(1) = 0$. The unique cubic polynomial satisfying these conditions is

$$\alpha(t) = 2t^3 - 3t^2 + 1 \tag{3.5}$$

  The algorithm starts by looking forward and backward L_2 frames. For each direction, it finds the first frame where E is unconstrained or doubly constrained. If only singly constrained frames are encountered, or if an unconstrained frame is reached before any doubly constrained frames, then there is a *miss* in that direction; otherwise there is a *hit*. When both directions results in misses, then no changes are made, and $\mathbf{Q'_i} = \mathbf{Q_i}$. Otherwise there are three possibilities (Figure 3.5):

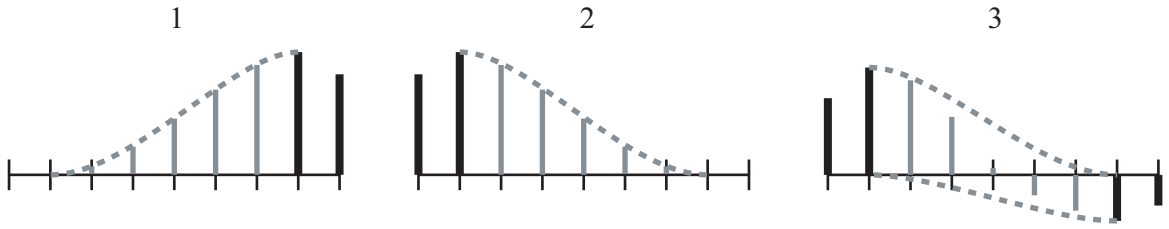

Figure 3.5: Blending off orientation adjustments from doubly constrained frames (thick black lines) to yield orientation adjustments on singly constrained frames (grey lines). There are three cases: 1) There is an adjustment in the forward direction but not the backward. 2) There is an adjustment in the backward direction but not the forward. 3) There are adjustments in both directions.

1. There is a hit going forward at frame $\mathbf{M_{i+k}}$, but a miss going backward. If the orientation adjustment applied to E on $\mathbf{M_{i+k}}$ is $\mathbf{\Delta_{Q_{i+k}}} = \mathbf{Q'_{i+k}Q_{i+k}^{-1}}$, then E's global orientation on the current frame is premultiplied by

$$\mathbf{\Delta_{Q_i}} = \mathbf{slerp}\left(\alpha\left(\frac{k}{L_2+1}\right), \mathbf{I_Q}, \mathbf{\Delta_{Q_{i+k}}}\right), \tag{3.6}$$

   where $\mathbf{slerp}$ denotes spherical linear interpolation and $\mathbf{I_Q}$ is the identity quaternion.

2. There is a hit going backward but not forward. This is handled in a manner similar to the previous case.

3. There are hits going forward at frame $\mathbf{M_{i+k_f}}$ and going backward at frame $\mathbf{M_{i-k_b}}$. In this case the orientation adjustments made to $\mathbf{M_{i+k_f}}$ and $\mathbf{M_{i-k_b}}$ are dissipated as in the previous two cases and combined. Let

$$\mathbf{\Delta_f} = \mathbf{slerp}\left(\alpha\left(\frac{k_f}{L_2+1}\right), \mathbf{I_Q}, \mathbf{\Delta_{Q_{i+k_f}}}\right) \tag{3.7}$$

   and

$$\mathbf{\Delta_b} = \mathbf{slerp}\left(\alpha\left(\frac{k_b}{L_2+1}\right), \mathbf{I_Q}, \mathbf{\Delta_{Q_{i-k_b}}}\right). \tag{3.8}$$

   Then E's global orientation on the current frame is premultiplied by

$$\mathbf{\Delta_{Q_i}} = \mathbf{slerp}\left(\alpha\left(\frac{k_b}{k_b+k_f}\right), \mathbf{\Delta_b}, \mathbf{\Delta_f}\right). \tag{3.9}$$

- E **is unconstrained.** Even though there are no constraints to satisfy, changes must still in general be made to E's configuration, since discontinuities will occur if plant constraints affecting E exist on $\mathbf{M_{i-1}}$ or $\mathbf{M_{i+1}}$. A discussion of this case is deferred until Section 3.2.6, which addresses the general problem of eliminating discontinuities when skeletal parameters switch from being constrained to being completely unconstrained.

3.2.4 Root Placement

The target position of a constrained end effector may be sufficiently far away that it cannot be reached even when the limb containing that end effector is at full extension. This problem can be corrected by adjusting the root position so the limb is closer to the target. Let $\mathbf{o_B}$ be the positional offset of the base of a limb (i.e., the shoulder/hip) from the root on the current frame. Also, let $\mathbf{p_R}$ be the root position, $\mathbf{p_T}$ be the target position of the end effector, and l_{max} be the length of the limb at full extension. As shown in Figure 3.6, $\mathbf{p_T}$ is reachable only if $\|\mathbf{p_T} - (\mathbf{p_R} + \mathbf{o_B})\| \leq l_{max}$ — that is, only if the root is inside a sphere of radius l_{max} centered at $\mathbf{p_T} - \mathbf{o_B}$. More generally, if n end effectors are constrained, then the root must be within the intersection of n spheres. If this is not true, then the root must be projected onto the surface of this intersection region. We perform this projection with the algorithm of Shin et al. [83], which involves a simple sequence of projections onto spheres, intersections of two spheres (which are circles), and intersections of three spheres (which are sets of discrete points).

When constraints switch on and off, the sphere intersection region changes discontinuously. If the root is projected onto the intersection region independently on each frame, the root trajectory can therefore also become discontinuous (Figure 3.7). Even small discontinuities can be quite distracting, since movement of the root affects the entire body. At the same time, attempting to generate smooth root displacements subject to sphere-interiority constraints is a non-linearly constrained variational problem which is difficult to solve efficiently. Our algorithm instead computes root positions independently on each frame and then filters these positions with a gaussian kernel of width $2L_3 + 1$, where L_3 is a user-defined constant (note that this requires knowing the target

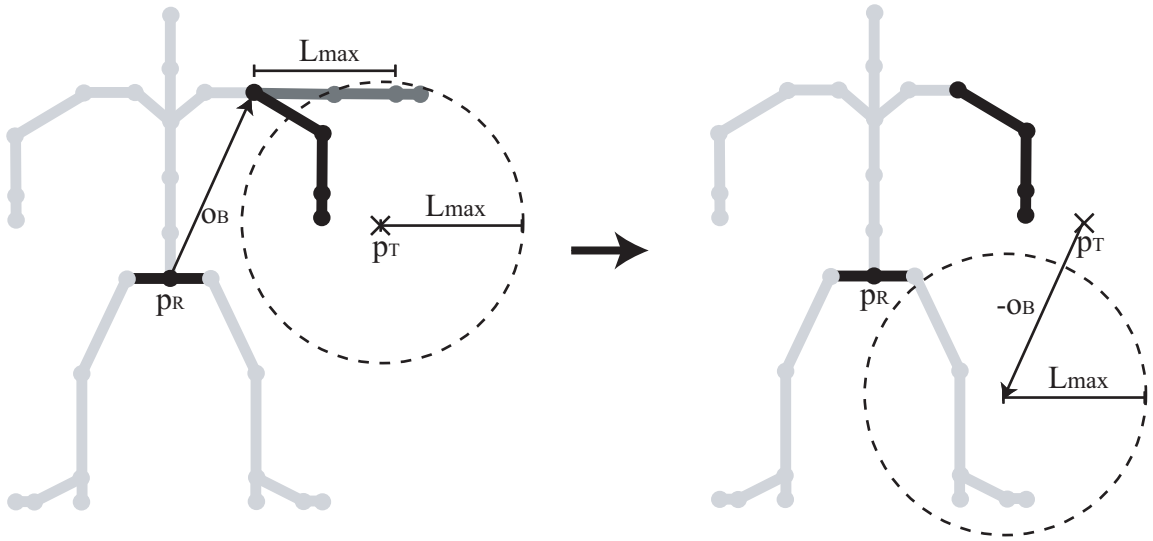

Figure 3.6: Left: For $\mathbf{p_T}$ to be reachable, the base of the limb must be within a sphere centered at $\mathbf{p_T}$ whose radius is l_{max}. **Right:** Equivalently, the root must be in a sphere of radius l_{max} centered at $\mathbf{p_T} - \mathbf{o_B}$

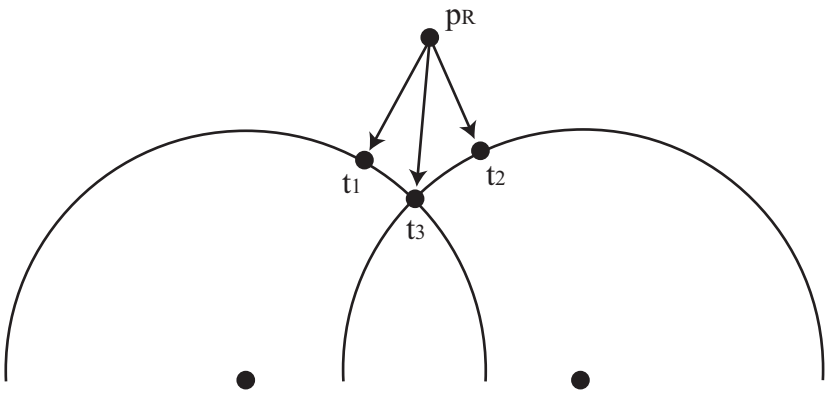

Figure 3.7: A constraint on one end effector causes the root to move to $\mathbf{t_1}$. If a second end effector were constrained instead, the root would project to $\mathbf{t_2}$. If both were constrained, the root would project to $\mathbf{t_3}$.

configurations of all constrained end effectors on frames $\mathbf{M_{i-L_3}}$ through $\mathbf{M_{i+L_3}}$). While this filtering ensures that the root path is smooth, it may also leave the root outside of the sphere intersection on some frames. However, this discrepancy is likely to be small, and end effector positions can still be exactly reached by stretching the limbs.

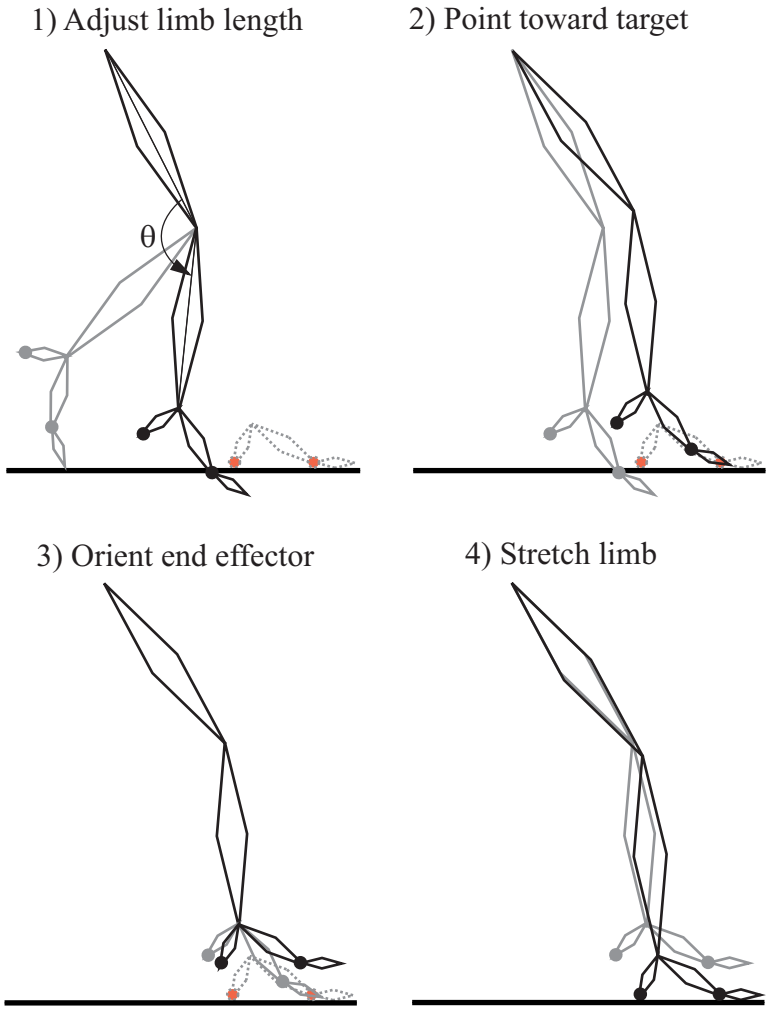

Figure 3.8: Steps in our single-limb IK algorithm.

3.2.5 Meeting the Target End Effector Configuration

Once the root position has been fixed, each constrained limb can be independently adjusted to place its end effector in the target configuration. This is accomplished with a modified version of a standard single-limb IK algorithm [89, 54] (see Figure 3.8). We first discuss the original algorithm for the case of the leg; the arm is handled similarly. Let the global positions of the hip and ankle on frame $\mathbf{M_i}$ respectively be $\mathbf{p_H}$ and $\mathbf{p_A}$, and let the target ankle position be $\mathbf{p_T}$. The first step is to rotate the knee so $\|\mathbf{p_A} - \mathbf{p_H}\| = \|\mathbf{p_T} - \mathbf{p_H}\|$. In general, the plane of rotation for the knee will not be the same as the plane defined by the thigh and shin. If l_1 and l_2 denote the length of the

upper and lower leg bones and l_1' and l_2' denote the lengths of these bones when projected onto the plane of rotation, it can be shown that the desired knee angle θ is

$$\theta = \arccos\left(\frac{l_1^2 + l_2^2 + 2\sqrt{l_1^2 - {l_1'}^2}\sqrt{l_2^2 - {l_2'}^2} - \|\mathbf{p_H} - \mathbf{p_T}\|^2}{2l_1'l_2'}\right), \tag{3.10}$$

where the solution is chosen so $0 \leq \theta \leq \pi$. See Lee and Shin [54] for a concise derivation. Once the knee has been adjusted, the ankle is in a new position $\mathbf{p}'_\mathbf{A}$. To move the ankle to the target position, the hip is next rotated so $(\mathbf{p}'_\mathbf{A} - \mathbf{p_H})$ is oriented in the same direction as $(\mathbf{p}'_\mathbf{T} - \mathbf{p_H})$, using the $\mathbf{rot}$ function defined in Section 3.2.3. Finally, the ankle is rotated to meet the target orientation.

Since the limb has 7 DOF and the target end effector configuration only specifies 6 constraints (3 for position, 3 for orientation), there is a remaining DOF that can be varied without breaking these constraints. This DOF corresponds to a rotation $\mathbf{Q}_\phi$ of the hip by ϕ degrees about the unit axis $\hat{\mathbf{n}} = \frac{\mathbf{p_T} - \mathbf{p_H}}{\|\mathbf{p_T} - \mathbf{p_H}\|}$, followed by a compensating rotation of the ankle. There are several ways of exploiting this extra DOF, such as enforcing joint limits on the hip and ankle or placing the knee as close as possible to a user-defined position [89]. Our algorithm adjusts this DOF so the ankle and hip are closer to their original orientations relative to their parent coordinate systems. Specifically, note that when $\phi = 0$ the hip is rotated as little as possible, and for some value $\phi = \phi_0$ the ankle is rotated as little as possible. A reasonable value of ϕ is then $\lambda\phi_0$ for some weight $\lambda \in [0, 1]$; in our implementation we use $\lambda = 0.5$.

The only remaining problem is to compute ϕ_0. When $\phi = 0$, let $\mathbf{\Delta Q}$ be the rotation that must be applied to the ankle so it is in the target orientation $\mathbf{Q}'_\mathbf{A}$. If the quaternion coordinates of $\mathbf{\Delta Q}$ are written as $(w, \mathbf{v})$, where w is a scalar and $\mathbf{v}$ is a 3-vector, then

$$\phi_0 = \arctan\left(\frac{w}{\hat{\mathbf{n}} \cdot \mathbf{v}}\right) \pm \pi, \tag{3.11}$$

where the sign is chosen to maximize the dot product $\mathbf{\Delta Q} \cdot \mathbf{Q}_\phi$ (treating $\mathbf{\Delta Q}$ and $\mathbf{Q}_\phi$ as ordinary 4-vectors). See Shin et al. [83] for a detailed derivation.

If the limb parameters and target end effector configurations vary continuously, then this algorithm produces a continuous sequence of adjustments. However, when the limb is nearly straight

a small change in target position can require a much larger change in knee/elbow angle. The is because limb length $L = \|\mathbf{p_A} - \mathbf{p_H}\|$ is a nonlinear function of knee/elbow angle: by the law of cosines,

$$L^2 = l_1^2 + l_2^2 - 2l_1 l_2 \cos\theta, \tag{3.12}$$

and differentiating this equation yields

$$\frac{dL}{d\theta} = \frac{l_1 l_2 \sin\theta}{L}, \tag{3.13}$$

which drops to 0 as $\theta \to \pi$. The amount that θ must be changed to produce a given change in limb length therefore grows dramatically as the starting limb length approaches its maximum value. For example, Figure 3.9 shows the knee/elbow angle adjustment necessary to extend the limb by 1% of its maximum length, as a function of the current length of the limb. The sharp rise in the graph manifests itself as an unnaturally fast extension or contraction of the knee/elbow, which we call a "pop". Even a minor pop can stand out because it affects the shape of the entire limb, and pops occur quite often because in many common motions (e.g., walking, reaching, or any motion where the character is standing) some of the limbs spend a great deal of time near full extension.

To eliminate popping artifacts, the IK algorithm is modified to limit knee/elbow rotation when the limb is near full extension. We refer to this as knee/elbow damping. Let $\rho \in (0, \pi)$ and define $f(x)$ as

$$\begin{array}{rcll} f(x) & = & 1, & x < \rho \\ & & \alpha\left(\frac{x-\rho}{\pi-\rho}\right) & \rho \le x < \pi \\ & & 0, & x \ge \pi \end{array} \tag{3.14}$$

where α is defined as in Equation 3.5. Then if the original knee/elbow angle is θ_0 and the adjustment to it is Δ_θ, the final angle θ is

$$\theta = \theta_0 + \int_{\theta_0}^{\theta_0 + \Delta_\theta} f(x)dx. \tag{3.15}$$

In our experiments we set $\rho = 135^\circ$, at which point the limb length is about 85% of its maximum value.

As a result of knee/elbow damping, the end effector may not able to reach its target position. This can be corrected by adjusting the length of the limb. If the limb length is L after rotating

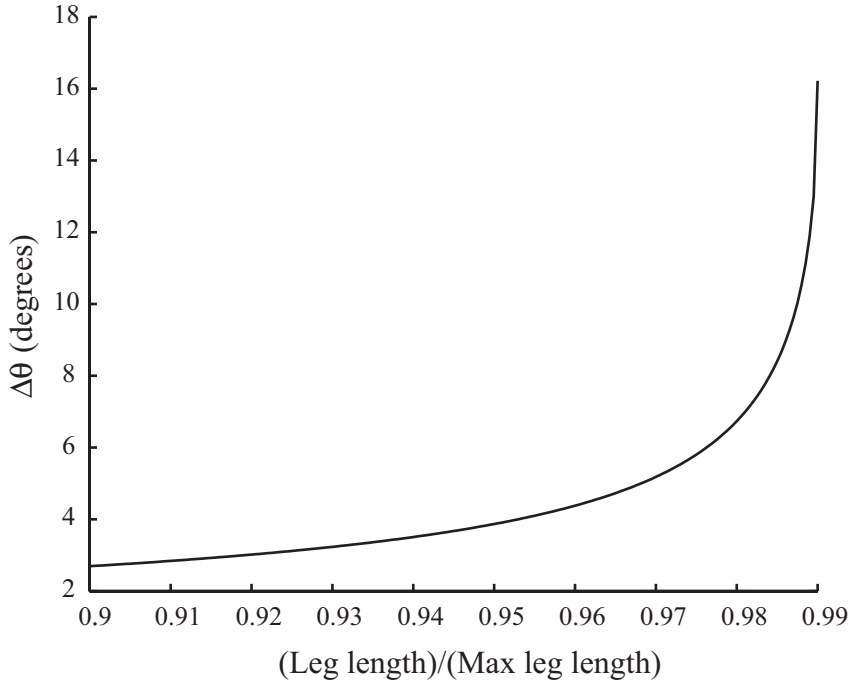

Figure 3.9: The adjustment in knee/elbow angle necessary to extend a limb by 1% of its maximum length versus the current length of the limb. Limb length is expressed as a percentage of its maximum length, and $l_1 = l_2$.

the knee/elbow and the distance from the target end effector position to the limb base is L', then both bone offsets are scaled by $\frac{L'}{L}$. This allows the end effector to be placed exactly at its target position. Note that as long as the limb is not perfectly straight, the distance from the limb base to the end effector is only affected by a fraction of the length changes applied to the bones. However, since bone lengths are adjusted only when the limb is nearly straight, almost all of the length adjustment goes directly into changing the limb length, and so usually only small amounts of scaling are necessary (no more than 3% in our experiments). Moreover, in our experience these length changes are considerably less distracting than knee/elbow popping.

3.2.6 Final Processing

The methods of Section 3.2.3–3.2.5 ensure that only smooth changes are made to a given DOF as long as it is affected by at least one constraint, but discontinuities can still occur when a DOF

switches from being constrained to being completely unconstrained or vice-versa. These discontinuities can be eliminated by blending off adjustments made to constrained DOFs into frames where they are unconstrained. We assume that all constraints have been enforced in the neighborhood $\mathbf{M_{i-L_4}}$ to $\mathbf{M_{i+L_4}}$, where L_4 is a user-defined constant. For each skeletal parameter on frame $\mathbf{M_i}$ that is not influenced by any constraints, the algorithm scans forwards and backwards L_4 frames until a frame is encountered where that parameter was adjusted to enforce a constraint. If there are no such frames in either direction, then that parameter is left unchanged. Otherwise, that parameter is adjusted as in Section 3.2.3. If the parameter is a root position or a limb scale factor, then $\mathbf{slerp}$ in Equations 3.6–3.9 is replaced by linear interpolation.

So far we have only discussed plant constraints, but it is quite common to also require each end effector to stay above a ground plane. This can be ensured with two final post-processing steps. First, each end effector with no plant constraints on $\mathbf{M_i}$ is translated vertically by the smallest amount necessary to ensure that each heel, ball, wrist, and fingertips is above or on the floor. Each end effector with one plant constraint is rotated about the plant location by the smallest amount necessary to ensure that the second attached joint is above or on the floor. The IK algorithm of Section 3.2.5 is then used to adjust the limbs accordingly. Since the motion is continuous prior to this adjustment and the floor itself is a continuous surface, these adjustments are also continuous.

The final step is to ensure that neither of the toes penetrate the floor. A positive toe rotation is defined as one that bends the toes toward the top of the foot. Our algorithm applies the smallest non-negative rotation that places each toe above the floor. Let $\hat{\mathbf{n}}$ be the unit normal defining the plane of rotation of the toes, c be the vertical distance from the position of the ball to the floor, and $\mathbf{v_T}$ be the location of the toes relative to the ball. Then what we seek is a vector $\mathbf{v'_T}$ that 1) has value c in the vertical direction (y axis), 2) has length $\|\mathbf{v_T}\|$, and 3) is orthogonal to $\hat{\mathbf{n}}$. Defining

$$\kappa = \sqrt{(\hat{n}_x^2 + \hat{n}_z^2)(\|\mathbf{v_T}\|^2 - c^2)}, \tag{3.16}$$

the solution to this system of equations is

$$\mathbf{v'_T} = (\frac{-c\hat{n}_y \mp \hat{n}_z\kappa}{\hat{n}_x^2 + \hat{n}_z^2}, c, \frac{-c\hat{n}_z \pm \hat{n}_z\kappa}{\hat{n}_x^2 + \hat{n}_z^2}), \tag{3.17}$$

where the sign is chosen such that $(\mathbf{v_T} \times \mathbf{v'_T}) \cdot \hat{\mathbf{n}}$ is positive.

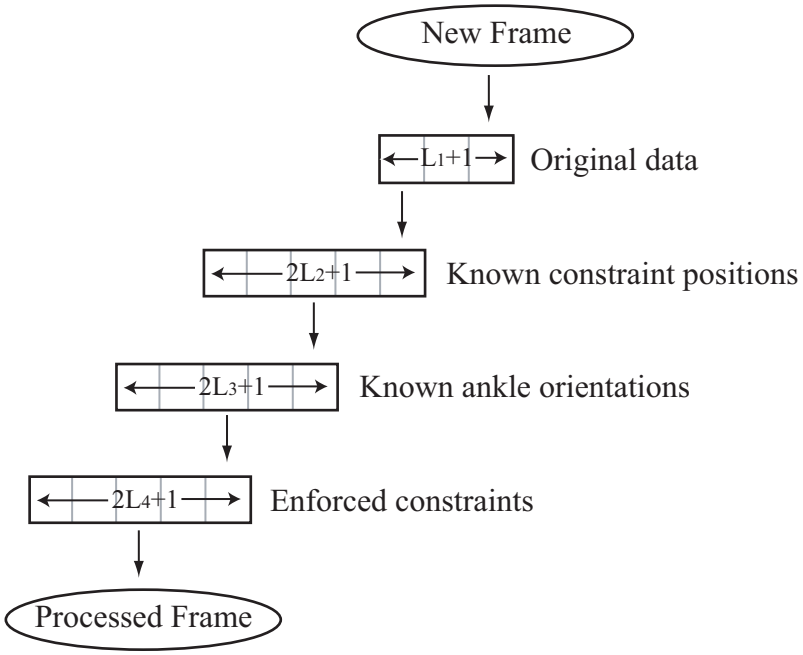

Figure 3.10: Diagram of an efficient online implementation. Four buffers are maintained to store original frame data, end effector configurations, frames with enforced constraints, and fully processed frames. Each buffer contains all the information necessary to create a single entry in the next buffer. Each time a new frame enters the top buffer, a processed frame is extracted from the bottom buffer.

3.2.7 Efficient Implementation

As shown in Figure 3.2, processing a single frame involves a calculation over a neighborhood of nearby frames. Since these neighborhoods overlap, calculations will be needlessly repeated if each frame is processed completely independently. A more efficient implementation, depicted in Figure 3.10, is to maintain a series of buffers that hold information pertaining to the different steps of the algorithm. The first buffer holds $L_1 + 1$ frames of original data. Constraint locations are extracted from this buffer and passed into a second buffer of length $2L_2 + 1$. In this buffer the ankle configuration for the central frame can be calculated using the methods of Section 3.2.3. The result is then sent into the third buffer, which holds $2L_3 + 1$ frames. The root translation for the central frame is set as in Section 3.2.4 and then the leg parameters are adjusted according to Section 3.2.5. Finally, the constraint-solved frames are placed into a fourth buffer of length $2L_4+1$,

where adjustments are blended off as described in Section 3.2.6. At the center of this final buffer is a completely processed frame.

Each buffer contains exactly the amount of information necessary to calculate a single entry in the next buffer. Hence every time a new data frame is added, one entry is added to the beginning of each buffer and removed from the end.

3.3 Results

This section present results of some experiments run on an implementation of our constraint enforcement algorithm. We set $L_1 = L_2 = L_3 = L_4 = \frac{1}{4}s$ and $\rho = 135°$. Each experiment was run on a machine with a 1.3GHz Athlon processor. Without optimizing the code beyond what was described in Section 3.2.7, the average processing time for a frame with constraints on each end effector was 1.03ms, for a frame rate of about 1000fps. In general, the average time needed to process a frame was proportional to the average number of constrained end effectors. The maximum amount that a limb was stretched or shrunk in our experiments was 3% of its original length at full extension, and the average amount of length change was less than 1%. For comparison, in a series of perceptual experiments involving two-link chains, Harrison et al. [37] found that length changes below 2.7% are typically undetectable even to viewers who are focused on finding them, and considerably larger changes can go unnoticed if the viewer's attention is focused elsewhere.

We first compared our algorithm with one in Kaydara's FILMBOX 3, a popular software package for processing motion capture data. Data from an optical motion capture system was fit to a skeleton using both FILMBOX's default settings and one with the "reach-feet" option activated, which tracks foot markers to better preserve footplants. We then annotated the feet of the default motion with plant constraints and applied our end effector cleanup algorithm. While our algorithm exactly satisfied plant constraints, some footskate remained in FILMBOX's motion, possibly due to errors in marker tracking and skeleton fitting. Figure 3.11 illustrates this by showing the height of the right ball as a function of time for each version of the motion. Also, FILMBOX's motion

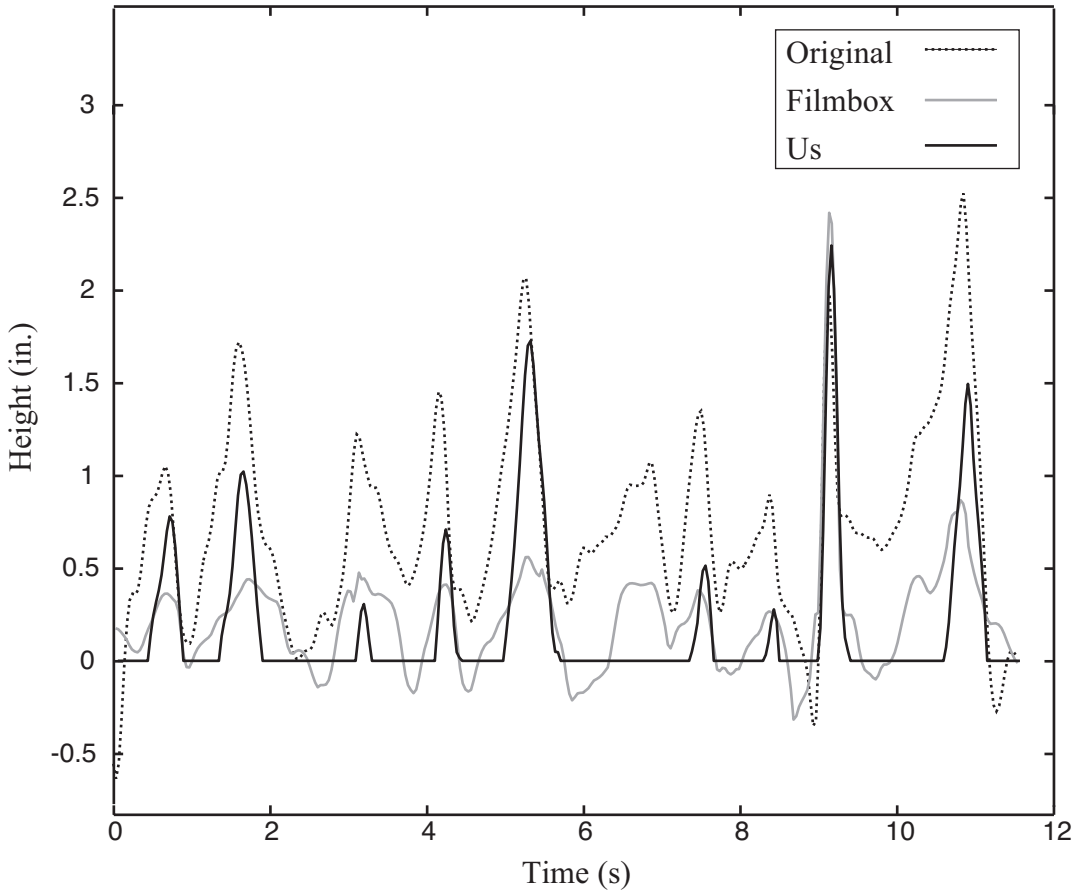

Figure 3.11: The height of the right ball in a segment of motion where the actor turned around in place several times. The dotted plot results from skeletal data generated using FILMBOX 3.0's default settings for fitting marker data to a skeleton. The grey plot is based on FILMBOX's "reach-feet" mode, which executes their method for eliminating artifacts like footskate. The black plot shows the results of applying our algorithm to the default FILMBOX fit.

exhibited some knee popping, whereas our motions was free of such artifacts. We note that our comparative success is in part due to the fact that our algorithm was supplied with the exact time intervals when footplants were to occur, whereas FILMBOX relied on measured marker positions.

We next tested our algorithm in the context of several motion editing applications. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/Cleanup/.

1. **Blend Postprocessing.** Existing methods for creating blends (typically, transitions and interpolations) [69, 96, 76] can fail to preserve important kinematic constraints, and the same

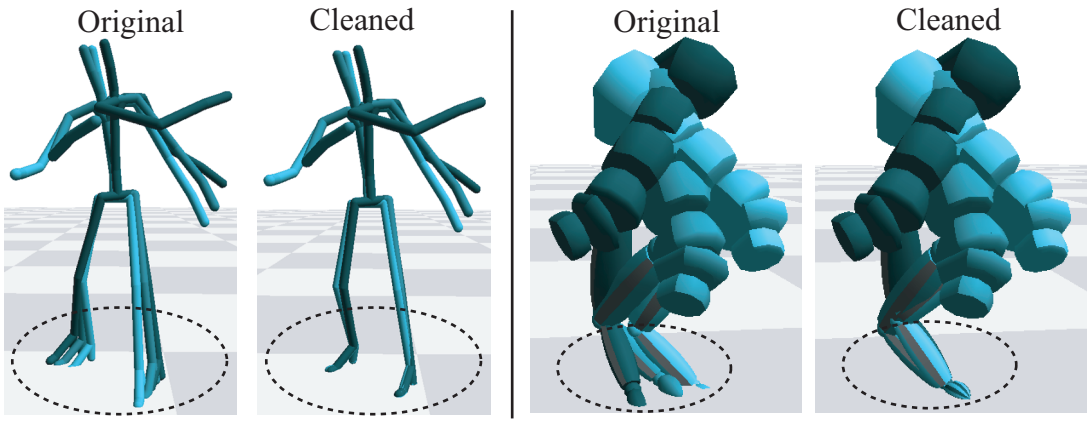

Figure 3.12: Left: A transition from walking to snatching causes the character's feet to splay apart when they should be fixed on the ground. **Right:** An interpolation of two one-handed handsprings yields a motion where the character's hand unnaturally slides across the floor. For clarity, only the torso and support arm are shown.

is true of the methods developed in this dissertation. We used our algorithm to enforce plant constraints in several transitions and interpolations; Figure 3.12 shows results for a transition from walking to snatching an object off a table and for an interpolation of two one-handed handsprings. This application of end effector cleanup will be used extensively in later chapters.

2. **Motion Salvage.** We received a motion from a professional capture studio that was damaged because the actor adjusted his pants during the shoot, thereby throwing off the calibration of some of the markers. We salvaged this motion by using our algorithm to eliminate the considerable footskate without adding any new visible artifacts. Figure 3.13 shows the height of the right ball on the damaged motion before and after applying our algorithm. We also applied our algorithm to a cartwheel motion where the hands penetrated the ground and failed to remain stationary when they were bearing the character's weight (Figure 3.13).

3. **Retargeting.** *Motion retargeting* is the transfer of an existing motion to a character with a body different than the original performer. In general, motion retargeting can be quite challenging, particularly if one wants to adapt a human motion to an object with different articulations [31]. In simpler cases, such as when the performer and character have the same

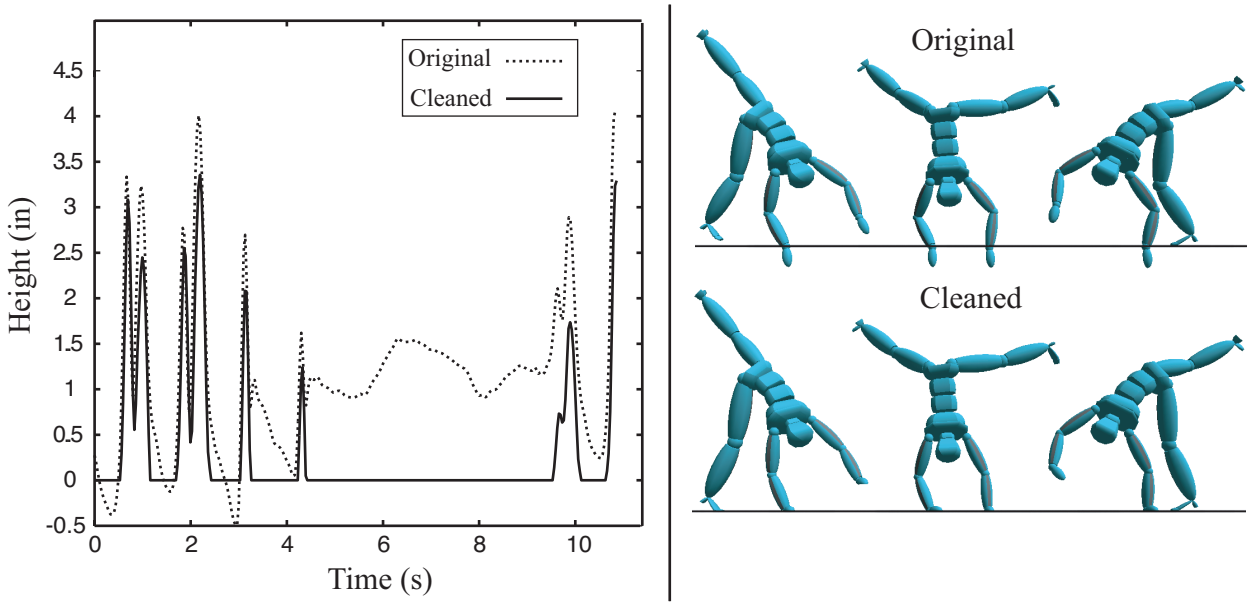

Figure 3.13: Left: The height of the right ball before and after applying our end effector cleanup algorithm. The character walked a few steps, stopped for a few seconds, and then continued walking. Note that the original motion has no clear footplants, even when the character stands still (4.2s-8.3s). **Right:** Before end effector cleanup, the character's hands penetrate the ground; afterward, they remain properly planted.

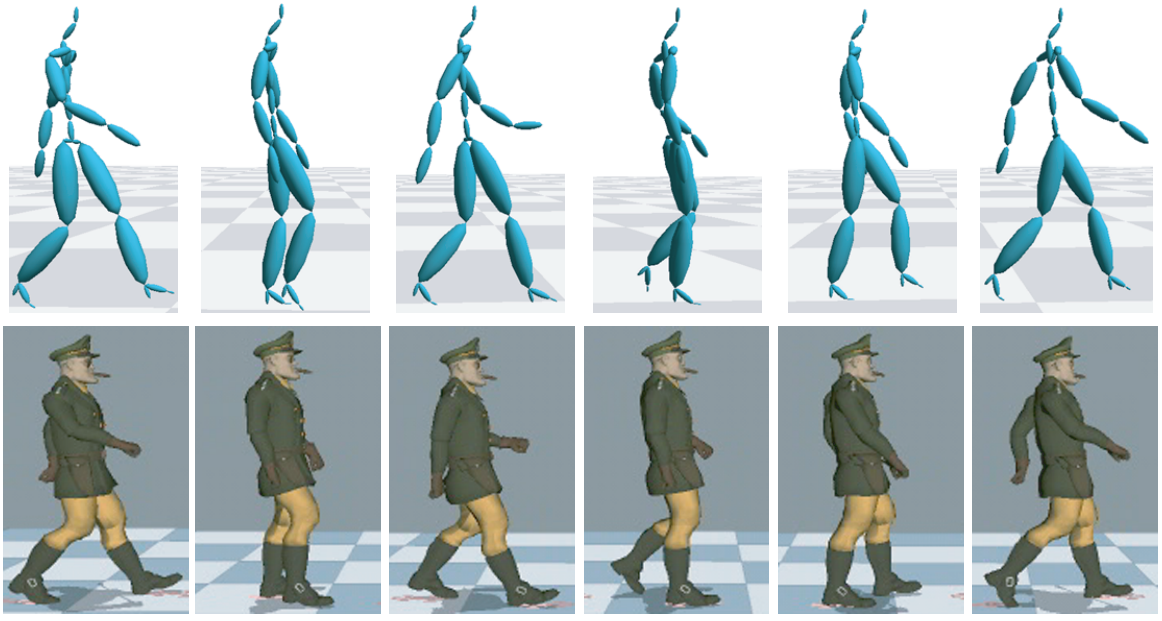

Figure 3.14: The skeleton for a walking motion and character mesh that it was retargeted to.

body topology but different proportions, our end effector cleanup algorithm can suffice. We retargeted a motion captured walk onto a cartoon character by scaling the root translation data according to differences in bone lengths and then using our end effector cleanup algorithm to satisfy footplant constraints. Figure 3.14 shows the original skeleton and the character mesh.

4. **Path Editing.** We have used our end effector cleanup algorithm to enforce footplant constraints after applying the path editing operation developed by Gleicher [32]. Our algorithm is a simple, efficient alternative to the spacetime optimization used in the original paper.

3.4 Discussion

This chapter has introduced a simple and efficient algorithm for adjusting a motion to satisfy kinematic constraints on end effectors, while at the same time ensuring that only smooth changes are made. This algorithm is not only useful on its own, but also can be employed as a postprocessing step for other algorithms that might fail to preserve kinematic constraints. In particular, this algorithm is a crucial component of the motion models developed in Chapters 4–6: by automatically enforcing end effector constraints, the range of realistic motion that can be produced by graph-based and blending-based models is greatly increased.

The methods in this chapter only guarantee C_0 continuity; it is possible that limb velocities will change discontinuously. We believe this is reasonable since in some cases (e.g., sudden impacts) velocities truly are effectively discontinuous. We have experimented with using C_1 displacement maps [45] for the interpolation methods of Sections 3.2.3 and 3.2.6, based on finite-difference estimates of the derivatives of parameter adjustments, but we did not notice a qualitative difference in the resulting motion.

The primary disadvantage of our algorithm is that it is *not* a general IK solver; we limit the types of adjustments that can be made in order to allow an analytic solution (for example, spine DOFs are not changed, and we do not consider joint limits or self-intersection). This was a conscious decision made to accommodate automatic (i.e., unsupervised) constraint enforcement, which is needed in

subsequent chapters. Specifically, since our motion models are limited to synthesizing motions that are reasonably similar to the original examples, constraint violations in synthesized motion are likely to be small. It is therefore more desirable to have a robust algorithm that will always enforce constraints smoothly than a semi-stable algorithm that exercises every available degree of freedom. For more general IK tasks, however, algorithms employing constrained nonlinear optimization methods are more appropriate [101].

The success of our algorithm was made possible by taking the somewhat unusual measure of allowing small length changes in the skeleton's bones, whereas previous work has used strictly rigid skeletons. More generally, it would be useful to understand how different artifacts — foot-skate, over-stretched limbs, sudden changes in joint orientation, and so on — may be balanced so as to produce a desired change while minimizing visual disturbance. Although recent perceptual studies [74, 37] have shed some light on this issue, considerably more work is needed in order to have a principled understanding of how different kinds of adjustments affect a viewer's evaluation of a motion.

Chapter 4

Motion Graphs

Most motions more than a few seconds in duration are naturally thought of as sequences of atomic actions. For example, dancing consists of individual movements tied to a musical beat, and navigation through a building is typically composed of steps in various directions interspersed with actions like opening doors and pushing elevator buttons. Ideally, these longer motions could be created simply by specifying which actions are needed and what order they are to occur in. By itself, however, motion capture only provides fixed clips of finite duration. For example, an individual data file might contain someone opening a door and stepping through it or walking forward two steps and then making a 90° turn to the right. If a user needs a longer motion or a motion with a different sequence of actions, then the technology offers little alternative but to capture this new motion separately.

This chapter shows how a finite set of fixed example motions can be converted into a model that allows one to controllably generate new sequences of motion of arbitrary length. The basic idea is to automatically create transition motions that seamlessly connect different parts of the data set. The result is a graph (called a motion graph) where edges correspond to motion clips and nodes indicate how these clips can attach; some of these edges will correspond to original data and others will correspond to synthesized transitions (Figure 4.1). Motion graphs are similar to *move trees* [62], which have long been used by the video game industry to control characters. Like motion graphs, move trees are graphs where edges correspond to clips, but move trees are constructed manually: given a data set, an animator identifies places where motions can connect

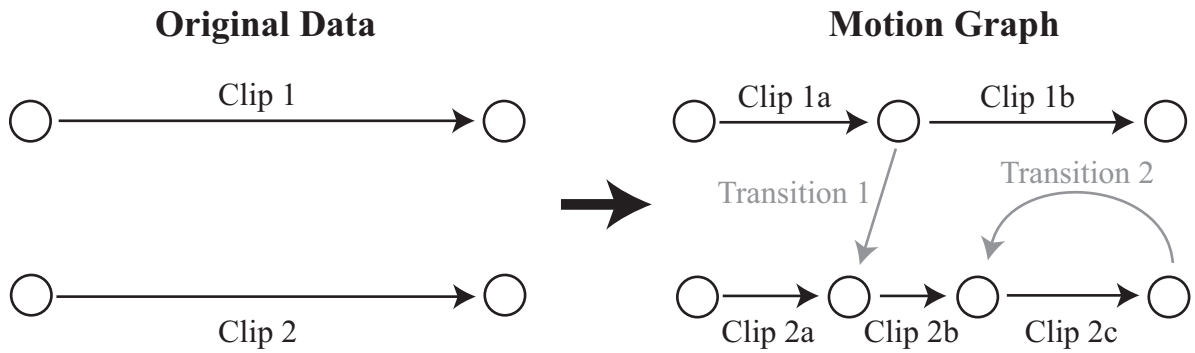

Figure 4.1: Building a motion graph from a data set of two clips. The original clips are divided into smaller segments, and transition motions are added between certain segments. Note that playing clip 1a and then clip 1b produces exactly the same motion as in the original clip 1.

together and uses software tools to adjust these motions so they will join seamlessly at these points, a tedious and time-consuming process. Our strategy is instead to automatically identify where motions are similar (as determined by an appropriate distance metric) and synthesize transitions at all such points. Restricting transitions to these places is crucial because automatic transition synthesis is reliable only when motions are reasonably similar. On the other hand, opportunistically creating transitions at *all* such places allows us to make connections between the data that might not have been obvious to the user.

Synthesizing motion with a motion graph amounts to selecting a sequence of edges, or *walk*, on the graph. However, because transitions are created opportunistically, motion graphs generally have a complicated structure and are unwieldy to work with directly. To simplify the task of extracting motion from a motion graph, we present a general search framework for extracting walks that minimize a user-defined cost function, and we apply this framework to the problem of directing very general kinds of locomotion down arbitrary paths.

To demonstrate the potential of our approach, we introduce a simple example. We were donated 78 seconds of motion capture (about 2400 frames at 30Hz) of a performer randomly walking around. Since the motion was donated, we did not carefully plan out each movement, as the literature suggests is critical to successful application of motion capture data [94]. Using this data, we constructed a motion graph and used the methods of Section 4.3 to extract motions that

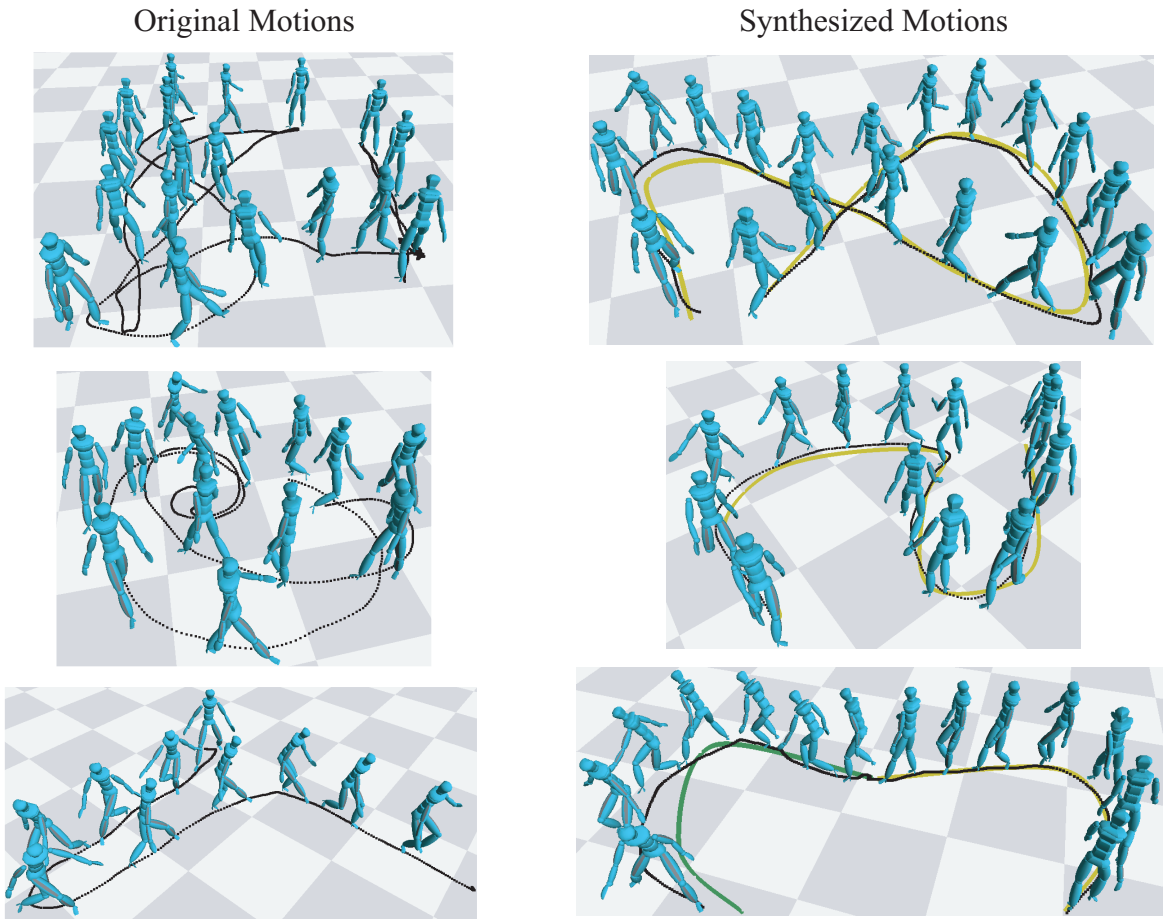

Figure 4.2: Left: Original motion capture data. The top two images show walking motions and the bottom image shows a sneaking motion. **Right:** New motions synthesized from a motion graph. The graph was built from the motions on the left and their mirror reflections. The top two images show walking motions that were fit to the yellow paths; the paths of the synthesized motions are in black. The bottom image shows a motion that was required to switch from walking to sneaking halfway down the path.

travelled along paths sketched on the ground. As seen in Figure 4.2, the actions in each synthesized motion were appropriate for the shape of the path; e.g., when the path had a sharp corner, the character executed a sharp turn. Combining the walking data with motion of a person sneaking, we were further able to control the style of the character's locomotion, requiring it to walk on certain portions of the path and sneak on others.

The remainder of this chapter is organized as follows. Section 4.1 explains how motion graphs are constructed and Section 4.2 describes a general framework for extracting motion that meets user

specifications. Section 4.3 then discusses the specific problem of generating movements that follow a path and presents experimental results. Finally, Section 4.4 concludes with a brief summary and a discussion of the scalability of our methods.

4.1 Building Motion Graphs

In addition to constraint information, in this chapter we assume that motions are annotated with descriptive labels, such as "walking" or "ballet". The annotation process can be simplified with existing automated tools [6], and in practice we have found that even a purely manual labelling is not that time-consuming (about 10-15 minutes to label a minute of data).

A motion graph is built by creating transition motions that connects different parts of a data set. The difficulty of constructing a transition is highly dependent on what motions are to be connected. In particular, generating transitions between very different motions is arguably just as difficult as generating realistic motion from scratch. Imagine, for example, creating a transition between a run and a backflip. In real life this would require several seconds for an athlete to perform, and the transition motion looks little like the motions it connects. On the other hand, if two motions are already similar then simple interpolation methods can reliably generate a transition. In light of this, our strategy is to identify places where the captured motions are already reasonably similar and then blend the motions at these points to create transitions. The remainder of this section discusses in detail how this can be done.

4.1.1 Locating Transitions

4.1.1.1 A Distance Metric for Motions

To find places where motions are similar, we introduce a distance metric for comparing two frames of motion. Since each frame of data may be thought of as a vector containing the position of the root and the orientation of each joint, a simple approach is to compute a weighted L_p norm of these vectors. However, this metric fails to address several important issues:

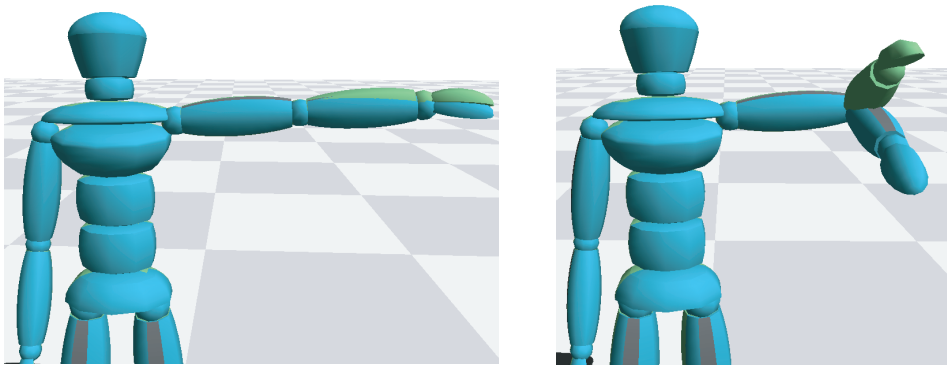

Figure 4.3: Twisting the left shoulder by 30° has a different impact on body shape when the arm is straight (left) than when it is bent (right). The original pose is in blue and the adjusted pose is in green.

1. The motion at a particular frame is determined not just by body posture, but also by joint velocities, accelerations, and higher-order derivatives.

2. Some joint orientations have more impact on the overall shape of the body than others (e.g., hip orientation vs. wrist orientation). Moreover, there is no simple way of assigning fixed weights to different joints because the effect of perturbing one joint orientation in general depends on other joint orientations (Figure 4.3).

3. A motion is fundamentally unchanged if it is translated within the floor plane or rotated about the vertical axis, since this amounts to viewing the motion with a different camera[1]. In light of this, any distance metric should be invariant under rigid 2D transformations of the input motions.

Our distance metric accounts for each of these matters. Instead of directly comparing joint orientations, our strategy is to attach markers to the joints and then compare the resulting point

[1]Movement out of the floor plane, however, *does* represent a fundamental change, due to the influence of gravity. Also, the *meaning* of a motion may be tied to its global position and orientation (e.g., if character leans against a tree, then it must be near the tree), but one can always create a different environment where the exact same motion makes sense under a different global position and orientation (e.g., create a new tree wherever the character is leaning).

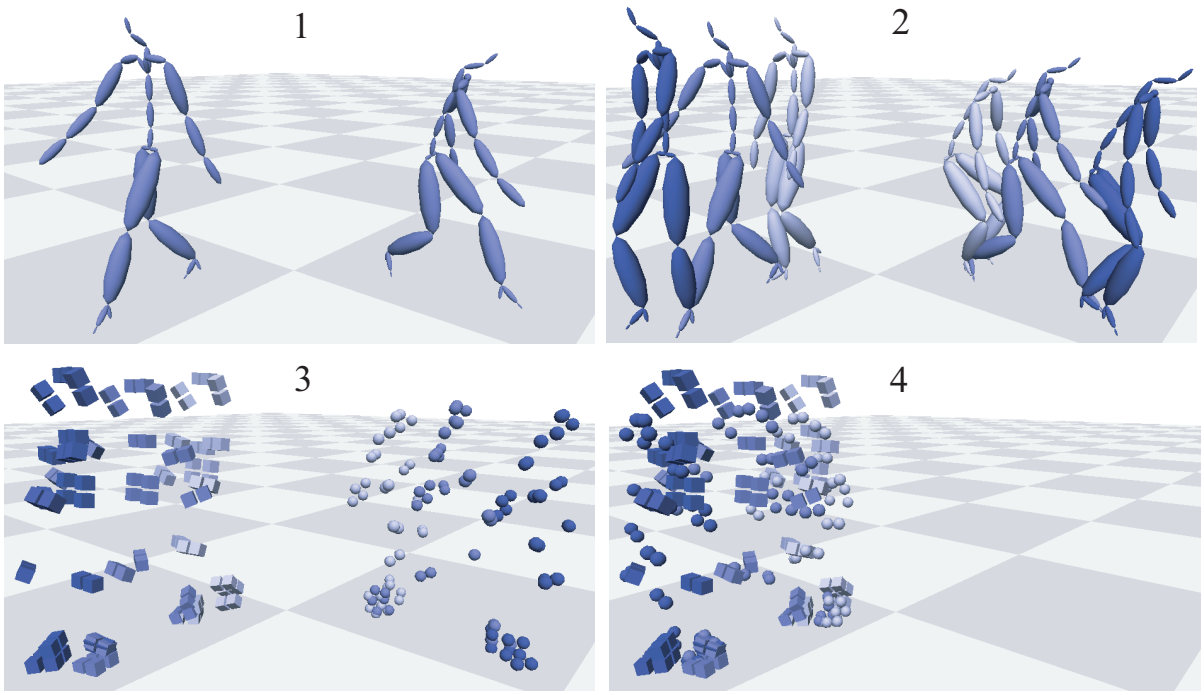

Figure 4.4: To compute the distance between two frames of motion (1), local neighborhoods of frames are extracted (2) and converted into point clouds (3). The squared distance between corresponding points is then calculated through an optimization that factors out rotation about the vertical axis and translation in the floor plane (4).

clouds. Intuitively, this is analogous to placing dots on the surface of a person's skin and comparing the movement of these dots. Formally, computation of the distance $D(\mathbf{M_i}, \mathbf{M'_j})$ between two frames $\mathbf{M_i}$ and $\mathbf{M'_j}$ proceeds in three steps (Figure 4.4):

1. Extract neighborhoods of $2L + 1$ frames centered around $\mathbf{M_i}$ and $\mathbf{M'_j}$. This has the effect of incorporating derivative information, with higher values of L including higher-order derivatives. We set L so the frame windows are the same length as a transition ($\frac{1}{2}$s in our implementation), which ensures that the value of $D(\mathbf{M_i}, \mathbf{M'_j})$ is influenced by each frame affected by the transition.

2. Attach markers to each joint. In our implementation these markers are rigidly attached, e.g., they remain at a fixed offset in the joint's local coordinate system. However, other methods could be used to position markers based on skeletal pose.

3. If there are n_m markers total, then at this stage there are two point clouds with $n_p = n_m(2L+1)$ points each. The distance between these two points clouds is defined as the sum of squared Euclidean distance between corresponding points, minimized over all translations in the floor plane and rotations about the vertical axis:

$$D(\mathbf{M_i}, \mathbf{M'_j}) = \min_{\theta, x_0, z_0} \sum_{k=1}^{n_p} w_k \|\mathbf{p_k} - \mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}} \mathbf{p'_k}\|^2, \tag{4.1}$$

where $\mathbf{p_k}$ and $\mathbf{p'_k}$ denote the k^{th} point in the point clouds for $\mathbf{M_i}$ and $\mathbf{M'_j}$, and $\mathbf{T}_{\theta,\mathbf{x_0},\mathbf{z_0}}$ is a rigid transformation composed of a rotation by θ degrees about the y (vertical) axis followed by a translation of (x_0, z_0) in the floor plane. The scalars w_k can be used to preferentially weight different markers, and we assume that $\sum_{k=1}^{n_p} w_k = 1$. In our implementation we weighted all markers from the same frame equally, and a triangular kernel was used to weight markers on central frames more heavily than markers toward the edges of the window.

The optimization in step 3 has a closed-form solution, which may be derived by setting the gradient of the objective in Equation 4.1 to zero and solving the resulting system of equations. The optimal coordinate transformation is

$$\theta = \arctan \frac{\sum_k w_k(x_k z'_k - x'_k z_k) - (\sum_k w_k x_k \sum_k w_k z'_k - \sum_k w_k x'_k \sum_k w_k z_k)}{\sum_k w_k(x_k x'_k + z_k z'_k) - (\sum_k w_k x_k \sum_k w_k x'_k + \sum_k w_k z_k \sum_k w_k z'_k)} \tag{4.2}$$

$$x_0 = \sum_k w_k x_k - \cos\theta \sum_k w_k x'_k - \sin\theta \sum_k w_k z'_k \tag{4.3}$$

$$z_0 = \sum_k w_k z_k + \sin\theta \sum_k w_k x'_k - \cos\theta \sum_k w_k z'_k, \tag{4.4}$$

and the corresponding distance can be found by substituting these values into Equation 4.1.

Using this distance metric, each motion in the data set is compared against each other motion, including itself. The distance between every pair of frames is computed, forming a sampled 2D function such as the one shown in Figure 4.5, and local minima are identified by comparing the value at each point with the values of its eight neighbors. These local minima correspond to "sweet spots" at which transitions are locally the most opportune, and hence they serve as candidate

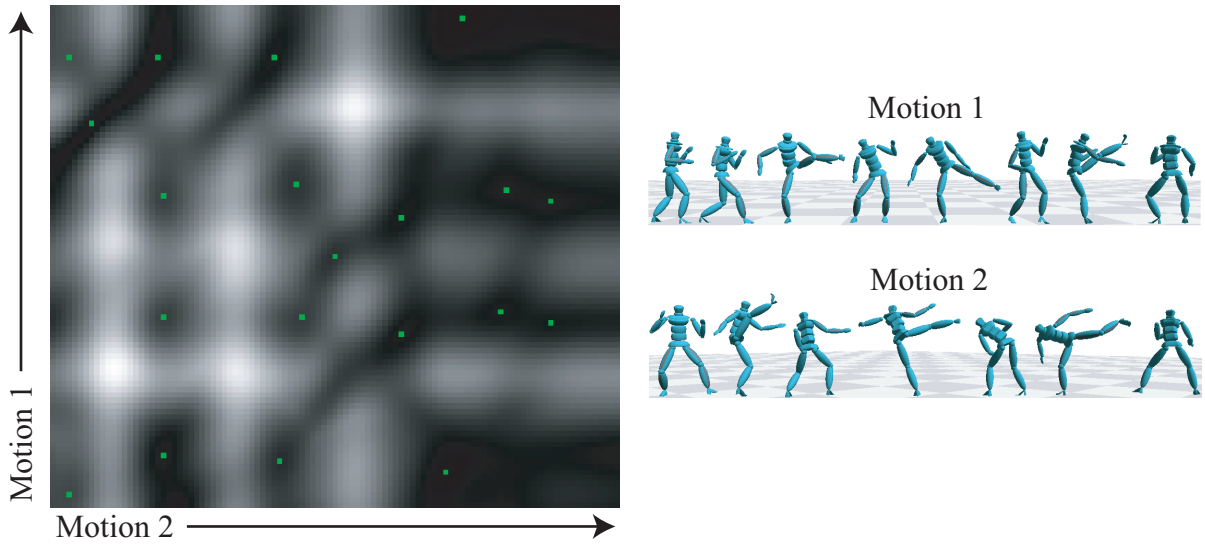

Figure 4.5: A distance function for two motions. The entry at cell (i, j) contains the distance between frame i of the first motion and frame j of the second motion. Darker values correspond to smaller distances, and green cells indicate local minima (in the 8-neighbor sense). The smoothness stems from the continuity of the original motions and the fact that calculating the distance between two frames involves a computation over the surrounding neighborhoods.

transition points. A similar strategy was used by Schödl et al. [82] to create graph structures for connecting different segments of video.

4.1.1.2 Computing Distances Efficiently

The computations in Equations 4.1–4.4 contain summations of expressions that involve point positions in windows of frames centered on $\mathbf{M_i}$ and $\mathbf{M'_j}$. These windows overlap for nearby pairs of frames, and hence there is considerable redundant calculation if each pair of frames is processed independently. This redundancy can be made clear by rewriting equations Equations 4.1–4.4. Let $\mathbf{f_{r,s}} = (x_{r,s}, y_{r,s}, z_{r,s})$ be the location of marker s on frame $\mathbf{M_r}$ and let $\mathbf{f'_{r,s}}$ be defined similarly. Also, let each weight w_k in Equations 4.1–4.4 be the product of two weights σ_s and ρ_r, the former associated with marker index (so $s \in [1, n_m]$) and the latter associated with the frame index within the point cloud (so $r \in [-L, L]$). We assume $\sum_{s=1}^{n_m} \sigma_s = 1$ and $\sum_{r=-L}^{L} \rho_r = 1$. Equation 4.1 can

then be rewritten as

$$D(\mathbf{M_i}, \mathbf{M'_j}) = \min_{\theta, x_0, z_0} \sum_{r=-L}^{L} \rho_r \sum_{s=1}^{n_m} \sigma_s \|\mathbf{f_{i+r,s}} - \mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}} \mathbf{f'_{j+r,s}}\|^2. \tag{4.5}$$

Defining

$$\overline{\alpha_r} = \sum_{s=1}^{n_p} \sigma_s \alpha_{r,s}, \tag{4.6}$$

where $\alpha_{r,s}$ is any expression involving the coordinates of marker s on frames $\mathbf{M_{i+r}}$ and $\mathbf{M'_{j+r}}$, this equation can be expanded into

$$D(\mathbf{M_i}, \mathbf{M'_j}) = \min_{\theta, x_0, z_0} x_0^2 + z_0^2 + \sum_{r=-L}^{L} \rho_r \left(\overline{\|\mathbf{f_r}\|^2} + \overline{\|\mathbf{f'_r}\|^2} + 2\left(\overline{a_r} \cos\theta + \overline{b_r} \sin\theta + \overline{c_r} \right) \right), \tag{4.7}$$

where

$$\begin{aligned}
\overline{a_r} &= -\overline{(x_r x'_r + z_r z'_r)} + x_0 \overline{x'_r} + z_0 \overline{z'_r} \\
\overline{b_r} &= -\overline{(x_r z'_r - z_r x'_r)} + x_0 \overline{z'_r} - z_0 \overline{x'_r} \\
\overline{c_r} &= -\overline{(y_r y'_r)} - x_0 \overline{x_r} - z_0 \overline{z_r}.
\end{aligned}$$

The indices i and j have been suppressed for compactness; for example, $\overline{x_r}$ is used in place of $\overline{x_{i+r}}$, which is the weighted average of the x coordinate of each marker on frame $\mathbf{M_{i+r}}$. Similarly, the equations for θ, x_0, and z_0 can be written as

$$\theta(\mathbf{M_i}, \mathbf{M'_j}) = \arctan \left(\frac{\sum_r \rho_r \overline{(x_r z'_r - z_r x'_r)} - \left(\sum_r \rho_r \overline{x_r} \sum_r \rho_r \overline{z'_r} - \sum_r \rho_r \overline{z_r} \sum_r \rho_r \overline{x'_r} \right)}{\sum_r \rho_r \overline{(x_r x'_r + z_r z'_r)} - \left(\sum_r \rho_r \overline{x_r} \sum_r \rho_r \overline{x'_r} + \sum_r \rho_r \overline{z_r} \sum_r \rho_r \overline{z'_r} \right)} \right) \tag{4.8}$$

$$x_0(\mathbf{M_i}, \mathbf{M'_j}) = \sum_r \rho_r \overline{x_r} - \left(\sum_r \rho_r \overline{x'_r} \right) \cos\theta - \left(\sum_r \rho_r \overline{z'_r} \right) \sin\theta \tag{4.9}$$

$$z_0(\mathbf{M_i}, \mathbf{M'_j}) = \sum_r \rho_r \overline{z_r} + \left(\sum_r \rho_r \overline{x'_r} \right) \sin\theta - \left(\sum_r \rho_r \overline{z'_r} \right) \cos\theta, \tag{4.10}$$

where again we have suppressed the i's and j's.

Each of Equations 4.7–4.10 contains sums over $2L + 1$ elements. Each sum computed for the frame pair $(\mathbf{M_{i+\Delta}}, \mathbf{M'_{j+\Delta}})$ contains $\max(2L + 1 - |\Delta|, 0)$ elements of the corresponding sum

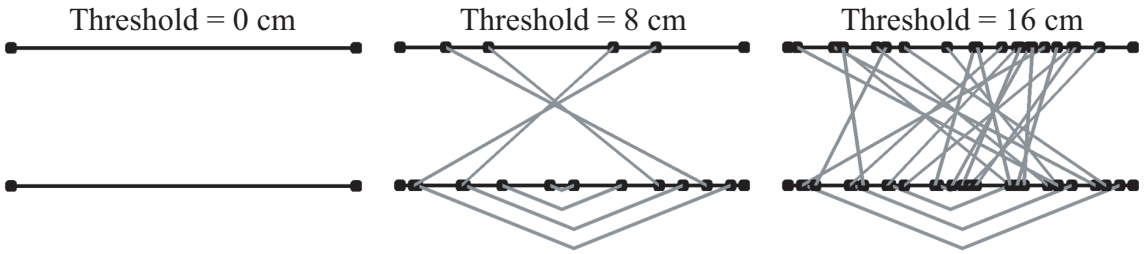

Figure 4.6: Motion graphs resulting from different distance thresholds, starting from a data set of two motions. The black lines represent the original motions and the gray lines represent transitions. The thresholds are stated in terms of $\sqrt{D}$.

computed for the frame pair $(\mathbf{M_i}, \mathbf{M'_j})$. To eliminate this redundancy, the quantities $\overline{x}$, $\overline{z}$, and $\overline{\|\mathbf{f}\|^2}$ can be precomputed for each frame in the data set, and $\overline{(xx' + zz')}$, $\overline{(xz' - zx')}$, and $\overline{(yy')}$ can be precomputed for each pair of frames. This reduces the number of operations needed to evaluate the right hand sides of Equations 4.7–4.10 for all frame pairs by about a factor of $2L + 1$, which in our experiments was over an order of magnitude savings in computation time.

4.1.1.3 Selecting Transition Points

Transitions are created at local minima whose value is below a user-defined threshold. By varying the threshold, a user can determine an acceptable tradeoff between better guarantees on motion quality (lower thresholds) and higher transition density (higher thresholds). Figure 4.6 shows some motion graphs resulting from different thresholds.

Different types of motion will require different thresholds. For example, we have found that transitions between walking motions require considerably lower thresholds than transitions involving martial arts or gymnastic motions. This probably stems from the fact that an average person sees other people walk on a daily basis, but views motions like kicks or flips considerably less frequently. Hence transitions among familiar motions (like walking) must have higher fidelity than transitions involving unusual motions (like gymnastic maneuvers). We allow users to set different thresholds based on the descriptive annotations attached to the motion data.

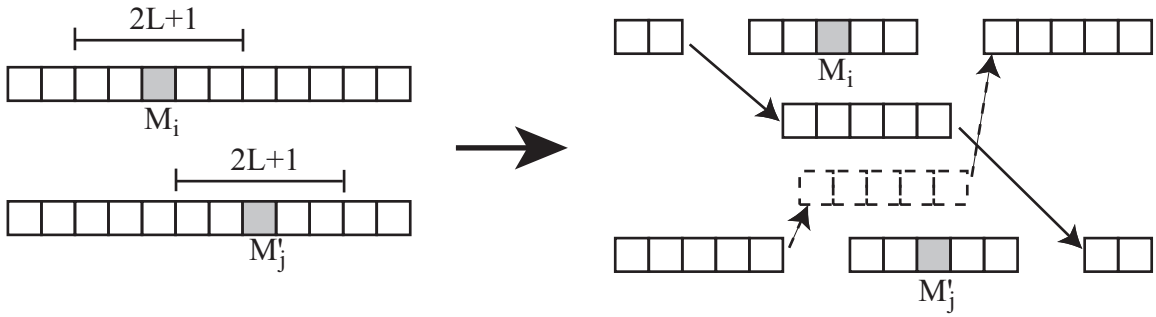

Figure 4.7: Adding transitions about the local minimum $D(\mathbf{M_i}, \mathbf{M'_j})$ ($L = 2$). Note that one transition is from $\mathbf{M}$ to $\mathbf{M}'$ (solid lines) and the other is from $\mathbf{M}'$ to $\mathbf{M}$ (dashed lines).

4.1.2 Creating Transitions

If $D(\mathbf{M_i}, \mathbf{M'_j})$ is a local minimum below the distance threshold, then two transitions can be created about this pair of frames, one from $\mathbf{M}$ to $\mathbf{M}'$ and the other from $\mathbf{M}'$ to $\mathbf{M}$ (Figure 4.7). We discuss the former, and the latter is handled similarity. The transition motion spans $2L + 1$ frames and is created by blending $\mathbf{M_{i-L}}, \ldots, \mathbf{M_{i+L}}$ with $\mathbf{M'_{j-L}}, \ldots, \mathbf{M'_{j+L}}$, where $\mathbf{M}'$ is assumed to have been adjusted by the coordinate transformation specified in Equations 4.2–4.4 so it is aligned with $\mathbf{M}$ at the transition point. The root position $\widetilde{\mathbf{p}}$ during the transition is a linear interpolation of the root positions $\mathbf{p}$ from $\mathbf{M}$ and the root positions $\mathbf{p}'$ from $\mathbf{M}'$:

$$\widetilde{\mathbf{p}}_\mathbf{k} = \alpha\left(\frac{k+1}{2L+2}\right)\mathbf{p_{i-L+k}} + \left(1 - \alpha\left(\frac{k+1}{2L+2}\right)\right)\mathbf{p'_{j-L+k}}, \tag{4.11}$$

where $k \in [0, 2L]$ and α is defined as in Chapter 3:

$$\alpha(t) = 2t^3 - 3t^2 + 1 \tag{4.12}$$

Similarly, each joint orientation $\widetilde{\mathbf{q}}$ in the transition is a spherical linear interpolation of the corresponding joint orientations $\mathbf{q}$ in $\mathbf{M}$ and $\mathbf{q}'$ in $\mathbf{M}'$:

$$\widetilde{\mathbf{q}}_\mathbf{k} = \mathbf{slerp}\left(\alpha\left(\frac{k+1}{2L+2}\right), \mathbf{q'_{j-L+k}}, \mathbf{q_{i-L+k}}\right). \tag{4.13}$$

Note that this interpolation scheme preserves C^1 continuity since α is itself C^1 continuous. Higher-order continuity could trivially be achieved with different α's, but we have found C^1 continuity to be sufficient in practice.

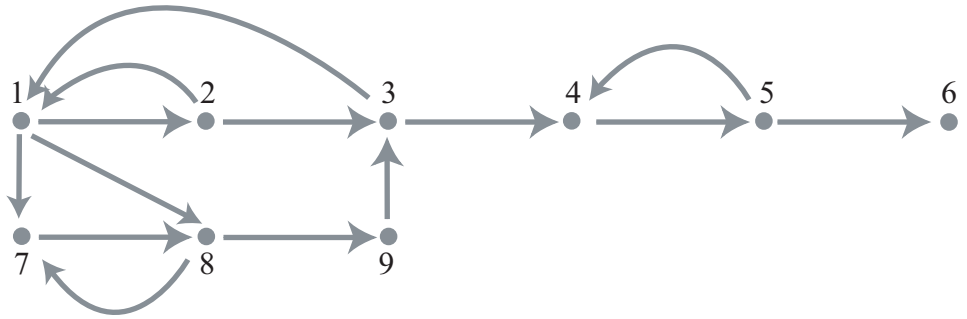

Figure 4.8: A simple motion graph. The largest strongly connected component is $[1, 2, 3, 7, 8, 9]$. Nodes 4 and 5 are sinks and 6 is a dead end.

The use of linear blends means that kinematic constraints such as footplants may be violated. To correct this, we infer constraints on the transition from constraint annotations in the original motions. Treating constraints as binary signals and blending them in analogy to Equations 4.11 and 4.13 amounts to using the constraints from $\mathbf{M}$ in the first half of the transition and the constraints from $\mathbf{M}'$ in the second half. These constraints may be satisfied as a postprocessing step once motions are extracted from the graph; we use the method described in Chapter 3. Descriptive labels attached to the original motions are also carried into transitions: a blend frame that combines a frame from $\mathbf{M}$ with label set S_1 and a frame from $\mathbf{M}'$ with label set S_2 has the union of these labels $S_1 \cup S_2$.

4.1.3 Pruning the Graph

If no corrective action is taken, then a motion graph may contain *dead end* nodes, which are nodes that are not part of any cycle (Figure 4.8). Once such a node is entered, there is a bound on how much additional motion can be generated. Other nodes (called *sinks*) may be part of one or more cycles but nonetheless only be able to reach a small fraction of the total number of nodes in the graph. While arbitrarily long motion may still be generated once a sink is entered, this motion is confined to a small part of the data set. Finally, some nodes may have incoming edges with labels that are not matched by any outgoing edge. This is dangerous since logical discontinuities

may be forced into a motion. For example, a character currently in a "boxing" motion may have no choice but to transition to a "ballet" motion.

To eliminate these problem nodes, the motion graph is pruned such that, starting from any edge, it is possible to generate arbitrarily long streams of motion of the same type while also having access to as much of the data set as possible. This is done as follows. For each descriptive label in the data set, a subgraph is formed from all edges whose frames contain this label. The strongly connected components (SCCs) of this subgraph are then computed, where a SCC is a maximal set of nodes such that there is a connecting graph walk for any ordered pair of nodes. The SCCs can be computed in time linear in the number of edges and nodes using Tarjan's algorithm [21]. Any edge that does not attach two nodes in the largest SCC of at least one subgraph is discarded, and nodes which have all of their edges removed are similarly eliminated.

A warning is given to the user if the largest SCC for a given label contains below a threshold number of frames of data. Also, a warning is given if for any ordered pair of SCCs there is no way to transition from the first to the second. In either case, the user may wish to adjust the transition thresholds (Section 4.1.1.3) to give the motion graph greater connectivity.

4.1.4 Results and Discussion

While we constructed several motion graphs of varying sizes, we only report results for the largest motion graph, since it included all of the motion data used in our experiments. This graph was constructed from 6200 frames of motion, or 3 minutes and 36 seconds of data sampled at 30Hz. On a machine with a 1.3GHz Athlon processor, 44.5s were needed to find all local minima of D ($L = 7$, $n_m = 36$). The data set contained three kinds of motion: walking, sneaking, and martial arts movements such as kicking and punching. Separate transition thresholds were set for each of the 9 possible style-to-style transitions (e.g., walking to walking, walking to sneaking, etc.); a simple GUI allowed a user to see how graph connectivity varied with each threshold. Approximately five minutes of user time were needed to select the thresholds, and 0.8 seconds of processor time were needed to compute transition motions and prune the motion graph. The total size on disk of the local minima (including frame indices, value of D, and the aligning coordinate

transformation) was 2.1MB, or 83% of the size of the motion data. See Section 4.4 for a discussion regarding the scalability of graph construction.

Since transitions can only occur at local minima of D, altering the frame distance computation will in general yield different graphs. We have experimented with different settings for the following parameters:

1. Neighborhood widths (between 0.1s and 0.75s)

2. Weights ρ for different frames in the neighborhood (we used box, triangle, and gaussian weight functions)

3. Number of markers per joint (1 will only capture joint position while 2 or 3 will also include orientation information)

4. Weights σ for different joints (we used both uniform weights and only having nonzero weight on the shoulders, elbows, hips, knees, pelvis and spine, as suggested by Lee et al. [52]).

While individual transitions may be lost or gained as the minima of D are altered (in particular, longer neighborhood widths lead to smoother distance grids with fewer local minima), we have found the general distribution of transitions to be largely insensitive to the specific choice of parameters. Moreover, we have found that a typical transition will change position by only a frame or two, which results in blends that are visually nearly indistinguishable. This seems to contradict the results of Wang and Bodenheimer [93], who sought to compute optimal weights for the metric of Lee et al. [52]. They discovered that the globally best transition point between two motions can change dramatically if joint weights are varied. However, we are interested not in a single globally optimal transition, but rather a set of locally optimal transitions — in our experience, the *values* of these local minima will fluctuate if the parameters of D are changed (and hence the global optimum may shift), but their *locations* are quite stable. We speculate that alternative distance metrics (i.e., those of Arikan and Forsyth [5] and Lee et al. [52]) are similar in this regard, although we have not conducted experiments.

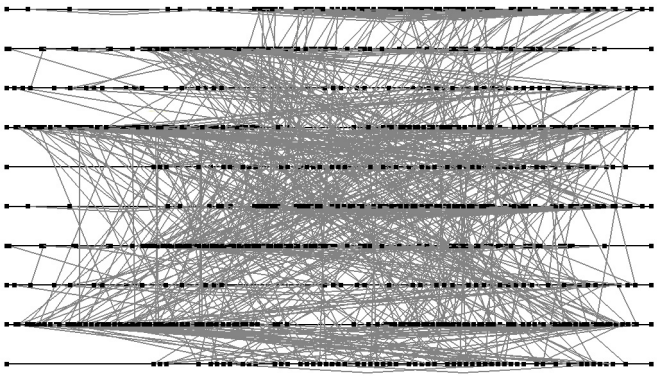

Figure 4.9: A motion graph constructed from a minute of data (about 1800 frames at 30Hz).

4.2 Using Motion Graphs

Any walk on a motion graph can be converted into a continuous motion by applying the appropriate coordinate transformation to each edge's clip and concatenating them. It is assumed that each transition clip stores the coordinate transformation that aligns the motion segment that follows it with the one that precedes it. Kinematic constraints on the reconstructed motion are enforced using the algorithm discussed in Chapter 3.

Motion graphs can have a complicated internal structure that is unwieldy to work with directly (Figure 4.9), so it is important that users have high-level tools for building graph walks. The simplest approach is to build random walks, but the arbitrary motion that this would generate is of limited practical use beyond an elaborate screen saver. Another possibility is to employ shortest-path graph algorithms to find graphs walks that minimize metrics such as time elapsed or distance travelled. While this would allow users to build motions that connect two given clips, it is less useful than it might appear at first. First, there are no guarantees that the shortest graph walk is short in an absolute sense. For example, in our larger test graphs (which contained several thousand nodes) the duration of the minimal-time walk between any two nodes was on the order of two and a half seconds. This is not because the graphs were poorly connected — since the transitions were about half of a second each, on average only four or five transitions separated any two of the thousands of nodes. A second and more important problem is that there is no control over

what happens during the generated motion. One cannot, for example, specify relatively simple properties like what direction the character travels in or where it ends up.

To provide users with more control in extracting graph walks, we cast motion synthesis as an optimization problem. Given an objective function and hard constraints, our algorithm attempts to find a graph walk that minimizes the objective while satisfying the constraints. For all but the most trivial objective functions, there is no efficient algorithm for extracting the globally optimal walk, so we instead employ search methods to find a satisfactory walk within a reasonable amount of time. The remainder of this section describes the search algorithm and discusses issues in selecting good optimization criteria.

4.2.1 Searching For Motion

To build a graph walk, the user supplies a scalar objective function $g(\mathbf{w}, \mathbf{e})$ that evaluates the error accrued by appending an edge $\mathbf{e}$ to existing graph walk $\mathbf{w}$, which may be the empty walk $\emptyset$. The total error $f(\mathbf{w})$ of a graph walk is then

$$f(\mathbf{w}) = f([\mathbf{e_1}, \ldots, \mathbf{e_n}]) = \sum_{i=1}^{n} g([\mathbf{e_1}, \ldots, \mathbf{e_{i-1}}], \mathbf{e_i}), \tag{4.14}$$

where $\mathbf{w}$ is comprised of the edges $\mathbf{e_1}, \ldots, \mathbf{e_n}$. It is assumed that $g(\mathbf{w}, \mathbf{e})$ is nonnegative, which means that total error can never be decreased by adding an edge to a graph walk. The user must also supply a halting condition indicating when no additional edges should be added to the walk; a walk satisfying this condition is said to be *complete*. The start of the walk may either be specified by the user or chosen by the search algorithm. Finally, the user can specify hard constraints to restrict the search space. For example, a graph walk may be prohibited from causing the character to intersect with an object or from using particular kinds of motion.

A simple way of optimizing f is to generate all complete graph walks with depth-first search and then select the one with minimal cost. Illegal motions can be pruned by terminating a branch of the search whenever an edge is encountered that, when appended to the current walk, violates the constraints. Unfortunately, this algorithm is infeasible for constructing longer motions because the number of possible graph walks grows exponentially with the average size of a complete graph

walk. A considerable increase in efficiency can be gained by exploiting the fact that $f(\mathbf{w})$ can never decrease as a result of adding edges to $\mathbf{w}$. The algorithm keeps track of the current best complete graph walk $\mathbf{w_{opt}}$ and immediately halts any branch of the search for which the current error exceeds $f(\mathbf{w_{opt}})$; this strategy is sometimes referred to as branch and bound. The user may also define a threshold error ϵ such that if $f(\mathbf{w}) < \epsilon$ and $\mathbf{w}$ is complete, then the search is immediately halted.

Branch and bound is most successful when a tight lower bound is obtained early in the search process, and it is therefore advantageous to order the search so lower-error walks are likely to be explored first. We use a simple greedy heuristic: given a set of unexplored child edges $\mathbf{e_1}, \ldots, \mathbf{e_n}$, we start with the one that minimizes $g(\mathbf{w}, \mathbf{e_i})$.

While branch and bound reduces the number of graph walks that must be evaluated, it does not change the fact that the search process is inherently exponential; it merely lowers the effective branching factor. For this reason we generate the desired graph walk incrementally. At each step branch and bound is used to find an optimal graph walk of n frames. The first m frames of this walk are retained and the final retained node is used as the starting point for another search, and this process continues until a complete graph walk is generated. This technique is known in the search literature as beam search. In our implementation we used values of n from 60 to 120 frames (2 to 4 seconds) and m from 25 to 30 frames (about one second).

Sometimes it is useful to have a degree of randomness in the search process, such as when one is animating a crowd. One simple way of adding randomness is to select the starting point(s) for the search at random. Another simple method is retain the r best graph walks at the end of each search iteration and randomly pick among the ones whose error is within some tolerance of the best solution.

4.2.2 Deciding What To Ask For

Since the motion extracted from the graph is determined by the optimization criteria, it is worth considering what sorts of criteria are likely to produce desirable results. To illustrate the

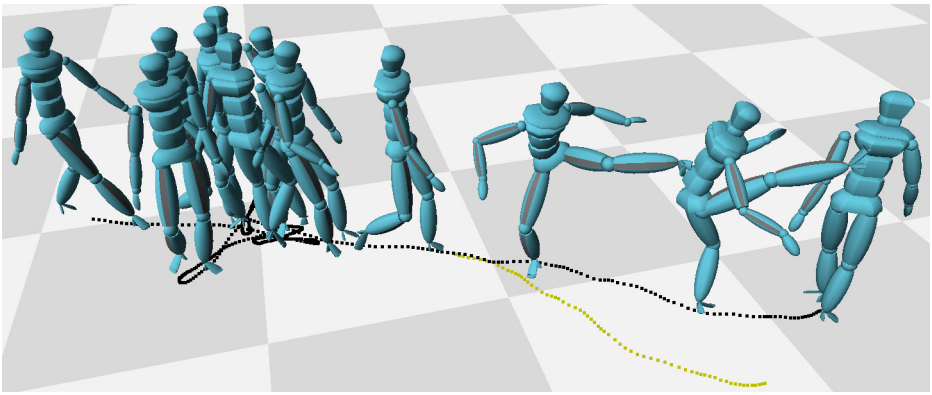

Figure 4.10: Search result where the goal was to connect walking with a series of kicks. The halting condition is that a particular kicking clip must be played, and the objective function measures the deviation of the global translation and orientation of this clip from user-specified values. The character spends approximately seven seconds turning in place in an effort to better align itself with the final clip. The dotted black line shows the path of the root in the floor plane, and the dotted yellow line indicates the path of the kicking clip under the desired coordinate transformation.

issues involved, we present a simple example. Imagine that we want to place two clips in a particular position and orientation and then find a motion that connects them. This formally corresponds to a search where the starting edge is $\mathbf{e_0}$, the halting condition is that the final edge is $\mathbf{e_T}$, and the objective function measures the difference between the desired coordinate transformation $\{\theta, x_0, z_0\}$ for $\mathbf{e_T}$ and the actual transformation $\{\theta', x_0', z_0'\}$. We used the objective function $c_1 (\theta' - \theta)^2 + c_2 \left((x_0' - x_0)^2 + (z_0' - z_0)^2\right)$, where c_1 and c_2 are weights balancing orientation errors with position errors. While in principle the search algorithm can return a reasonable result, it might also return a bizarre motion like the one shown in Figure 4.10, where the character turns around in place several times in order to better align the final clip with its target configuration.

Two lessons can be drawn from this example. First, the objective function gave no indication as to what should be done in the middle of the motion; all that mattered was that the final clip be in the right position and orientation. This is unwise since arbitrary motion is almost never desirable — the objective function should provide guidance throughout the entire duration of the motion. Second, the halting condition was probably more specific than necessary. If it did not matter what kick was performed, then the search should have been allowed to choose a kick that did not

require such effort to aim. In general, the optimization criteria should be no more restrictive than necessary, so the search algorithm is not needlessly prevented from considering viable motions. Note the tradeoff here: guiding the search toward a particular result must be balanced against unduly preventing it from considering all available options.

4.3 Path Synthesis

While formulating good optimization criteria can require some care, it is nonetheless possible to devise criteria that allow practical problems to be solved. This section shows how our search framework can be applied to path synthesis, where the goal is to generate motion such that a character travels along a user-specified path. The section provides optimization criteria for path synthesis, presents results, and discusses applications of our technique.

4.3.1 Optimization Criteria

Let $\mathbf{P}(s)$ be the path that is to be followed and $\mathbf{P_w}(s)$ be an estimate of the path travelled by the character during graph walk $\mathbf{w}$, where both paths are parameterized according to arc length s. In our implementation $\mathbf{P_w}$ is formed by projecting the root joint onto the ground at each frame and linearly interpolating these points. Let $\mathbf{e}$ be a new edge that is to be added to an existing walk $\mathbf{w}$, forming a new walk $(\mathbf{w}+\mathbf{e})$, and let $\mathbf{e_i}$ be the i^{th} frame of $\mathbf{e}$ and $s(\mathbf{e_i})$ be the arc-length distance of $\mathbf{e_i}$ from the start of $\mathbf{P_{(w+e)}}$. The position on $\mathbf{P}$ corresponding to $\mathbf{P_{(w+e)}}\left(s(\mathbf{e_i})\right)$ is defined as the point at the same arc length, $\mathbf{P}\left(s(\mathbf{e_i})\right)$. The cost of appending $\mathbf{e}$ to $\mathbf{w}$ is then the sum of squared distances between these corresponding points:

$$g(\mathbf{w},\mathbf{e}) = \sum_{i=1}^{n} \|\mathbf{P_{(w+e)}}\left(s(\mathbf{e_i})\right) - \mathbf{P}\left(s(\mathbf{e_i})\right)\|^2, \tag{4.15}$$

where $\mathbf{e}$ has n frames. Note that $s(\mathbf{e_i})$ depends on the total arc length of $\mathbf{w}$, which is why g is a function of $\mathbf{w}$ as well as $\mathbf{e}$. Any frames on the graph walk at an arc length longer than the total length of $\mathbf{P}$ are mapped to the last point on $\mathbf{P}$, and the halting condition for the search is that the total length of $\mathbf{P_{(w+e)}}$ must meet or exceed that of $\mathbf{P}$.

To improve the efficiency of the search, we precomputed the root path for each edge corresponding to a transition motion, instead of linearly blending captured root positions on the fly. We also precomputed for each edge the arc length distance of each frame from the first frame of the edge, along with the distance of the first frame from the last frame of each predecessor edge. This allowed us to compute total arc length at run time simply by summing these stored frame-to-frame path distances.

The cost function in Equation 4.15 was chosen for several of reasons. First, it can be computed efficiently, which is important if the search algorithm is to produce results in a reasonable amount of time. Second, the character is given incentive to make definite progress along $\mathbf{P}$. In particular, if the character were merely required to be *near* $\mathbf{P}$ (e.g., by measuring for each point on $\mathbf{P}_{(\mathbf{w}+\mathbf{e})}$ the shortest distance to $\mathbf{P}$), then a low cost could be achieved by oscillating about a fixed point on $\mathbf{P}$. Finally, this metric allows the character to travel at whatever speed is appropriate for the current action. For example, a sharp turn will not typically cover distance at the same rate as walking straight forward, and since both types of actions can be important, one should not be given preference over the other.

One potential problem with this cost function is that if a character is ever at exactly the correct point of $\mathbf{P}$, then it can indefinitely accrue zero cost by remaining in place. While we have not found this to be a problem in practice, it can be countered by requiring at least a small amount of forward progress γ on each frame. Specifically, the function $s(\mathbf{e_i})$ in Equation 4.15 can be replaced with $t(\mathbf{e_i}) = max(t(\mathbf{e_{i-1}}) + s(\mathbf{e_i}) - s(\mathbf{e_{i-1}}), t(\mathbf{e_{i-1}}) + \gamma)$. A second limitation of this cost metric is that irrelevant high-frequency details in the character's path can cause it to have a greater arc length than one might intuitively expect. For example, when a person walks in a straight line their pelvis trajectory is not perfectly straight, but instead oscillates in rhythm with the walk cycle. As a result, the arc length distance measured by our algorithm will be greater than the Euclidean distance between the first point and the last point. In general, this discrepancy causes the character to appear to take small "short cuts" when following a path, since the target positions on $\mathbf{P}$ gets pushed ahead by the extra amount of arc length. Filtering $\mathbf{P_w}$ to remove high frequencies is not a viable way of correcting this problem, since many high frequency characteristics are important to

the shape of the path (e.g., sharp turns). Regardless, in practice we have found the errors caused by this phenomenon to be small, especially when compared to other errors introduced in the process of fitting a discrete sequence of motion clips to a continuous path.

In many circumstances the user will want all synthesized motion to be of a single type, such as walking. To achieve this, the search can be confined to the subgraph containing the appropriate descriptive label. One can also require different types of motion on different parts of the path — for example, one might want the character to walk along the first half of the path and sneak down the rest. Consider the case where $\mathbf{P}$ is to be divided into two pieces $\mathbf{P_1}$ and $\mathbf{P_2}$ of arc length $s(\mathbf{P_1})$ and $s(\mathbf{P_2})$, with motion of type T_1 on $\mathbf{P_1}$ and of type T_2 on $\mathbf{P_2}$; the generalization to higher numbers of style transitions is straightforward. As long as the arc length of $\mathbf{P}_{(\mathbf{w+e})}$ is less than $s(\mathbf{P_1}) - \delta_s$ for some threshold δ_s, only edges with label T_1 are considered. Otherwise edges whose labels include either T_1 or T_2 are considered, and if an edge with label T_2 is chosen then all subsequent edges on that branch of the search are also required to have the label T_2 (that is, the character has switched to the second type of motion). As long as the character is in style T_1, then $\mathbf{P}$ is taken to terminate at the end of $\mathbf{P_1}$, so the path point of any frame on $\mathbf{P}_{(\mathbf{w+e})}$ whose arc length exceeds $s(\mathbf{P_1})$ will map to the end of $\mathbf{P_1}$ in Equation 4.15. In our implementation we set δ_s to half the height of the character.

4.3.2 Results

All experiments with our path synthesis algorithm were run on a machine with a 1.3GHz Athlon processor. The time needed to execute the search varied from example to example, but in the worst cases it was within ten percent of the duration of the synthesized motion. Relative to the search algorithm, the time needed to compute blends at transitions and to enforce constraints was negligible. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/MoGraphs.

While the examples in Figure 4.2 demonstrate that our algorithm can accurately fit motion to new paths, the success is perhaps unsurprising since the input motions already contained a fair amount of variation, including straight-ahead marches, sharp turns, and smooth changes of curvature. However, even with small data sets it is possible to fit motion to nontrivial paths. We built a

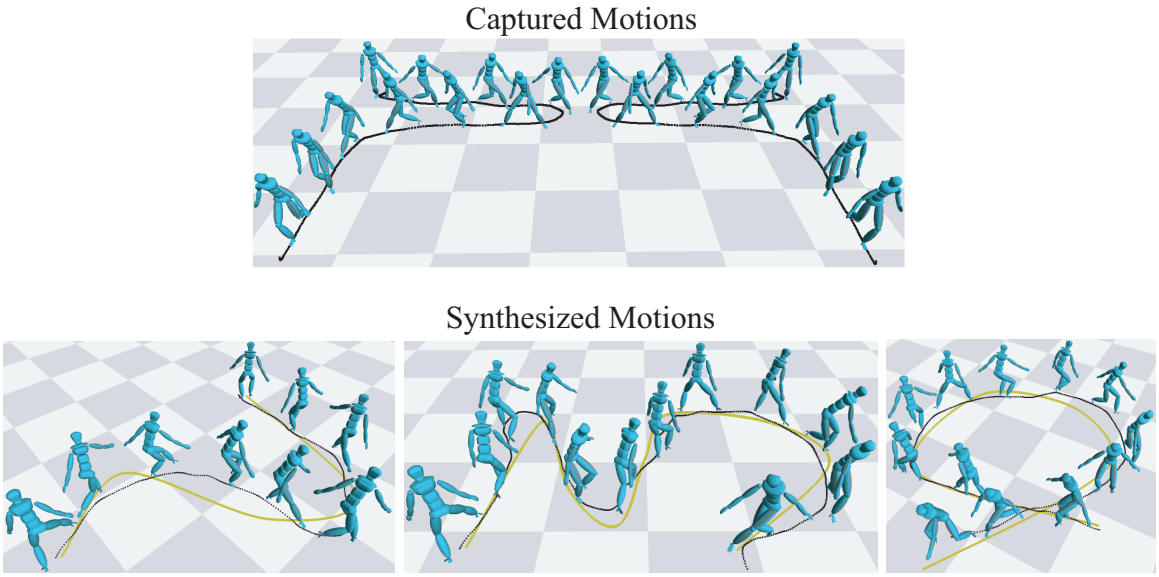

Figure 4.11: Original sneaking data and synthesized motions designed to fit paths. For the original data, the motion path is shown in black. For the synthesized motions, target paths are yellow and synthesized paths are black.

motion graph out of two 12.8s clips of an actor sneaking, one clip consisting of original data and the other of the mirror image of this data. We then constructed the new motions shown in Figure 4.11, each of which were also approximately 13s in duration. Note that while the synthesized motion is close to the target path, the fit is not exact. This is because the synthesized path is a concatenation of path segments from a discrete set (namely, the motion path of each edge in the motion graph), and hence arbitrary target paths cannot be perfectly reproduced.

With larger data sets it is possible to fit motion to more complicated paths, such as the two shown in Figure 4.12. For each path two motions were synthesized, one using a data set of about 100s of walking motion and the other using about 100s of martial arts movements. Note that our algorithm is successful for motions that are not obviously locomotion: in the martial arts examples, the character kicks, punches, ducks, and dodges in a contrived manner such that it follows the specified path.

Figure 4.13 shows paths containing constraints on the allowable motion type. Specifically, the character is required to walk on the yellow portions of the path, sneak on the purple portions, and

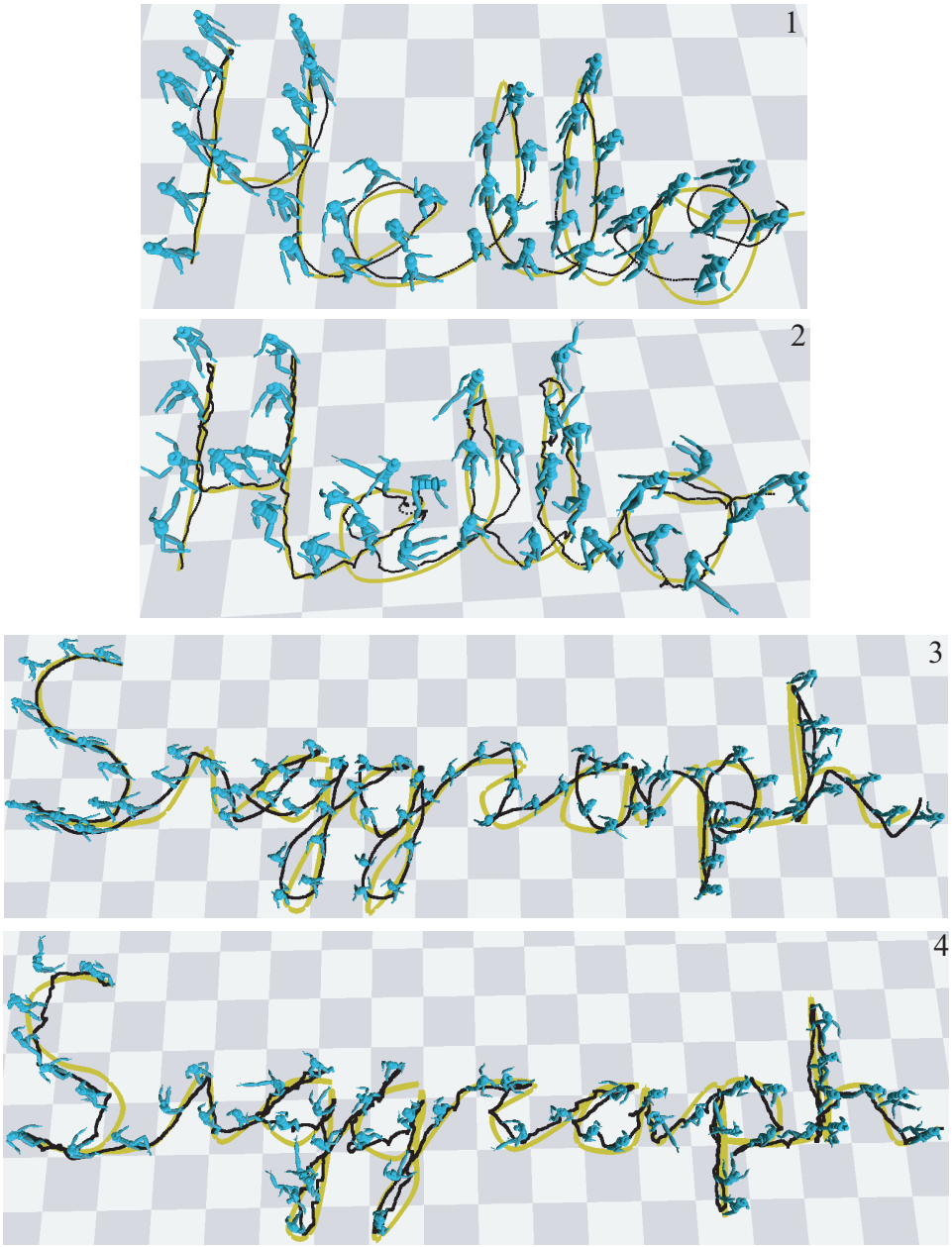

Figure 4.12: Fits to complicated paths. In each case the target path is yellow and the synthesized path is black. The motions in images (1) and (3) were synthesized from walking data and the motions in images (2) and (4) were synthesized from martial arts data.

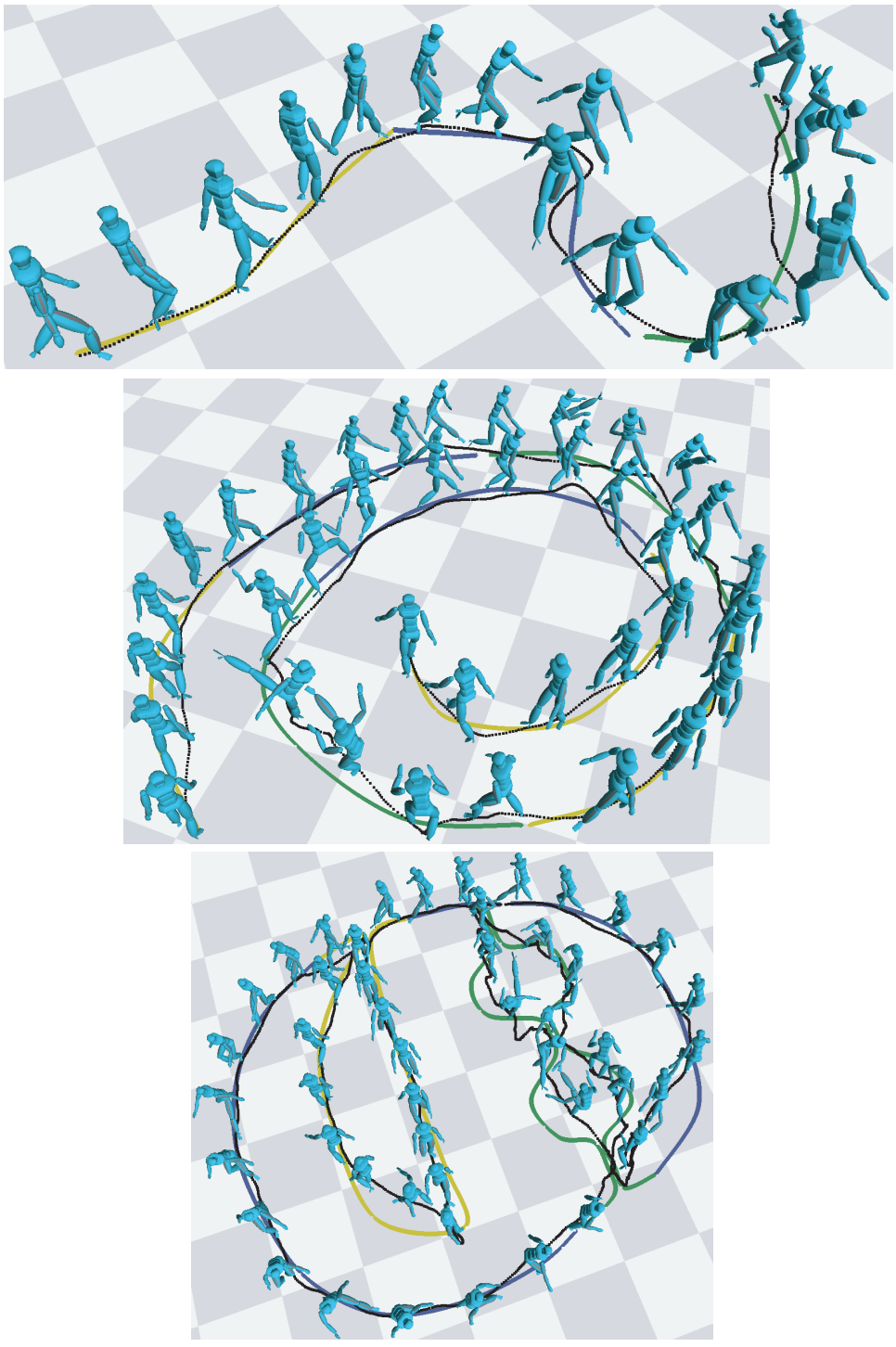

Figure 4.13: Fits to paths where the character is required to walk on the yellow portions, sneak on the purple portions, and perform martial arts movements on the green portions.

perform martial arts movements on the green portions. Note that the character both stays close to the path and transitions between motion types near the appropriate points. The motion graph used in these experiments was built from about three and a half minutes of motion data.

While the ability to fit characters to complicated paths is useful in and of itself, we also note that with previous methods a tremendous investment of effort would have been necessary to create motions as lengthy and complex as the ones shown in Figures 4.11–4.13. Starting with just the raw data, an artist might easily have spent weeks producing the more complicated motions. In contrast, with our system the entire amount of time needed to produce even the longest of these motions (about 2.5 minutes in duration) was less than fifteen minutes, including the time needed to build the motion graph (identify local minima and set thresholds) and execute the search algorithm.

The range of paths that can be fit with our algorithm is highly dependent on the input data set — paths with sharp turns, for example, can not be fit accurately if no sharp turns are present in the data. Similarly, if the character only has a few ways of accomplishing certain maneuvers, than these motion segments will necessarily appear repeatedly. For instance, in our walking data the character only had one way of making a roughly 180 degree turn, and hence this movement was repeated at the tops of the "*H*" and the two "*l*"'s in the "*Hello*" fit of Figure 4.12. We also noticed that sometimes walking motions would contain speed variations when fit to paths consisting of a single straight line. While such speed changes are natural when a turn is being made, people tend to walk at the same rate when travelling directly forward. The synthesized motion hence sometimes appeared unnatural, not because of flaws in the joint trajectories themselves, but rather because the sequence of actions was atypical. This is a general limitation of our search framework: properties relating to the *meaning* of a motion can be difficult to encode in terms of optimization criteria.

4.3.3 Applications Of Path Synthesis

Directable locomotion is needed in many applications, and here we summarize some circumstances in which our path synthesis algorithm might be useful.

High-Level Keyframing. If a character must perform certain types of actions in a specific sequence and in particular locations, then a user can simply draw a series of paths labelled with

appropriate actions types. This allows complex animations to be controllably generated without the tedium of manual keyframing. Similarly, if an AI algorithm is used to determine what actions must be carried out based on the environment and the internal state of a character, then our technique may be used to synthesize an appropriate animation.

Interactive Control. Our path synthesis technique can be modified to give a user interactive control over a character — for example, the user might press arrow keys to control the direction in which a character travels. In a preprocessing stage, for each edge in the graph our path fitting algorithm is used to find motions that travel in straight lines pointing in a sampling of directions. The first edge of each optimal graph walk is retained and stored in a lookup table. Motion is then synthesized at run time by selecting the entry in the current edge's table that best matches the intended direction of travel.

We implemented this on two data sets, one containing 27s of kicking motions and the other containing 100s of walking motions, with the lookup tables containing 16 entries (22.5° separation between directions). Each execution of the search algorithm found 90 frames of motion that best travelled in the desired direction. While the amount of storage needed was modest (3.5% and 7.2% of the data set size for, respectively, the kick and walk examples), the computation demands were considerable (about 1 hour for the kicking data set and 8 hours for the walking data set). However, our method for generating lookup tables is essentially brute force, and we believe that more efficient methods could be devised.

Crowds. While our discussion has so far focused on a single character, our methods could easily be applied to several characters in parallel. For example, a standard collision-avoidance algorithm could be used to generate a path for each individual, and the motion graph could then generate motion that conforms to this path. Moreover, the techniques described at the end of Section 4.2.1 can be used to add variability to the generated motion.

4.4 Discussion

This chapter has presented a framework for controllably generating lengthy, realistic motions by piecing together short segments of captured motion. The basic strategy is to add special transition motions to the original data set to form a graph, and then to search this graph for motion that optimizes a user-supplied objective function. Our methods are highly automated: the only inputs needed for graph construction are thresholds which determine where transitions may occur, and for synthesis the user only needs to supply appropriate optimization criteria. We have demonstrated our methods on the important problem of synthesizing motion that travels down a given path.

While the time and space requirements for building and using motion graphs were quite modest for our data sets, our example motions were limited to a few different kinds of actions (primarily walking, sneaking, and kicking). While we believe our experiments are sufficient to show the potential of our method, a character with a truly diverse set of actions will require a much larger data set, and the scalability of our framework therefore bears discussion. The principal computational bottleneck in graph construction is locating candidate transition points, since this requires comparison of $O(n^2)$ pairs of frames for an n-frame data set. This calculation is trivial to parallelize, and it may be sped up by finding local minima in a coarse-to-fine fashion (e.g., subsample the original data, find local minima, and then refine the regions around these minima at a higher resolution). Also, the distances between old frames need not be recomputed if additions are made to the database. Nonetheless, the total computation fundamentally requires quadratic time, and if the data set size makes this computation prohibitive, the user may want to manually segment the data set to narrow the search for transitions (e.g., perhaps no attempt would be made to find transitions directly connecting football motions with ballet).

The number of edges leaving a node in general grows with the size of the motion graph, and hence the branching factor in our search algorithm may grow as well. However, we believe that in practice a large motion graph will necessarily contain many distinct kinds actions. In particular, a point is reached at which there is little advantage to having more motion of the same type — for example, obtaining more walking motion is not useful once one can already direct a character

along nearly any path. Hence the branching factor in a particular subgraph will remain stationary once that subgraph is sufficiently large. We anticipate that typical graph searches will be restricted to a small number of subgraphs, and so we expect that our search algorithm will remain practical even for larger graphs.

Although motion graphs allow one to quickly generate lengthy and complicated motions, they suffer from several limitations that may frustruate their use in online applications. First, at present there is no general way of characterizing the range of motions that can be produced. For example, there is no simple way of placing bounds on how much a synthesized motion will deviate from the path that it is to follow, even in the limit where this motion is the globally optimal graph walk. Second, a motion can be controlled only at nodes, and if the average edge length is too long then a character may be insufficiently responsive. While the average edge length can be reduced by allowing transitions to occur more quickly, our current methods provide no general way of minimizing the time between transitions. Finally, although in our experiments motion could be synthesized in approximately real time, in an online setting synthesis must occur much faster than real time in order to allow time for other tasks, such as rendering and simulation. Extensions to motion graphs that are targeted toward online control have been made by Gleicher et al. [33] and by Lee and Lee [53].

4.4.1 Comparison with Concurrent Work

Concurrent with this work, Arikan and Forsyth [5] and Lee et al. [52] proposed graph-based synthesis models similar to our own. As with our work, they automatically synthesized transitions at places where motions were locally similar, and they used search algorithms to extract motions from the resulting graph. Despite these high-level similarities, their models differed from ours in several details:

1. **Distance Metric.** Arikan and Forsyth [5] defined frame distance as a weighted sum of the Euclidean distance between joint positions and velocities, measured relative to the root's coordinate frame. Lee et al. [52] defined frame distance as a weighted sum of the great-arc distance between joint orientations and the Euclidean distance between joint velocities (see

Section 4.1.1.1 for comments on directly comparing joint orientations). Joint data was either represented in the global coordinate frame (to preserve interactions with stationary objects) or relative to the root's local coordinate frame. Our metric incorporates information from higher-order derivatives than velocity, and our optimization for aligning local coordinate frames may be viewed as a generalization of directly aligning the roots. Also, we note that our distance metric could trivially be adapted to preserve global positions and orientations; one would simply skip the optimization of Equation 4.1 and set $\theta = x_0 = z_0 = 0$. While we have not run any formal comparisons of these different distance metrics, we suspect that none of them are qualitatively superior to the others.

2. **Transition Locations.** As with our work, Lee et al. [52] only allowed transitions at local minima of D, and they further disallowed transitions between frames with different contact states. With constraint enforcement, we have found the latter to be unnecessary. Arikan and Forsyth [5] allowed transitions at any point of D below a threshold. Nearby points were clustered to create entire regions where transitions could occur, and these clusters were subdivided to create a hierarchy of graphs where transitions could occur at progressively finer granularities. This approach provides some added flexibility in selecting precise transition locations at the expense of requiring additional storage.

3. **Transition Synthesis.** Both Arikan and Forsyth [5] and Lee et al. [52] generated transitions by making jump cuts and adding displacement maps [98] to preserve, respectively, C_0 and C_1 continuity. Roughly speaking, this makes transitions more localized than with our approach: building a displacement map for a transition from $\mathbf{M_i}$ to $\mathbf{M'_j}$ only uses information from these two frames (and possibly the frames immediately before and after, to compute velocity differences), whereas our method uses information from every frame in the transition window. An advantage of using displacement maps is that transitions can be made very close to the boundaries of a motion, whereas our method requires $\mathbf{M_i}$ and $\mathbf{M'_j}$ to be at least L frames from, respectively, the start of $\mathbf{M}$ and the end of $\mathbf{M'}$. On the other hand, it is difficult to reliably preserve C_1 continuity with displacement maps because velocities must

be estimated with finite differences, and errors in this estimate can create subtle but disturbing jumps in velocity. Our method guarantees C_1 continuity without having to explicitly compute velocities.

4. **Search Methods.** Arikan and Forsyth [5] employed a random search method to find motions with constraints at sparse frames (e.g., the beginning and end must be in specified positions and orientations). Our branch-and-bound method is poorly suited to these sparse constraints, because the cost of a motion is not influenced by frames in between constraints. However, following the discussion of Section 4.2.2, we believe that in practice sparse constraints are undesirable, because large portions of a motion could literally do arbitrary things. Lee et al. [52] clustered similar frames and then used the motion graph to determine connections between clusters. Each frame in the graph with multiple outgoing transitions was then associated with a tree representing all possible sequences of clusters up to some maximum depth; these trees were used to speed up searching at the cost of additional storage. Both of these alternative approaches to searching could be applied to the motion graphs built with our system.

Chapter 5

Registration Curves

Chapter 4 showed how new motions can be synthesized by building graph structures that facilitate the rearrangement and attachment of short segments of captured motion. While this is a natural strategy for controlling the sequencing of different actions, it breaks down when one wants individual actions that are not present in the original data set. Imagine, for example, that one has collected a data set containing several motions of a character reaching to different positions on a shelf, with the goal of being able to animate new reaching motions. Clearly, graph-based methods are of no avail, since there is no sensible way of rearranging captured reaches to form new ones. Similarly, while one can edit a captured reach to reposition the hand (e.g., using inverse kinematics methods such as the one described in Chapter 3), subtle but important correlations between the movement of different parts of the body may be lost if the adjustment is too large — even if the character's hand ends up in the right place, the motion may look awkward or unnatural. More generally, existing motion editing methods [15, 98, 31, 54] provide no simple way of synthesizing the detail present in captured motions; indeed, they are specifically designed to adjust motions *without* changing the finer details.

An alternative to rearranging or editing captured motions is to blend them [96, 76]. Motion blending combines captured examples according to weight functions that specify how much influence each example has at each point in time. Different weight functions yield different synthesis operations; see Figure 5.1. A transition between two motions can be created by initially placing all of the weight on one motion and then smoothly shifting it to the other motion; this technique

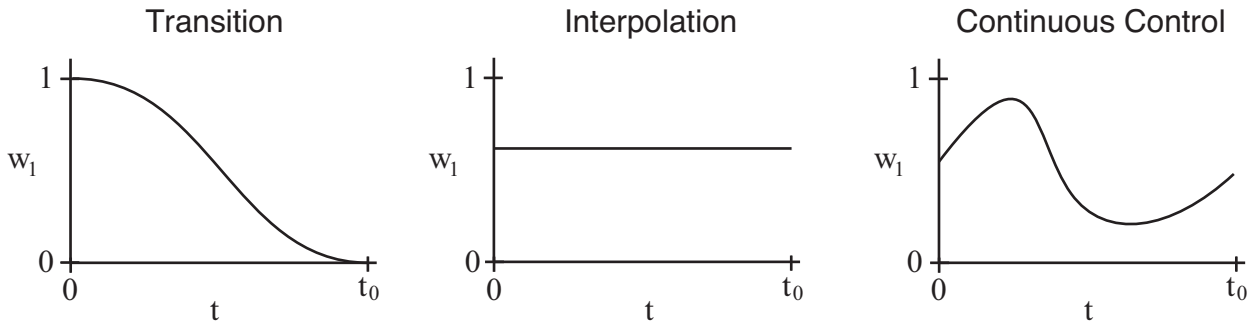

Figure 5.1: Different weight functions yield different blending operations. These graphs depict weight functions $\mathbf{w}(t) = (w_1(t), w_2(t))$ that might be used for transitioning, interpolation, and continuous control. In each case $w_2(t) = 1 - w_1(t)$.

was used in Chapter 4. An interpolation of the input motions can be created by holding the blend weights constant. This is useful for creating intermediary actions, such as a reach that has a target location in between those of the captured examples. Finally, continuous control can be gained by continually varying the blend weights. For example, if the inputs are various kinds of locomotion, then one could adjust the blend weights to interactively control the speed and direction at which a character travels. Blending operations such as these have considerable practical importance and have proven useful in commercial applications like video games [62, 88].

While blending is a powerful technique, it clearly will not produce realistic results on arbitrary sets of input motions. Instead, these inputs must be chosen with some care, and the range of motions that can be successfully blended depends on how much information is given to the blending algorithm. In this dissertation we are interested in *automatic* motion blending, where nothing is known about the input motions other than the root position, joint orientations, and constraint annotations at each frame. This chapter shows how many of the limitations of existing automatic blending algorithms can be avoided by building *registration curves*, which are data structures that encapsulate relationships involving the timing, local coordinate frame, and constraint states of an arbitrary number of input motions. While some existing blending algorithms use similar timing and constraint (but not coordinate frame) relationships [76, 68], they require a user to supply this information directly through special annotations added to the input motions. Registration curves

can be built and used without user intervention, both automating these manual algorithms and expanding the range of motion that can be blended successfully.

The remainder of this chapter describes how to construct and use registration curves. First, Section 5.1 provides an overview of registration curves. Sections 5.2 and 5.3 then explain how to build registration curves and use them to create blends. Finally, Section 5.4 demonstrate registration curves in common blending applications, and Section 5.5 concludes with a discussion of the advantages and limitations of our methods.

5.1 Overview

A blend $\mathbf{B}(t)$ is a motion constructed from n input motions $\mathbf{M_1}, \ldots, \mathbf{M_n}$ and an n-dimensional weight function $\mathbf{w}(t)$. While the specific sequence of skeletal poses in $\mathbf{B}(t)$ depends on the blending algorithm, if $w_i = 1$ at t_0 and all other weights are 0, then $\mathbf{B}(t_0)$ is identical to a frame from $\mathbf{M_i}$. Blend weights usually sum to 1 and are often assumed to be non-negative, but neither of these properties are required [76].

When the only available information is the input motions themselves, a standard blending method is *linear blending*. The i^{th} frame of a linear blend is a weighted average of the skeletal parameters at the i^{th} frame of each input motion. The blended root position is calculated by averaging either the input root positions or the input root velocities, and blended joint angles can be calculated in a variety of ways, including averaging Euler angles [76] and computing the exponential map of averaged logarithmic maps [68]. Constraints are not handled by linear blending.

Linear blending produces reasonable results when it is used to create short blends of similar motions. Indeed, in Chapter 4 brief transition motions were created for motion graphs by applying a modification of linear blending that inferred constraints from the input motions. However, linear blending suffers from several problems that make it of limited use in more general circumstances. We propose an alternative blending algorithm that, as with linear blending, combines frames via averaging, but also automatically extracts information from the input motions to help decide which frames to combine, how to position and orient them prior to averaging, and what constraints should

exist on the result. This information is used to create a registration curve, which is a composite data structure consisting of a *timewarp curve*, an *alignment curve*, and a set of *constraint matches*. The remainder of this section provides an overview of what these elements are and why they are needed.

5.1.1 Timing

Logically similar motions can have different timing in the sense that corresponding events may occur at different absolute times. For example, successive heel strikes of a walk are spaced further apart in time than those of a run, and a jab will reach its apex sooner than a stronger, more committed punch. If linear blending is used to combine motions such as these, then frames from unrelated portions of the motions will be combined, which can create pronounced and disturbing artifacts (Figure 5.2). A solution is to timewarp the input motions so corresponding events occur simultaneously. This involves determining a timewarp curve $\mathbf{S}(u)$ where each point provides the indices of corresponding frames, one from each input motion (Figure 5.3). These frame indices may specify times in between data samples; when this occurs, the captured skeletal poses are interpolated by linearly interpolating root position data and applying spherical linear interpolation to joint orientation data. During blending, only corresponding frames are every combined — that is, the i^{th} blend frame is a composition of the input frames from some point $\mathbf{S}(u_i)$ on the timewarp curve.

The timewarp curve is assumed to be continuous and strictly increasing, so if $u_1 > u_0$, then $S_i(u_1) > S_i(u_0)$. As a result, given a frame at any time t in motion $\mathbf{M_i}$, a unique position $u_{S_i}(t)$ on the timewarp curve can be determined. The parameter u may be thought of as defining a reference time system: each input motion can be viewed as a realization of some canonical motion, and when this canonical motion is at frame u then $\mathbf{M_i}$ is at frame $S_i(u)$. Note that this timewarping may make sense *locally* even if the input motions themselves are quite different overall. For example, say that in $\mathbf{M_1}$ a character walks straight forward and opens a door, and in $\mathbf{M_2}$ it stumbles a few steps and falls into an open manhole. The walking portion of $\mathbf{M_1}$ and the stumbling portion of $\mathbf{M_2}$ may have

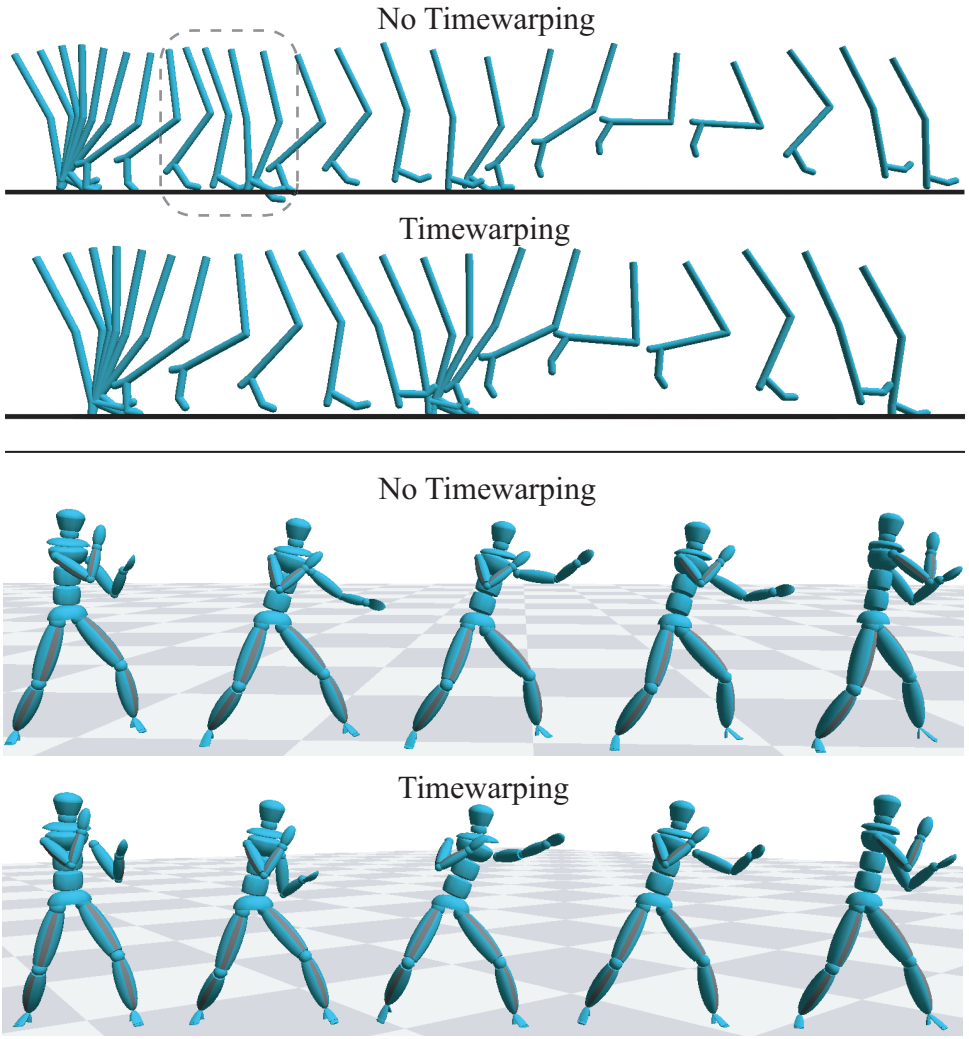

Figure 5.2: Top: A transition between walking and jogging that spans two locomotion cycles. For clarity, only the right leg is shown. Without timewarping, out-of-phase frames are combined and the character's leg stutters as if an extra step is being taken (see circled region). **Bottom:** An interpolation of a jab and a stronger, more committed punch. Without timewarping, the character's arm wobbles.

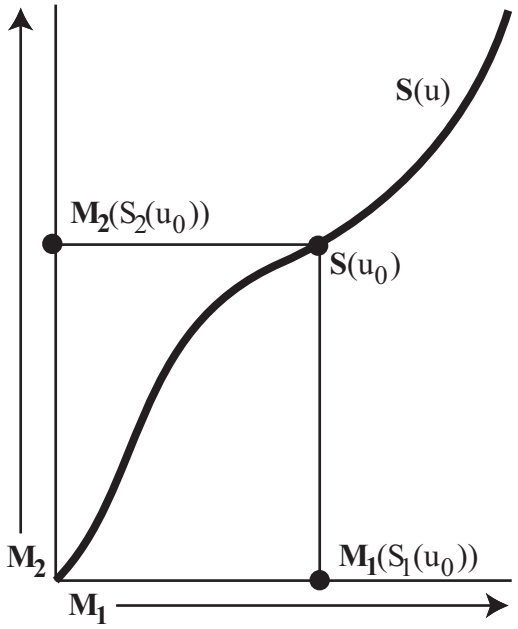

Figure 5.3: An example timewarp curve for two motions.

a perfectly reasonable timewarp curve even if the entire motions do not. Timewarping is therefore valid for inherently local blending operations like transitioning.

Since no information is present other than the motion data, $\mathbf{S}(u)$ is created using the heuristic that the more similar the skeletal poses are for a set of frames, the more likely it is that these frames correspond. This heuristic complements the way frames are combined during blending: it is more reasonable to average skeletal parameters if they are already similar, and thus motions are aligned in time so corresponding skeletal poses are as similar as possible. Section 5.2.1 provides details on how timewarp curves are constructed.

5.1.2 Coordinate Frame Alignment

Linear blending can fail even when frames that occur at the same time have very similar skeletal poses. Imagine using linear blending to create a halfway interpolation of two walking motions, one curving to the left and the other to the right. While one would intuitively expect the character to walk straight forward, the blended root path collapses because the input root positions are combined through linear interpolation (Figure 5.4). Also, the character suddenly flips around near

Figure 5.4: A halfway interpolation of two walking motions that curve in opposite directions. The lines show the projection of the root trajectory onto the ground at each frame, and the triangles show the root's position and orientation at selected frames. Linear blending causes the root trajectory to collapse. Note the sudden change in root orientation at the end of the motion, when the accumulated angle between the roots exceeds $180°$.

the end of the motion. This is because standard orientation averaging schemes like spherical linear interpolation have discontinuities when the input angles change by more than $180°$.

To solve this problem, we use the fact that motions are fundamentally unchanged by rotations about the vertical axis and translations in the floor plane (see Section 4.1.1.1 of Chapter 4). In particular, if specific frames $\mathbf{M_1}(t_1), \ldots, \mathbf{M_n}(t_n)$ are to be combined, then the motions can be aligned so they are as similar as possible at these frames, a process we refer to as *coordinate frame alignment*[1] (Figure 5.5). If the character's body were a single point located at the character's root, then coordinate frame alignment would be the same as finding transformations that place each root in the same global position and orientation. Since we instead represent the body as a skeleton, the methods of Chapter 4 are used to find transformations that optimally align these skeletal postures.

Coordinate frame alignment allows us to construct an alignment curve $\mathbf{A}(u)$, which gives a set of transformations that align the frames at each point $\mathbf{S}(u)$ on the timewarp curve. By analyzing how these transformations change during a blend, we can incrementally blend root parameters in a way that avoids the artifacts of traditional methods (Figure 5.4). Section 5.2.2 explains how to construct an alignment curve and Section 5.3.2 provides details on using it in blending.

[1] Whenever the word "frame" is used in isolation, it refers to a frame of motion, and should not be confused with a coordinate frame.

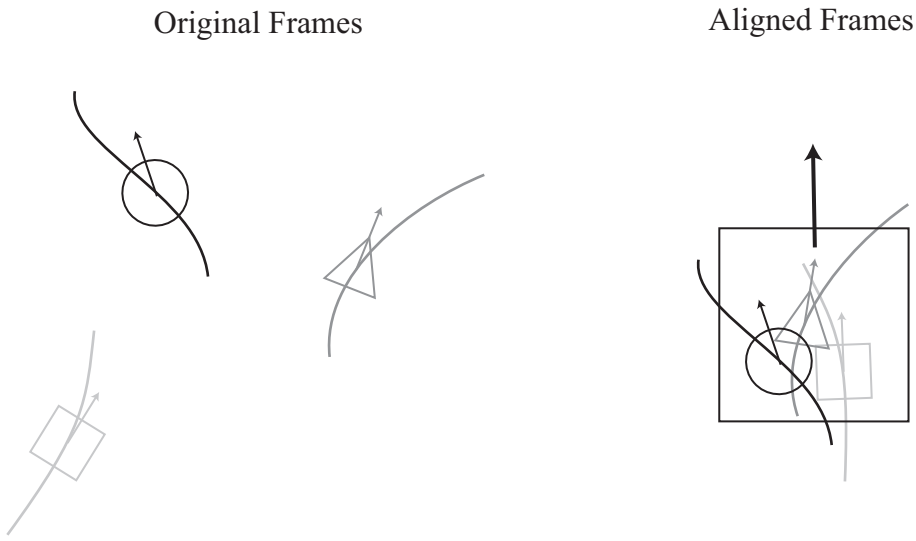

Figure 5.5: Left: Before alignment, a set of corresponding frames is scattered within the global coordinate system. The circle, triangle, and square represent root configurations of three frames, with the arrows indicating orientations in the ground plane. The curved lines depict the root trajectories of the surrounding segments of motion. **Right:** The same set of frames after alignment. Aligning transformations are based on the entire skeletal posture at each frame, and hence in general the roots do not end up in the same global position and orientation.

5.1.3 Constraint Matching

Once a position and orientation is determined for the root, the rest of the blended skeletal posture is created simply by averaging joint parameters. As a result, kinematic constraints on the blended motion may be violated. While it is straightforward to adjust the blend so constraints are satisfied (using, for example, the methods of Chapter 3), these constraints must first be explicitly labelled. Even if the constraints happen to already be satisfied, it is desirable for the blend to have explicit constraint annotations in the event that further changes are made to the motion, such as if it is used as an input for another blend. However, determining constraints can take some care because the input motions may have fundamentally different constraint states. For example, while a walking motion always has at least one foot planted, a running motion contains flight stages where no constraints exist, and the intervals over which constraints are active will not match even if the motions are timewarped. Also, the user might want to blend motions with different numbers of

constraints. For instance, the user might want to create different turning motions by interpolating a motion where a character stands still with one where it turns in place (and hence picks up its feet).

If the interval of each constraint is represented in terms of the global time parameter u, then it is reasonable to expect that constraints existing over nearby intervals have the same structural meaning, even if they do not span identical regions of time. For example, in both walking and running a foot is planted whenever it is supporting the character's weight, although the duration of this plant relative to the locomotion cycle varies. To infer the constraints on a blend, the global times at which these related constraints start and end may be averaged just like ordinary skeletal parameters. Section 5.2.3 presents an algorithm for automatically identifying sets of related constraints, which we call constraint matches. To handle input motions with different numbers of constraints, this algorithm allows individual constraints to be discarded or split into smaller pieces.

5.2 Building Registration Curves

Given n motions $\mathbf{M_1}, \ldots, \mathbf{M_n}$, a registration curve is created in three stages:

1. Build a timewarp curve $\mathbf{S}(u)$ representing sets of corresponding frames.
2. Build an alignment curve $\mathbf{A}(u)$ that specifies rigid $2D$ transformations that align the frames at each point on $\mathbf{S}(u)$.
3. Map constraints from the input motions into the global time frame defined by $\mathbf{S}(u)$ and identify constraint matches.

This section explains each of these stages in detail.

5.2.1 Timewarp Curves

A timewarp curve is built around a dense collection of *frame correspondences*, which are sets of n frame indices, one from each input motion, that occur at structurally similar points in the motions. The entire collection of frame correspondences is called a *time alignment*. A time alignment is effectively a discrete approximation of a timewarp curve, and a proper timewarp curve can be

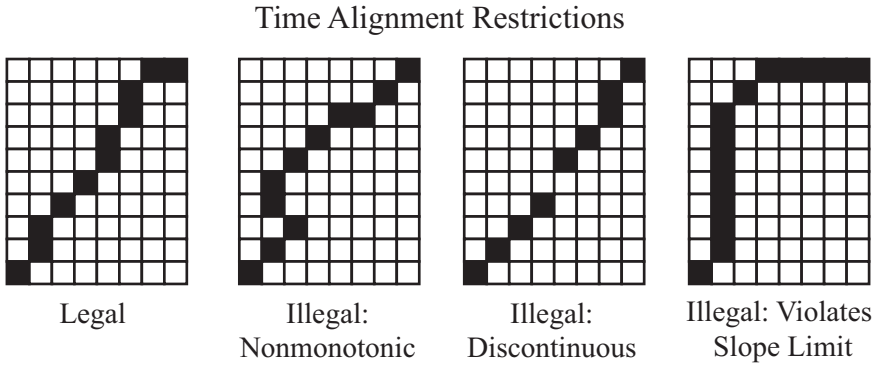

Figure 5.6: Legal and illegal time alignments ($W = 2$).

generated by fitting a strictly increasing spline to it. We first discuss the construction of timewarp curves when there are just two input motions, and then generalize the procedure to an arbitrary number of input motions.

5.2.1.1 Timewarp Curves for Two Motions

Let $\mathbf{M_1}$ have r_1 frames and $\mathbf{M_2}$ have r_2 frames, and imagine an r_1 by r_2 grid where cell (i, j) represents the frame pair $(\mathbf{M_1}(t_i), \mathbf{M_2}(t_j))$. A time alignment corresponds to a sequence of cells on this grid. To be a reasonable approximation of a timewarp curve, the time alignment is required to obey the following properties, which are illustrated in Figure 5.6.

1. **Continuity.** Each cell on the path must share a corner or edge with another cell on the path.

2. **Monotonicity.** Paths must always travel up and/or to the right.

3. **Slope Limit.** At most W consecutive steps may be taken in the same direction (horizontally or vertically).

The slope limit restriction prevents "degenerate" time alignments (and hence degenerate timewarp curves) where a large section of one motion maps to a small section of another. While such a mapping can be logically reasonable — e.g., if one motion contains a pause that is not present in the other — in practice it often introduces undesirable distortions into blends. Intuitively, this is because timewarping may be viewed as a process that trades off error in body posture (combining

frames with different skeletal poses) with error in timing (allowing time to flow at different rates for different motions), and neither of these errors can be too large if blending is too succeed. In our implementation we set W to 2 or 3.

There are many time alignments that obey the above restrictions, and we therefore construct one that is, under some measure, optimal. We use the heuristic that corresponding frames should appear more similar than frames that do not correspond, where similarity is measured with the distance metric defined in Equation 4.1 of Chapter 4. Section 5.5 discusses some of the limitations of this approach; for now we simply note that it is reasonable given that no information is present beyond the captured body postures. In our implementation the point cloud windows used in the distance computation were three frames wide ($L = 1$, or about a tenth of a second), although we have found that the optimal time alignment changes little with shorter or longer windows (between 0s and $\frac{1}{3}$s).

A continuous, monotonic, and slope-limited time alignment that minimizes the sum of the distances between corresponding frames can be found with dynamic programming, which is also called dynamic timewarping in the speech recognition literature [73]. Assume that a starting cell c (i.e., an initial frame correspondence) is known. For every cell c' that can be connected to c with at least one valid path, dynamic timewarping efficiently finds the optimal path by exploiting the fact that if the optimal path from c to c' passes through c'', then the subpaths from c to c'' and c'' to c' must also be optimal. Let $D_{i,j}$ be the frame distance associated with cell (i, j), and let $v_{i,j}$ be the total cost of the optimal path from c to cell (i, j). As long as $D_{i,j}$ is strictly positive, the cost of taking a horizontal step on the grid followed by a vertical step (or vice-versa) is always more costly than taking a single diagonal step. In light of this and the continuity, monotonicity, and slope limit restrictions, the end of a partial time alignment can only be extended in the $2W - 1$ ways shown in Figure 5.7, and therefore when calculating the optimal path to cell (i, j) it is sufficient to consider only the $2W - 1$ cells that connect to it through one of these extensions. Hence $v_{i,j}$ can be computed recursively:

$$v_{i,j} = \min \left\{ \min_{r \in [1,W]} \left\{ v_{i-r,j-1} + \sum_{k=0}^{r-1} D_{i-k,j} \right\}, \min_{r \in [1,W]} \left\{ v_{i-1,j-r} + \sum_{k=0}^{r-1} D_{i,j-k} \right\} \right\}, \qquad (5.1)$$

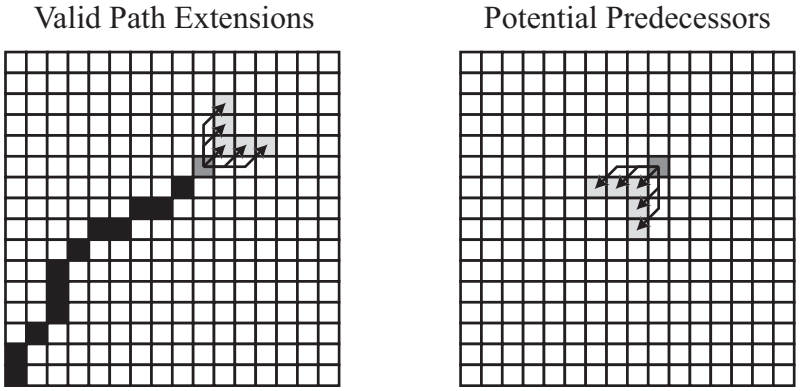

Figure 5.7: Let the slope limit be $W = 3$. **Left:** A valid path can only be extended in $2W - 1$ ways. **Right:** When computing the optimal path that terminates at a given cell, one only needs to consider the $2W - 1$ cells from which these extensions could originate.

where the first expression considers horizontal path sections that end at cell (i, j) and the second expressions considers vertical path sections. The total cost of the optimal path to each cell may now be determined by evaluating Equation 5.1 for all cells that are either 1 row above or 1 column to the right of c, then all cells either 2 rows above or 2 columns to the right of c, and so on. Processing cells in this order ensures that each v on the right hand side has a known value. Each cell in the grid is either marked as unreachable or is annotated with both its optimal cost $v_{i,j}$ and the cell that minimized Equation 5.1, which is called the *predecessor*. If a reachable cell $\tilde{c}$ is selected to be the end of the path, then the rest of the (optimal) path can be reconstructed by following the chain of predecessors back to c.

All that remains to be determined are the initial cell c and the final cell $\tilde{c}$. Typically c will be the lower left corner of the grid and $\tilde{c}$ will be the upper right corner, which is necessary to ensure that the entirety of the the input motions are spanned by the time alignment. However, for a transition it may be desirable to set c at the transition center, which will be in the interior of the grid. In this case, a time alignment can be constructed by separately computing "forward" and "backward" time alignments, where the former is computed as described above and the latter simply replaces up/right with down/left. Also, $\tilde{c}$ may have no preferred location, in which case our algorithm selects the cell on the grid boundary whose optimal path has the minimum average cell

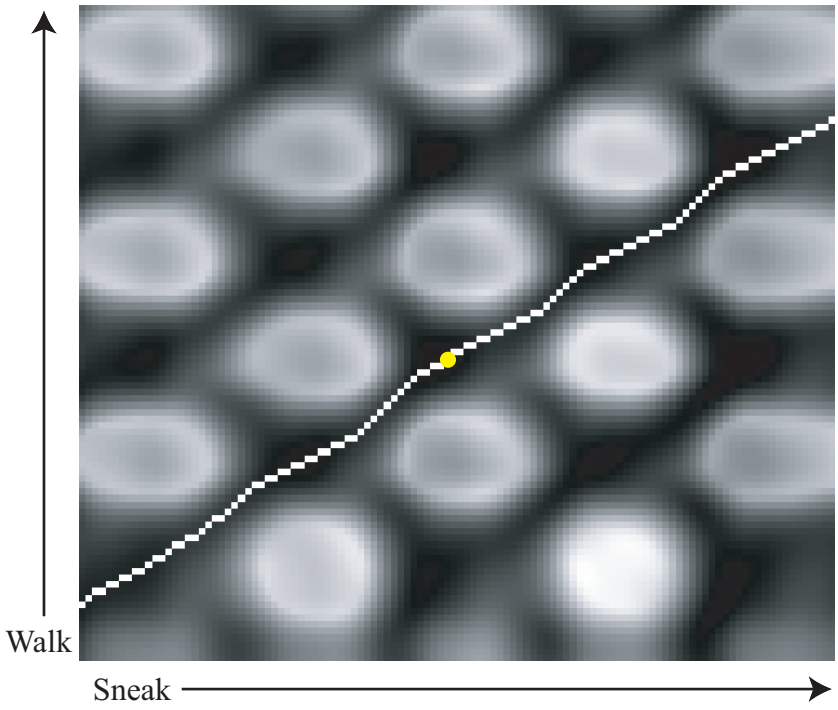

Figure 5.8: A time alignment for a walk and a sneak. The background image shows frame distances; smaller distances correspond to darker cells. The central yellow circle was chosen as the starting point, and optimal paths (shown in white) were separately generated forwards and backwards. The local slope variations arise because the character pauses slightly at each footstep in the sneaking motion but travels at a steady pace in the walking motion.

cost. To expedite this calculation, it is convenient to store not just v and the predecessor at each cell, but also the number of cells on the optimal path. Figure 5.8 shows an example time alignment where the starting cell was near the center of the grid and no preferred value was given for $\tilde{c}$.

Since the grid is r_1 by r_2, the total cost of using dynamic programming is $O(r_1 r_2)$, even if cells that are outside of the slope limits are not processed (Figure 5.9). A more efficient variant of the algorithm is to find the optimal path on a smaller k by k grid whose lower left corner is at c, retain the first k' cells, and iterate the process with cell $k'+1$ serving as the lower left corner of the new grid. This algorithm is $O\left(\frac{k^2}{k'}\left(r_1+r_2\right)\right)$, since $O\left(\frac{r_1+r_2}{k'}\right)$ grids must be processed with $O(k^2)$ work per grid. In practice we have found that relatively small values of k (15 to 20 frames; $k'=10$) yield results qualitatively similar to computing over the full grid. Indeed, in some cases

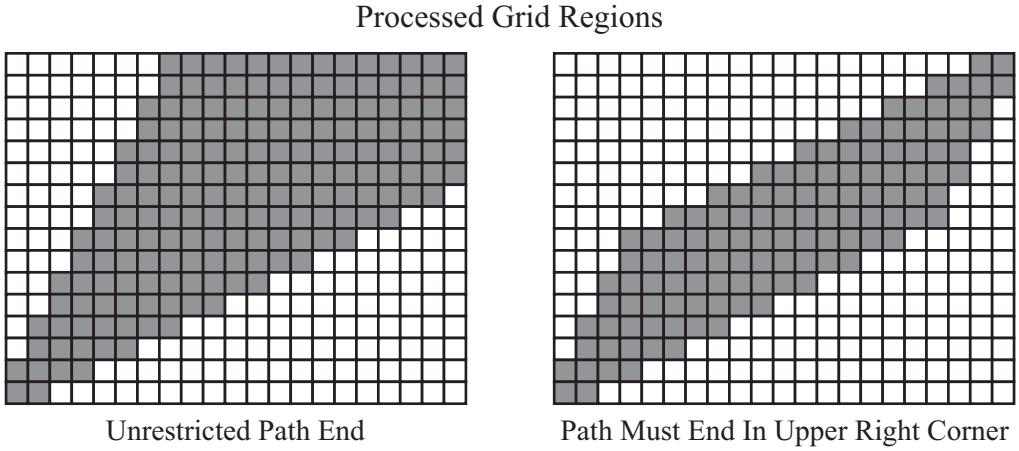

Figure 5.9: Only a subset of the grid needs to be processed, since the slope limits render some cells inaccessible (W=2). **Left:** Processed region with no restrictions on the last cell of the path. **Right:** Processed region when the last cell is required to be (r_1, r_2)

this process can produce superior results. For example, if the goal is to make a transition between two motions that are very dissimilar away from the transition point, then a globally optimal time alignment may be less desirable than one that is only optimal within the vicinity of the transition, since the globally optimal result will be forced to match unrelated frames while the incremental result can effectively ignore frames that are not included in the blend.

The frame correspondences generated by dynamic timewarping could be linearly interpolated to form a continuous mapping between the frames of $\mathbf{M_1}$ and $\mathbf{M_2}$, but the result would not be a proper timewarp curve because it is not in general strictly increasing — multiple frames of one motion may be matched to a single frame of the other. Moreover, while the slope limits place global restrictions on how steep or shallow this pseudo-timewarp curve can be, they do not ensure any smoothness properties. Our algorithm instead creates a timewarp curve by computing a strictly increasing two-dimensional quadratic B-spline with uniformly spaced knots that is minimally distant from the time alignment in a least-squares sense.

Let $p_{i,j}$ be the j^{th} dimension of the i^{th} control point of the desired B-spline. Then for any three consecutive control points $\mathbf{p_i}$, $\mathbf{p_{i+1}}$, and $\mathbf{p_{i+2}}$, the derivative $\frac{dS_j}{du}$ in the associated spline interval is always between $(p_{i+1,j} - p_{i,j})$ and $(p_{i+2,j} - p_{i+1,j})$, and the spline can therefore be forced to be

strictly increasing by requiring $(p_{i+1,j} - p_{i,j}) > 0$ for all i and j. Since a quadratic objective is to be minimized subject to linear inequality constraints, computing the optimal spline is a quadratic programming problem. However, in light of the causality and slope limit restrictions, it is likely that a spline fit without any constraints will be *approximately* strictly increasing. Our strategy is hence to fit a uniform quadratic B-spline as an unconstrained optimization (which is a linear least squares problem) and then adjust the knots. For each j, the knot adjustment algorithm starts at $i = 1$ and iteratively computes

$$\Delta_{i,j} = (p_{i+1,j} - p_{i,j}) - \epsilon, \tag{5.2}$$

where ϵ is a small positive number. If $\Delta_{i,j} \geq 0$, nothing is done. Otherwise, define

$$\lambda = \max\left(\frac{1}{2}, \frac{\Delta_{i-1,j}}{\Delta_{i,j}}\right), \tag{5.3}$$

where $\Delta_{i-1,j}$ is computed using the value of $p_{i-1,j}$ at the beginning of the current iteration. The value of $p_{i,j}$ is reduced by $\lambda\Delta_{i,j}$ and the value of $p_{i+1,j}$ is increased by $(1-\lambda)\,\Delta_{i,j}$. At this point $(p_{k+1,j} - p_{k,j}) \geq \epsilon$ for all k between 1 and i. Equation 5.3 chooses λ so the necessary change is split as evenly as possible between $p_{i,j}$ and $p_{i+1,j}$.

The user may want the first point on the timewarp curve to be exactly $(1, 1)$ and the last point to be exactly (r_1, r_2), so the entire frame ranges of $\mathbf{M_1}$ and $\mathbf{M_2}$ are spanned. If the timewarp curve has k control points, then this can be accomplished by requiring $\frac{1}{2}(\mathbf{p_1} + \mathbf{p_2}) = (1, 1)$ and $\frac{1}{2}(\mathbf{p_{k-1}} + \mathbf{p_k}) = (r_1, r_2)$, which can be enforced by adjusting the control points in a manner similar to what was described in the previous paragraph.

5.2.1.2 Timewarp Curves For More Than Two Motions

Directly applying the methods of Section 5.2.1.1 to arbitrarily many motions is computationally intractable, because generalizing the dynamic programming algorithm to n input motions involves processing an n-dimensional grid, which requires $O(n^r)$ work if each motion has on average r frames. On the other hand, a simple and efficient algorithm can be devised by combining multiple timewarp curves for pairs of motions. Consider building a timewarp curve for three motions $\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{M_3}$, and assume that timewarp curves $\mathbf{S^{1\leftrightarrow 2}}$, $\mathbf{S^{1\leftrightarrow 3}}$, and $\mathbf{S^{2\leftrightarrow 3}}$ have already been

constructed for each pair. If $S_1^{1\leftrightarrow 2}(t_1) = t_2$ and $S_1^{1\leftrightarrow 3}(t_1) = t_3$, then is reasonable to expect that $\{\mathbf{M_1}(t_1), \mathbf{M_2}(t_2), \mathbf{M_3}(t_3)\}$ will be an accurate frame correspondence, which implies that $S_1^{2\leftrightarrow 3}(t_2)$ is approximately t_3 and $S_2^{2\leftrightarrow 3}(t_3)$ is approximately t_2 . Hence $\mathbf{S^{2\leftrightarrow 3}}$ might be discarded and $\mathbf{S^{1\leftrightarrow 2}}$ and $\mathbf{S^{1\leftrightarrow 3}}$ could be merged into a single timewarp curve.

More generally, given a set of motions $\mathbf{M_1}, \ldots, \mathbf{M_n}$, a motion $\mathbf{M_{ref}}$ is selected to serve as a reference. In our implementation, $\mathbf{M_{ref}}$ was chosen to minimize the average distance to the other motions, where the distance between $\mathbf{M_i}$ and $\mathbf{M_j}$ is defined as the average distance between each frame pair of the time alignment. Let $\mathbf{S_i}(u) = (S_{i,1}(u), S_{i,2}(u))$ be the timewarp curve between $\mathbf{M_{ref}}$ and $\mathbf{M_i}$. For convenience and without loss of generality, we assume $S_{i,1}(u)$ returns frame indices of $\mathbf{M_{ref}}$ and $S_{i,2}(u)$ returns corresponding frame indices from $\mathbf{M_i}$. We sample the frame indices of $\mathbf{M_{ref}}$ and for each sample use the inverse functions $u_{S_{i,1}}(t)$ to determine the corresponding frames in each other motion. Specifically, the frame of $\mathbf{M_i}$ that corresponds to $\mathbf{M_{ref}}(t_0)$ is at time $S_{i,2}(u_{S_{i,1}}(t_0))$. The result of this sampling is a set of frame correspondences

$$\left\{\mathbf{M_{ref}}(t_i), \mathbf{M_1}\left(S_{1,2}(u_{S_{1,1}}(t_i))\right), \ldots, \mathbf{M_n}\left(S_{n,2}(u_{S_{n,1}}(t_i))\right)\right\}, \tag{5.4}$$

and the final timewarp curve is formed by fitting to these frame correspondences a strictly increasing n-dimensional quadratic spline with uniformly spaced knots, as in the two-dimensional case.

5.2.2 Alignment Curves

Given a timewarp curve $\mathbf{S}(u)$, an alignment curve $\mathbf{A}(u)$ is constructed by finding rigid $2D$ coordinate transformations that mutually align the frames at each point on $\mathbf{S}(u)$. We first consider the case where there are just two input motions, $\mathbf{M_1}$ and $\mathbf{M_2}$. For the i^{th} frame correspondence $\{\mathbf{M_1}(t_{1_i}), \mathbf{M_2}(t_{2_i})\}$ generated through dynamic programming, Equations 4.2–4.4 of Chapter 4 specify a rigid $2D$ transformation $\{\theta_i, x_{0_i}, z_{0_i}\}$ that, when applied to $\mathbf{M_2}(t_{2_i})$, aligns it with $\mathbf{M_1}(t_{1_i})$. A $3D$ quadratic spline may be fit to these transformations, yielding an alignment curve $\mathbf{A}(u) = (\mathbf{A}_1(u), \mathbf{A}_2(u))$ where every $\mathbf{A}_1(u)$ is the identity transformation and $\mathbf{A}_2(u) = \{\theta(u), x_0(u), z_0(u)\}$ is the result of the spline fit. To avoid angle discontinuities, prior to fitting the spline the sampled rotation parameters should be adjusted so $|\theta_i - \theta_{i-1}| \leq \pi$. Also, there are

sometimes small spikes in the transformation parameters when the dynamic timewarping algorithm shifts from one kind of step (vertical, horizontal, or diagonal) to another. We have found it useful to treat these spikes as impulse noise and remove them with a short median filter (kernel width of 3 to 5 frames) prior to fitting the spline. Finally, it is important that the knot spacing on the spline be sufficiently small that it remains close to the (filtered) samples; we have found that a knot for every 3 to 5 samples yields good results.

The case of n input motions is handled in direct analogy to Section 5.2.1.2. It is assumed that a timewarp curve has been constructed for the entire set of inputs and that pairwise alignment curves have been created between the reference motion $\mathbf{M_{ref}}$ and each other input motion $\mathbf{M_i}$. Frames are sampled from $\mathbf{M_{ref}}$, and for each sample $\mathbf{M_{ref}}(t_j)$ the corresponding frame of each other motion is determined through the timewarp curve. The alignment curves are then used to find transformations $\mathbf{A_i}$ that align the frame of each $\mathbf{M_i}$ with $\mathbf{M_{ref}}(t_j)$. Note that this assumes that $\mathbf{A_k^{-1}A_i}$ is a good approximation of the transformation that directly aligns the frame of $\mathbf{M_i}$ with the frame of $\mathbf{M_k}$. The result of this procedure is a collection of sets of n transformations; at least one transformation of each set is always the identity since it must align a frame of $\mathbf{M_{ref}}$ with itself. Finally, a $3n$-dimensional quadratic spline is fit to these sampled transformations as in the two-motion case.

We note that this method yields smooth alignment curves even when the root paths of the input motions have sharp turns or when they effectively degenerate into single points (such as when the characters are standing in place); see Figure 5.16. This is because the coordinate transformations are determined by the shape of the whole body over a small time window. Clearly, this shape cannot collapse to a point, since the body always has a nonzero extent. Similarly, while the root path may geometrically have a sharp turn, the body shape itself will nonetheless change smoothly as that turn is executed.

5.2.3 Constraint Matches

The final step in building a registration curve is to identify constraint matches. Each kind of constraint is treated independently — for example, left heelplants might be considered first, then

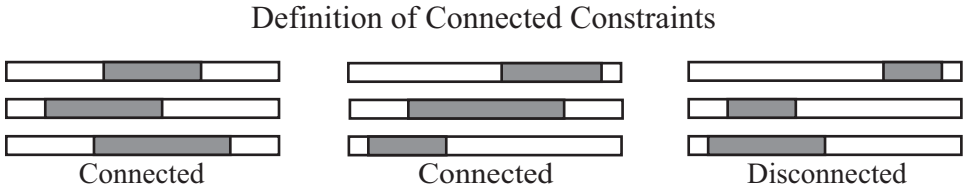

Figure 5.10: Examples of connected and disconnected constraints. Note that a set of constraints can be considered connected even if individual pairs do not overlap, such as in the central figure.

right wristplants, and so on. Also, the time intervals of each kind of constraint are assumed to be disjoint, so if two left heelplants occur respectively in the intervals $[t_1^s, t_1^e]$ and $[t_2^s, t_2^e]$, then $t_1^e < t_2^s$. The algorithm starts by mapping the duration of each constraint into the global time frame via the timewarp curve. Constraints are then grouped into constraint matches based on the heuristic that corresponding constraints should occur at similar points in (global) time. Specifically, there are two guidelines:

1. Each constraint match must contain exactly one constraint from each motion.

2. The elements of each constraint match must be "connected" in the sense that the union of all constraint intervals must form a single continuous interval (Figure 5.10).

Note that some constraints may not belong to any constraint match (for example, a constraint may overlap with no other constraints). In this case that constraint is eliminated. Intuitively, this is because a constraint which is not shared by all of the input motions may be viewed as incidental to the motion that has it, rather than a structural property of the kind of action that is taking place. To illustrate, imagine blending a motion where a character stands on its left leg with a motion where it stands on both legs. One expects the blend to consist of a character standing on its left leg and varying the height of its right foot according to the blend weights. If all of the weight happens to be on the second motion, then both legs will happen to be in contact with the ground, but except for this special case the left foot is logically planted and the right foot is not. It is therefore reasonable to eliminate the plant constraints on the right foot. Using similar reasoning, one or more gaps may be added to a constraint of one motion if it overlaps with multiple constraints of the other motions, thereby splitting it into shorter constraints that can be separately matched.

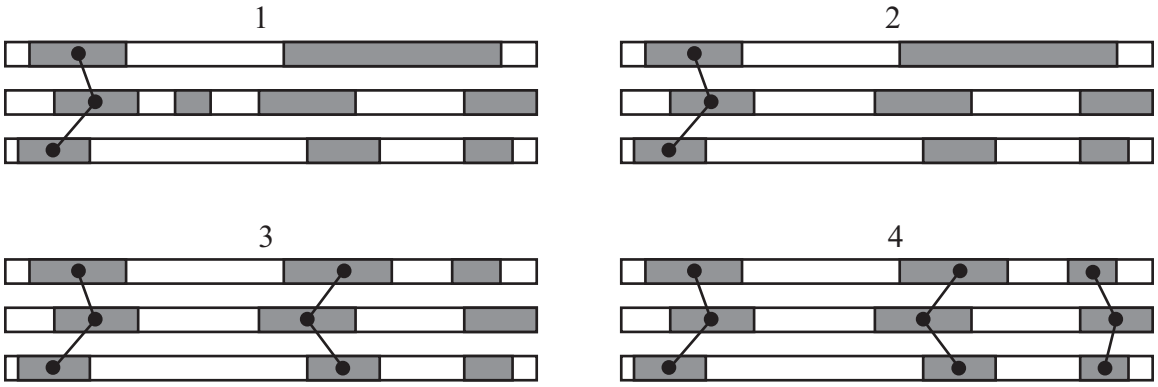

Figure 5.11: The steps taken by the constraint matching algorithm on a sample input. Note that a constraint is removed from the second motion and split in the first motion.

Our constraint matching algorithm proceeds as follows. Let $C_{i,j}$ be the j^{th} constraint of the i^{th} motion. Our algorithm processes the constraints sequentially: $C_{i,j}$ is either eliminated or added to a constraint match before $C_{i,j+1}$ is considered. Each iteration starts by checking whether the earliest unprocessed constraint of each motion are all connected in the sense of Figure 5.10. If not, the constraint which starts the earliest cannot be part of any constraint match, so we discard it and proceed to the next iteration. Otherwise a constraint match can be formed from these constraints. The algorithm first decides whether any of them should be split, and then a constraint match is built from the constraints that were not split and the first portion of the constraints that were split. Figure 5.11 shows the operation of our constraint matching algorithm on a sample input.

The only remaining details are how to determine whether a particular constraint should be split and how to actually perform such a split. Without loss of generality, we limit our discussion to the case where the set of candidate constraints for a constraint match are exactly the first constraint $C_{i,1}$ of each motion. The decision to split is based on *subsumption*: $C_{i,1}$ subsumes $C_{j,1}$ if

1. $C_{i,1}$ overlaps $C_{j,1}$ and $C_{j,2}$ ($C_{j,2}$ must exist)

2. $C_{i,2}$ does not overlap $C_{j,2}$ (possibly because $C_{i,2}$ does not exist)

Figure 5.12 illustrates subsumption. Note that if $C_{i,1}$ subsumes $C_{j,1}$, then $C_{j,1}$ does not subsume $C_{i,1}$. The constraints are partitioned into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ such that every element of $\mathcal{S}_1$ subsumes

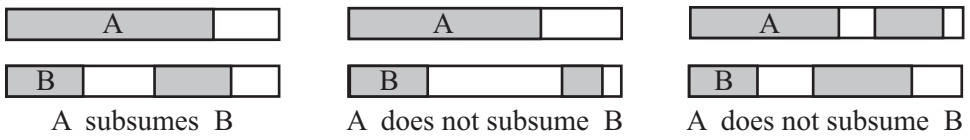

Figure 5.12: Examples of subsumption. In the rightmost case, there would be two separate constraint matches, one containing the first constraint of each motion and the other containing the second constraint of each motion.

every element of $\mathcal{S}_2$; the constraints that will be split are in $\mathcal{S}_1$ and the constraints that will remain intact are in $\mathcal{S}_2$. A candidate partition can be created by first placing a particular constraint $C_{i,1}$ into $\mathcal{S}_1$ and the remaining constraints into $\mathcal{S}_2$. Each constraint in $\mathcal{S}_2$ is next checked to see if it is subsumed by all constraints currently in $\mathcal{S}_1$, and if not then it is switched to $\mathcal{S}_1$. This process iterates until no further changes can be made. If $\mathcal{S}_2$ ends up being empty, nothing should be split, and hence all the constraints are transferred to $\mathcal{S}_2$. The sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are generated in this fashion for each constraint, and our algorithm keeps the partition where the "effective" constraint intervals are as similar as possible. More exactly, if the interval of $C_{i,j}$ in global time is $[u^s_{i,j}, u^e_{i,j}]$, then let I_i be $[u^s_{i,1}, u^e_{i,1}]$ if $C_{i,1} \in \mathcal{S}_1$ and $I_i = [u^s_{i,1}, u^e_{i,2}]$ if $C_{i,1} \in \mathcal{S}_2$. That is, if the constraint is in $\mathcal{S}_2$, then pretend that it terminates at the end of constraint that immediately follows it, since each constraint in $\mathcal{S}_1$ maps to *pairs* of constraints implicitly defined in $\mathcal{S}_2$. The quality of the partition σ is then defined as

$$\sigma = \sum_i \sum_j (I_i \cap I_j), \tag{5.5}$$

where the operator $\cap$ returns the size of the intersection of the two intervals.

Each constraint in $\mathcal{S}_1$ is now split into two shorter constraints separated by a gap; refer to Figure 5.13. Let $C_{i,1} \in \mathcal{S}_1$ be the current constraint that is to be split. Each constraint $C_{j,1} \in \mathcal{S}_2$ votes on where the start and the end of the gap in $C_{i,1}$ should be. First, the end of $C_{j,2}$ is mapped to a point p on $C_{i,1}$: if $C_{j,3}$ exists and $u^s_{j,3} < u^e_{i,1}$, then $p = u^e_{j,2}$, and otherwise $p = u^e_{i,1}$. The vote for the start of the gap is then

$$u^s_{i,1} + \frac{p - u^s_{i,1}}{u^e_{j,1} - u^s_{j,1}}(u^e_{j,1} - u^s_{j,1}) \tag{5.6}$$

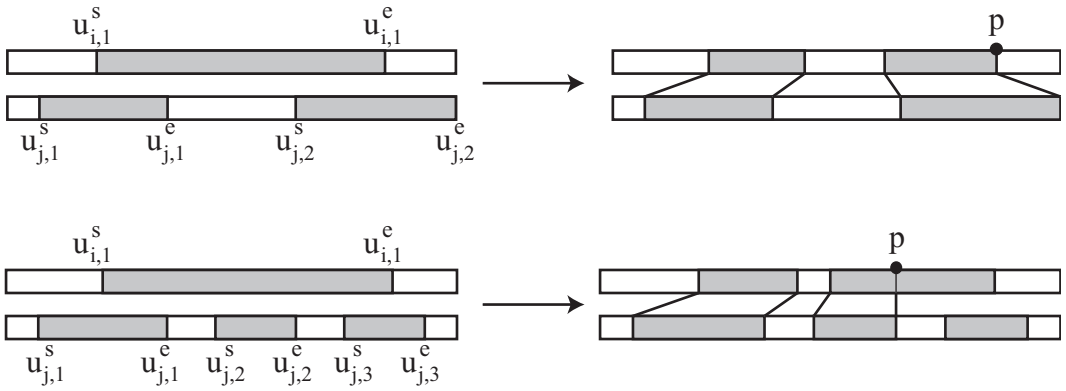

Figure 5.13: Adding a gap to a constraint.

and the vote for the end of the gap is similarly defined. The actual gap in $C_{i,1}$ is determined by averaging the votes.

5.3 Blending with Registration Curves

Creating a single frame $\mathbf{B}(t_i)$ of a blend involves four steps:

1. An appropriate point $\mathbf{S}(u_i)$ on the timewarp curve is calculated.
2. The frames at $\mathbf{S}(u_i)$ are positioned and oriented according to the corresponding point on the alignment curve, $\mathbf{A}(u_i)$.
3. These adjusted frames are combined using the blend weights $\mathbf{w}(t_i)$.
4. Constraints are determined for the blended frame.

We assume that for some blend frame $\mathbf{B}(t_0)$, u_0 is already known. This frame is created first and the other frames are generated in chronological order: first the forward frames ($\mathbf{B}(t_1)$, $\mathbf{B}(t_2)$, ...) and then (if $\mathbf{B}(t_0)$ is not the first blend frame) the backward frames ($\mathbf{B}(t_{-1})$, $\mathbf{B}(t_{-2})$, ...). We require $u(t)$ to be strictly increasing, so advancing forward in blend time t will result in advancing forward in global time u.

The remainder of this section explains how to create a single blend frame. The discussion is limited to the case of generating frames in forward order; moving backward in time is handled similarly.

5.3.1 Advancing Along the Timewarp Curve

If the first blend frame $\mathbf{B}(t_0)$ is being created, then this step is skipped, since u_0 is assumed to be known in advance. Otherwise, the previous blend frame combined input frames at some point $\mathbf{S}(u_{i-1})$ on the timewarp curve, and the goal is now to obtain a new set of input frames by moving to an updated location $\mathbf{S}(u_i)$. To understand how this might be done, imagine that for the remainder of the blend $w_j(t) = 1$ and the remaining blend weights are 0. In this case the rest of $\mathbf{B}$ should be identical to a portion of $\mathbf{M_j}$, and hence the position on the timewarp curve should change such that $\mathbf{M_j}$ is played at its natural rate. Specifically, to advance $\Delta t = t_i - t_{i-1}$ units of time, $\Delta u = u_i - u_{i-1}$ should be chosen such that

$$S_j(u_{i-1} + \Delta u) - S_j(u_{i-1}) = \Delta t \tag{5.7}$$

Taking the limit as $\Delta t \to 0$ and $\Delta u \to 0$,

$$\frac{dS_j}{dt} = \frac{dS_j}{du}\frac{du}{dt} = 1$$

$$\frac{du}{dt} = \left(\frac{dS_j}{du}\right)^{-1} \tag{5.8}$$

For general $\mathbf{w}$(t), we extend this equation to

$$\frac{du}{dt} = \sum_{j=1}^{k} w_j(t) \left(\frac{dS_j}{du}\right)^{-1}. \tag{5.9}$$

Intuitively, each motion votes on how fast the global time parameter u should flow, and these votes are combined according to the blend weights. Note that $\left(\frac{dS_j}{du}\right)^{-1}$ can be expressed analytically, since $S_j(u)$ is a quadratic function of u. In general this differential equation must be solved numerically, but since the time step between frames is small and $\mathbf{w}$ and $\mathbf{S}$ are smooth, we have found

that a single Euler step can be taken to advance one frame in the blend:

$$\Delta u = \left(\sum_{j=1}^{k} w_j(t_{i-1}) \left(\left. \frac{dS_j}{du} \right|_{u=u_{i-1}} \right)^{-1} \right) \Delta t \tag{5.10}$$

A similar strategy was used in [68]. For convex weights, which are commonly viewed as natural choices for blending operations, Equation 5.10 will always yield $\Delta u > 0$ and hence time will always flow forward. If arbitrary blend weights are allowed, then $w_j(t_{i-1})$ can be replaced with $\frac{|w_j(t_{i-1})|}{\sum_r |w_r(t_{i-1})|}$ in Equation 5.10, ensuring $\Delta u > 0$.

5.3.2 Positioning and Orienting Frames

Once u_i has been determined, the frames $\mathbf{M}_j(S_j(u_i))$ are extracted from the input motions, and the root configuration of each frame is transformed by $\mathbf{A}_j(u_i)$. This yields a group of mutually aligned frames that, just as with a single frame of motion, may be translated and rotated in the ground plane by a transformation $\mathbf{T}(t_i)$ (Figure 5.14), in which case the total transformation applied to the root of $\mathbf{M}_j(S_j(u_i))$ is $\mathbf{T}(t_i)\mathbf{A}_j(u_i)$. For the first frame of the blend, $\mathbf{T}(t_0)$ may be chosen arbitrarily. For the other frames, $\mathbf{T}(t_i)$ must be chosen so the position and orientation of $\mathbf{B}(t_i)$ is consistent with the preceding frame $\mathbf{B}(t_{i-1})$. To see how this can be done, assume temporarily that $w_j(t)$ is 1 and the other blend weights are all 0 for $t > t_{i-1}$. The remainder of the blend should then simply be a copy of a portion of $\mathbf{M}_j$, transformed rigidly by $\mathbf{T}(t_{i-1})\mathbf{A}_j(u_{i-1})$ so it connects seamlessly with $\mathbf{B}(t_{i-1})$. If $\Delta\mathbf{T}_j(t_i)$ is defined as

$$\Delta\mathbf{T}_j(t_i) = \mathbf{T}(t_{i-1})\mathbf{A}_j(u_{i-1}){\mathbf{A}_j}^{-1}(u_i), \tag{5.11}$$

then setting $\mathbf{T}(t_i) = \Delta\mathbf{T}_j(t_i)$ yields this result.

More generally, each motion $\mathbf{M_j}$ votes for the $\mathbf{T}(t_i)$ that leaves its local coordinate system unchanged (namely, $\Delta\mathbf{T}_j(t_i)$), and these votes are averaged according to the blend weights, as depicted in Figure 5.14. The first step in this process is to choose appropriate parameters to represent $\Delta\mathbf{T}_j(t_i)$. To locate an origin that is near the center of the transformed frames, each $\mathbf{M}_j(S_j(u_i))$ is transformed by $\Delta\mathbf{T}_j(t_i)\mathbf{A}_j(u_{i-1})$, and the new root positions are then projected onto the ground

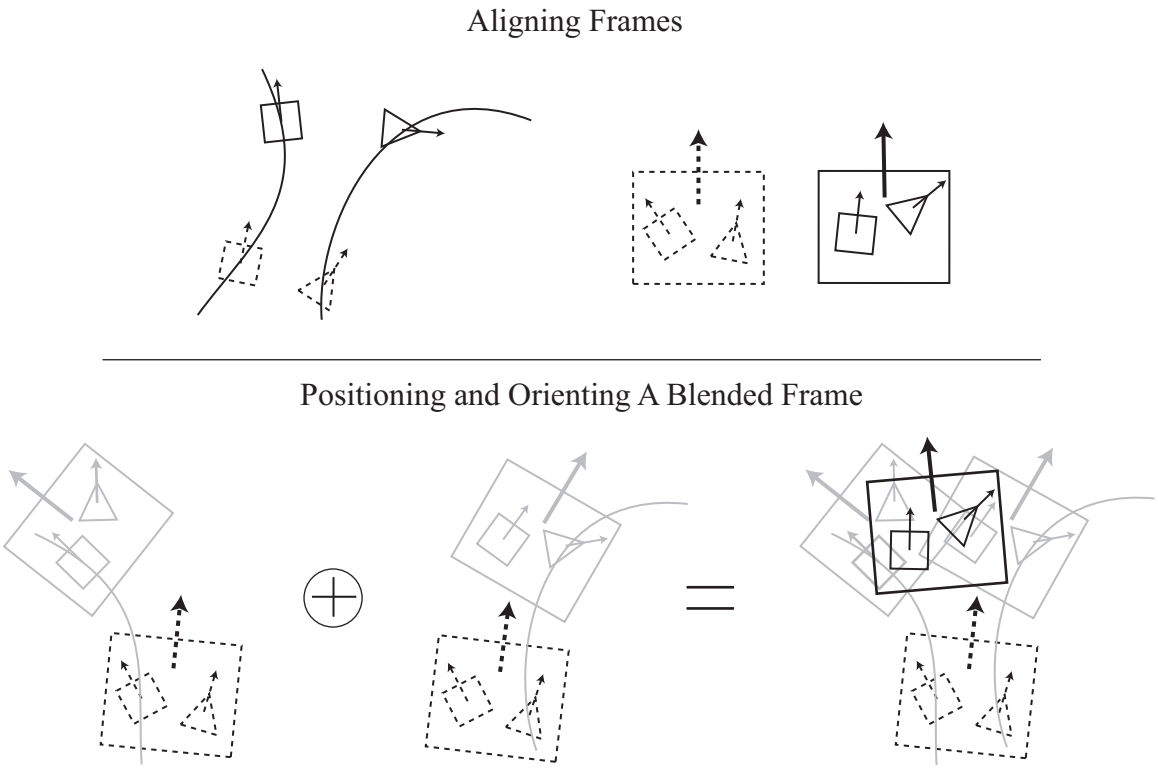

Figure 5.14: Top: Given any set of corresponding frames, the alignment curve specifies rigid $2D$ transformations that mutually align them. The curved lines on the left show the root trajectory of two motions, and the root configurations are indicated for two pairs of corresponding frames: the solid square and solid triangle, and the dashed square and dashed triangle. The right shows what these root configurations might look like when the frames are aligned. Each group of aligned frames can itself be rigidly transformed. **Bottom:** Based on the position and orientation of the previous blend frame, each motion votes on a new position and orientation for the current blend frame. These votes are combined according to the blend weights; here we show the case $w_1 = w_2 = 0.5$.

and averaged. $\Delta\mathbf{T}_j(t_i)$ is then represented by the parameter set $\{\phi_j, x_j, z_j\}$, which corresponds to a rotation by ϕ_j about this origin followed by a translation (x_j, z_j). $\mathbf{T}(t_i)$ is then defined as

$$\mathbf{T}(t_i) = \left\{ \sum_j w_j \phi_j, \sum_j w_j x_j, \sum_j w_j z_j \right\}. \tag{5.12}$$

5.3.3 Creating the Blended Frame

Having extracted frames from the input motions and placed them in the appropriate positions and orientations, the blended skeletal pose is formed by using the current blend weights to compute a weighted average of the input joint parameters. To average orientations, we originally adopted the algorithm advocated by Park et al [68]. Their method first computes a reference orientation $\mathbf{q_{ref}}$ that minimizes a distance measure to the input orientations $\mathbf{q_i}$. Each $\mathbf{q_i}$ is then transformed to the local coordinate frame of $\mathbf{q_{ref}}$ and converted to a logarithmic map representation, and these logarithmic maps are averaged. Finally, the average orientation $\overline{\mathbf{q}}$ is found by taking the exponential map and transforming back to the global coordinate frame:

$$\overline{\mathbf{q}} = \mathbf{q_{ref}}\mathbf{exp}\left(\sum_i w_i \mathbf{log}\left(\mathbf{q_{ref}^{-1}}\mathbf{q_i}\right)\right) \tag{5.13}$$

However, we subsequently determined that nearly identical results could be obtained in practice by placing each orientation on the same hemisphere of the quaternion sphere (i.e., replacing $\mathbf{q_i}$ with its antipode if $\mathbf{q_i} \cdot \mathbf{q_1} < 0$), averaging their coordinates like ordinary 4-vectors, and normalizing the result. Using this simpler operation reduced the total time needed to make a blend by a factor of three.

The final task is to determine the constraints on the new blend frame. For each constraint match $\mathcal{M}$, an interval $I_{\mathcal{M}}$ for the corresponding constraint is determined based on the current blend weights. Specifically, if the j^{th} constraint of $\mathcal{M}$ is active over the interval $[u_j^s, u_j^e]$, then

$$I_{\mathcal{M}} = \left[\sum_j w_j(t_i)u_j^s,\ \ \sum_j w_j(t_i)u_j^e\right] \tag{5.14}$$

If u_i is inside this interval, then the new blend frame is annotated with the appropriate constraint. Actually enforcing the constraints is up to the implementation; we used the method described in Chapter 3.

5.4 Results and Applications

Building a registration curve for n motions that are m frames each takes $O(m^2n^2)$ time, since timewarp and alignment curves must be generated for each pair of motions, and constructing the former requires solving an $O(m^2)$ dynamic programming problem (unless one uses the approximate algorithm discussed in Section 5.2.1.1, in which case it only takes $O(m)$ time). The largest registration curve we constructed in our experiments was composed of 18 motions with an average of 300 frames each (about 10s at 30Hz, or 3 minutes of data total). Using the $O(m^2)$ dynamic programming algorithm, it took 3.3s to construct this registration curve on a machine with a 1.3GHz Athlon processor. This time should be viewed as a preprocessing step, since a registration curve can be computed once and stored for use in multiple blending operations. The amount of space needed for a registration curve is $O(mn)$ (i.e., proportional to the total number of frames), since all that must be stored are the control points of the timewarp curve, the control points of the alignment curve, and the constraint intervals of each constraint match. On average the size of this information on disk was 1%–2% of the original data. The time needed to create a blend is also $O(mn)$, and computing a full 300-frame blend of the 18 motions mentioned earlier took about 0.3s.

Registration curves can be used as a back end for common blending applications. The rest of this section discusses using registration curves for transitions, interpolations, and continuous motion control. Videos are available at http://www.cs.wisc.edu/Gallery/Kovar/RegistrationCurves/.

5.4.1 Transitions

To create a transition, our algorithm must know its location and duration. The location can be conveniently specified through the central frames $\mathbf{M}_1(t_i)$ and $\mathbf{M}_2(t_j)$ — that is, if the transition were created simply by cutting from $\mathbf{M}_1$ to $\mathbf{M}_2$, then $\mathbf{M}_1(t_i)$ would immediately precede $\mathbf{M}_2(t_j)$. The duration of the transition can be given through its half-width h, in which case it spans a total of $2h + 1$ frames and the central frame is an equally weighted combination of $\mathbf{M}_1(t_i)$ and $\mathbf{M}_2(t_j)$.

When creating the registration curve, the grid cell $(\mathbf{M}_1(t_i), \mathbf{M}_2(t_j))$ serves as the starting point for the dynamic timewarping algorithm, and the timewarp curve is generated forwards and

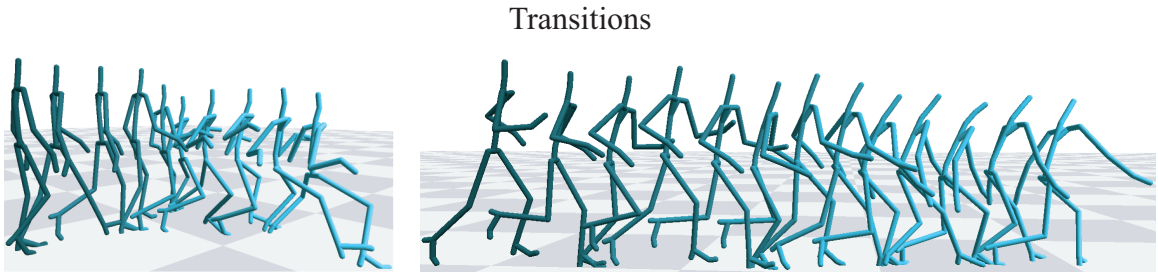

Figure 5.15: Transitions between different kinds of locomotion. Both transitions span two locomotion cycles. **Left:** Normal walking to a stylized walk. **Right:** Jogging to sneaking.

backwards from this point. When creating the blend, the starting position $\mathbf{S}(u_0)$ on the time-warp curve is found by averaging the u values that correspond to $\mathbf{M}_1(t_i)$ and $\mathbf{M}_2(t_j)$: $u_0 = \frac{1}{2}(u_{S_1}(t_i) + u_{S_2}(t_j))$. Once this central frame is created, the remainder of the blend is formed by generating h frames forward in time and then h frames backward in time. Blend weights that smoothly change from $(1, 0)$ to $(0, 1)$ are generated through the function defined in Equation 3.5 of Chapter 3. Figure 5.15 shows example transitions generated by our system.

In general, the first and last frames of the transition will not occur at integer times, and so $\mathbf{M}_1$ and $\mathbf{M}_2$ must be resampled if they are to connect smoothly to the transition boundaries. If this is undesirable, the boundaries can be forced to occur at integer times by generating $u_{-h}, \ldots, u_h$ as described in Section 5.3.1, applying a smooth displacement map that rounds u_{-h} and u_h to the nearest integers, and using these new values in place of the originals.

Registration curves could readily be used to generate transitions in the motion graph construction algorithm presented in Chapter 4. However, the transition times in this case are so short (0.5s, compared with 1-2s in the examples in this chapter) that we have found linear blending to suffice — indeed, we were unable to find a specific case where registration curves yielded clearly superior results. Intuitively, this is because in short transitions there is not enough time for motions to get sufficiently out of phase or for their root trajectories to diverge sufficiently that registration curves are necessary. On the other hand, registration curves could potentially be used to automatically build motion graphs where transition durations span a wide range of values instead of being fixed.

5.4.2 Interpolations

A set of input motions can be interpolated by using fixed blend weights. Our algorithm assumes that the entirety of each input motion is to be blended, and therefore the timewarp curve (and each pairwise time alignment) is forced to pass through the first frame of each motion and the last frame of each motion. The blend itself is created by setting u_0 to be the first point on the timewarp curve and stepping forward until the end of the timewarp curve is reached. Figures 5.16 and 5.17 show interpolations of several sets of input motions.

One of the principle applications of motion interpolation is the construction of parameterized motions. Parameterized motions are considered in more detail in Chapter 6, where methods are presented for automatically identifying and extracting variants of a given action from a data set (that is, obtaining the initial examples) and for building accurate maps between blend weights and relevant motion features.

Figure 5.16: Interpolations of three kicks are randomly sampled to create a large number of similar but distinct kicks.

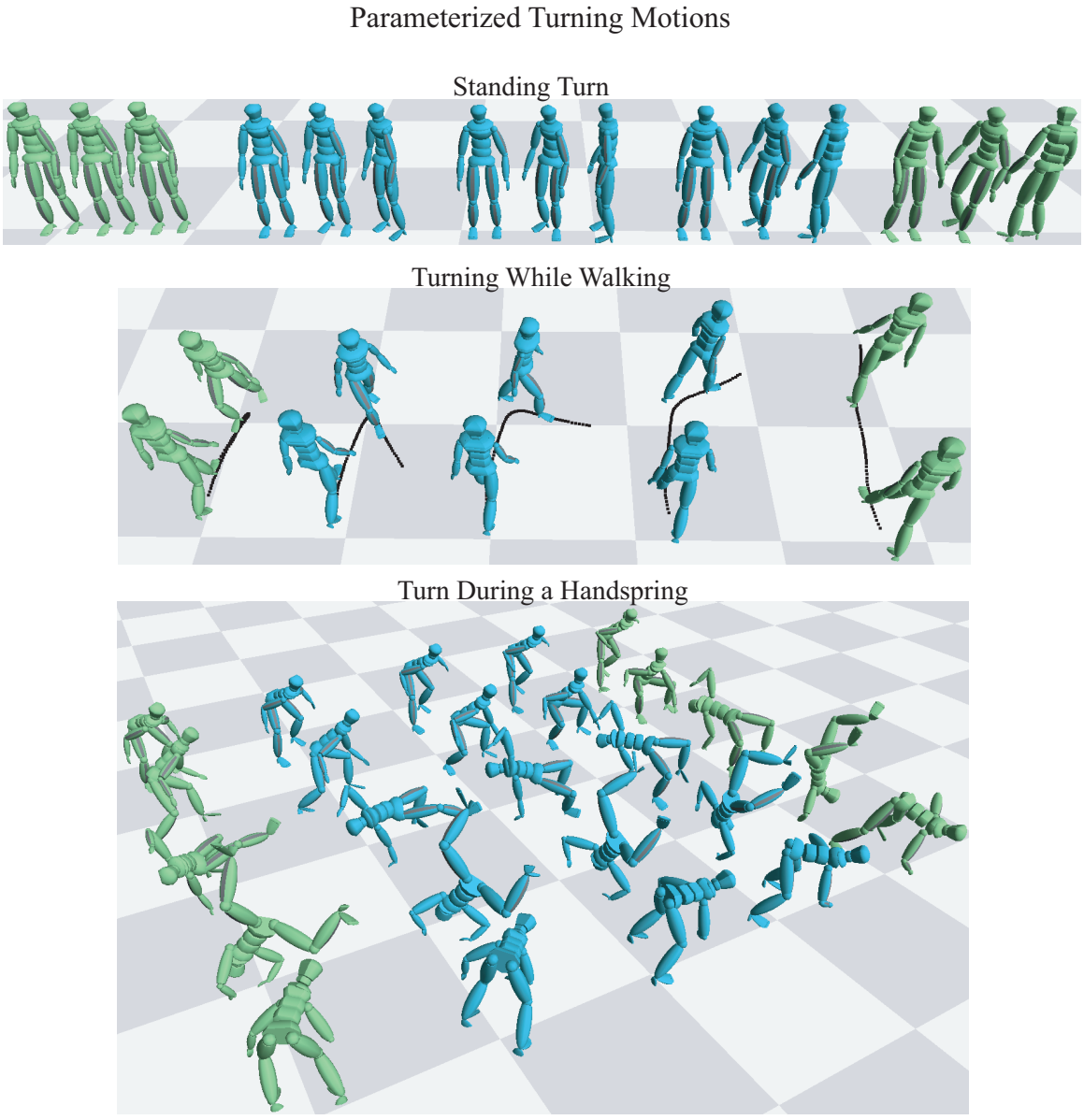

Figure 5.17: Parameterized motions created through interpolation. Original motions are green and interpolations are blue. In each case the parameterization is on the amount the character turns. Blends such as the center one are not possible with existing blending algorithms (even manual ones), because these algorithms can not handle input motions with sharply varying root trajectories.

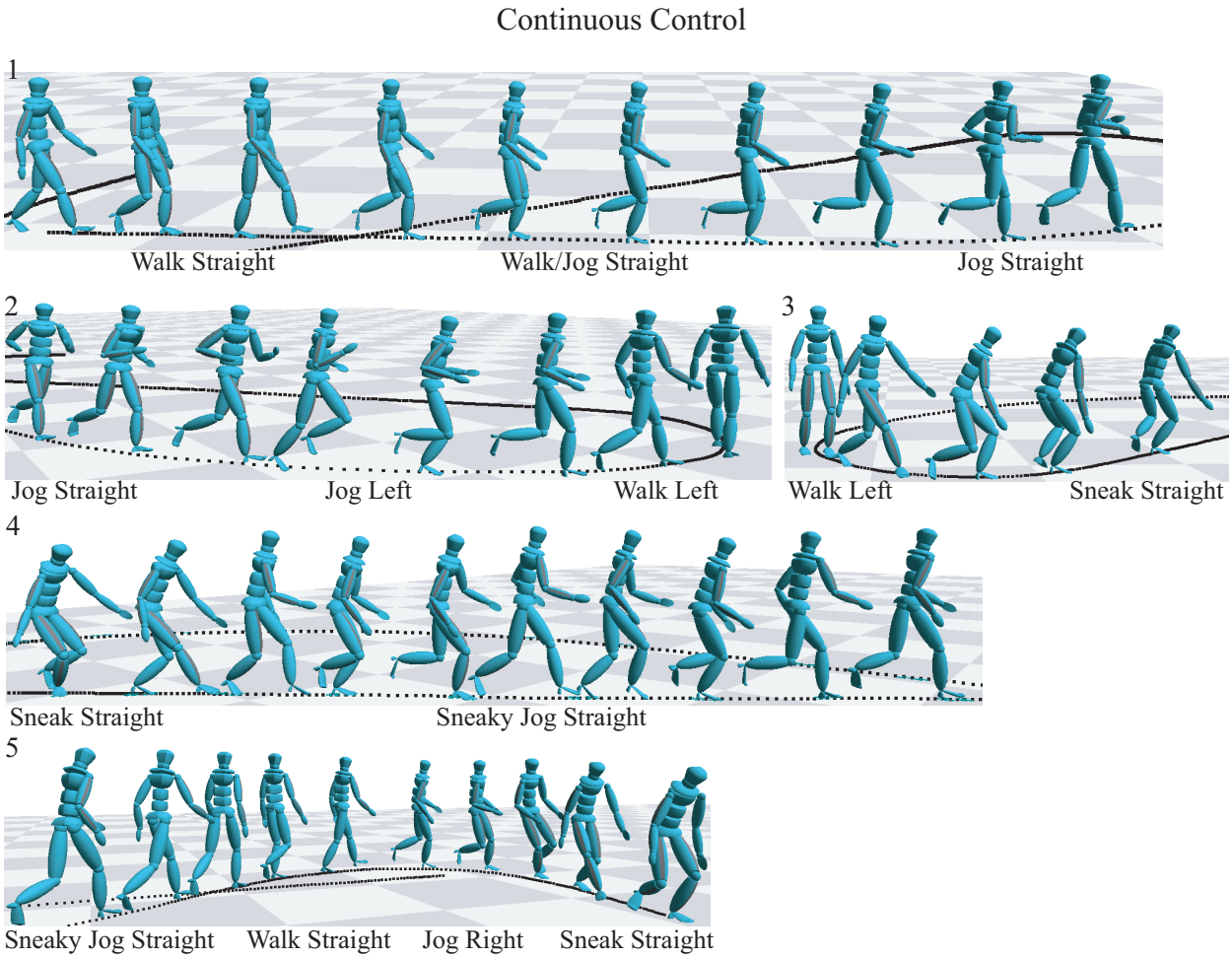

Figure 5.18: Starting with nine variations of locomotion, a registration curve was built and the speed, direction, and style of a character's movement was controlled by continuously varying blend weights. The synthesized motion has been split across several images, as it was too long to be displayed in a single picture.

5.4.3 Continuous Motion Control

For certain sets of input of motions, it makes sense to use blend weights that vary continuously. This can be implemented by generalizing interpolation to use an arbitrary user-specified weight function, rather than a constant function. To demonstrate this application, we obtained a set of nine walking, jogging, and sneaking motions; for each locomotion style there was a motion that travelled straight ahead, one that curved to the left, and one that curved to the right. A registration

curve was constructed and used to continuously control the speed, curvature, and "sneakiness" of a character. An example motion is shown in Figure 5.18.

5.5 Discussion

This chapter has introduced registration curves, a data structure that can be used to automatically generate blends of an arbitrary number of input motions. Registration curves significantly expand the range of motions that can be blended with automatic methods by encapsulating relationships between the timing, root trajectory, and constraint state of the input motions. Moreover, registration curves can seamlessly serve as a back end for common blending operations such as transitioning, interpolation, and continuous control.

As with any blending method, our algorithm is not guaranteed to create realistic blends for arbitrary inputs, and it is therefore important to understand when it is likely to succeed and when it will probably fail. Registration curves assume that structurally related parts of motions look more similar than unrelated parts. In many cases this assumption is valid. For example, corresponding skeletal poses of a locomotion cycle do in fact look more similar than skeletal poses which are out of phase. Similarly, registration curves are appropriate for motions which perform the same action in different styles, such as casually picking up a glass versus forcefully snatching it. On the other hand, in some cases logically corresponding parts of motions have the most *dissimilar* poses. Imagine two motions where a character picks up an object, one where it must stand on its tiptoes to reach to an upper shelf and another where it must bend down to reach near the ground. While the apexes of these reaches are logically identical, these are also the most dissimilar poses in the two motions, and the timewarp curve generated by our method would explicitly avoid matching these frames. However, this does not imply that our framework is unable to handle motions like reaching. If the set of desired motions is sampled sufficiently densely, then for reaches that target nearby locations it will in fact be true that poses at corresponding parts of the reach look the most similar, and correspondences between more distant motions could be inferred using intermediary motions. This idea will be explored in greater depth in Chapter 6.

Although arbitrary blend weights may be specified, some care should be exercised to ensure that synthesized motions retain the quality of the original examples. In our experience convex blend weights are the safest, although modest deviations from convexity can add some extra flexibility without unduly affecting motion quality. It is also important that blend weights not be changed too fast (i.e., during a transition or continuous control), since this effectively introduces a discontinuity into the blend. Acceptable rates of change must ultimately be determined by the application, since one might be willing to accept lower motion quality in order to improve responsiveness.

While blending is more likely to succeed when the input motions are similar (or, at least, when they sample the intended range of variation with reasonable density), ultimately there is no way to be certain without actually computing and evaluating specific blends. One of the biggest advantages of registration curves is that they simplify this process by allowing a user to almost immediately begin experimenting with different blend weights — even for comparatively large input sets, a registration curve can be generated in just a few seconds (Section 5.4), and entire blends can then be computed at interactive rates.

Chapter 6

Parameterized Motion

Many actions are naturally described through a relatively small number of high-level features. For example, a punch might be characterized by its speed and target location, and a walk cycle might be characterized by the final position and orientation of the character relative to its initial configuration. It would be convenient to animate these kinds of motions simply by stating what these features should be, rather than by adjusting low-level properties such as joint orientations. One way to accomplish this is to collect multiple variations of a motion (typically through motion capture) and then using blending methods like the one discussed in Chapter 5 to create interpolations. The set of interpolations is a continuous space of related motions, and if the interpolation weights can be mapped to relevant motion features, then the result is a *parameterized motion* that provides the intuitive control we seek.

Existing methods for constructing parameterized motions are designed for small data sets consisting exactly of example motions that uniformly (though perhaps not regularly) sample a predetermined range of variation [96, 76, 78, 68]. For instance, if one wanted a character that could reach to any location inside of a square, then the data set would consist of several reaching motions (separated into individual data files with analogous starting and ending poses) that target positions distributed evenly within that square. Most real-world data sets are not so contrived. At the very least, the data is almost never limited to a single kind of action, since characters generally must be able to perform a variety of tasks. It is also common for a given data file to contain more than just a single, logically independent action. If nothing else, the action of interest is usually surrounded

by "filler" where the actor prepares for or recovers from the movement, and a single data file might contain multiple actions because a continuous sequence of motion is often more natural than performing individual movements in isolation. To extract variations of a specific action, at present there is little recourse but to scan through the data and manually identify the start and end frames of each example motion segment. This can be tedious and time consuming, even when the data is annotated with descriptive labels. For example, a data file labelled "punching" might contain both several individual punches and related but distinct actions such as dodging a counter-blow. Also, in general one does not know in advance what range of variation is spanned by the data. Indeed, an important step in analyzing the utility of a data set is determining what sorts of actions can be derived from the available examples.

This chapter provides automated tools for constructing parameterized motions from large data sets. Specifically, this chapter introduces a method for extracting logically similar motions segments from a data set (i.e., the example motions) and shows how to construct efficient and accurate parameterizations of the space of interpolations. We start in Section 6.1 with an overview of our methods and a summary of our contributions. Section 6.2 then presents an algorithm for efficiently searching a data set for motion segments similar to a query motion, and it analyzes experiments performed on a test data set containing 37,000 frames (about ten minutes of motion sampled at 60Hz). Next, Section 6.3 introduces an algorithm for creating accurate parameterizations of the space of interpolations and demonstrates this algorithm on several examples. Finally, Section 6.4 concludes with a brief discussion of the advantages and limitations of our methods.

6.1 Overview

6.1.1 Searching Motion Data Sets

Given a segment of the data set (the *query*), our system automatically locates and extracts motion segments that are "similar", i.e., that represent variations of the same action or sequence of actions. Our method uses three key ideas, each of which is a contribution toward solving the problem of searching a motion data set.

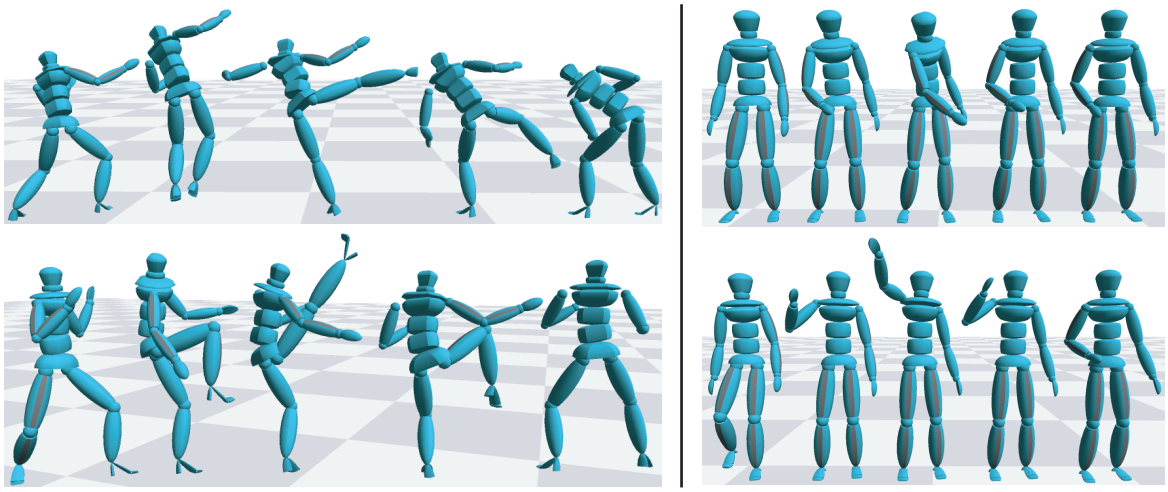

Figure 6.1: Logically similar motions may be numerically dissimilar. **Left:** A standing front kick vs. a leaping side kick. Note the differences in the arms, torso posture, and kick trajectory. **Right:** While these two reaching motions have somewhat similar skeletal postures, the changes in posture are in completely opposite directions.

1. **Multi-step search.** Existing search methods determine motion similarity through direct numerical comparison; i.e., they compare the "distance" between the motions against a threshold. This strategy can only reliably find motions that are *numerically* similar to the query in the sense that corresponding skeletal poses are roughly the same — a large distance may reflect either that motions are unrelated or that they are different variations of the same action (Figure 6.1), and there is no way to distinguish between these cases. On the other hand, in a large data set it is likely that some logically similar motions will also be numerically similar. We therefore add robustness to the search by concentrating on finding these closer motions and then using them as new queries in order to find more distant motions.

2. **Using time correspondences to determine similarity.** A simple way of measuring numerical similarity is to identify corresponding frames and compare their average distance to a threshold value. We complement this by analyzing the correspondences themselves: similar motions are required to have "clear" time correspondences. This is based on the intuition that if two motions are similar, it should be easy to pick out corresponding events.

3. **Interactivity through precomputation.** To provide interactive speeds, we do not start each search from scratch. Instead, we precompute a *match web*, which is a compact and efficiently searchable representation of all possibly similar motion segments.

6.1.2 Creating Parameterized Motions

Once example motions have been collected, they can be interpolated using the methods of Chapter 5, and the space of interpolations can be converted into a parameterized motion through a user-specified *parameterization function* $\mathbf{f}$ that picks out relevant motion features. For example, $\mathbf{f}$ might compute the position of the hand at the apex of a reach or the average speed and curvature of the root path during a walk cycle. Abstractly, $\mathbf{f}$ maps interpolation weights to motion parameters, and our goal is to compute $\mathbf{f}^{-1}$: given a set of target parameters, we want interpolation weights that produce the appropriate motion. Since $\mathbf{f}^{-1}$ in general has no closed form representation, it is common to approximate it with scattered data interpolation[1] methods that assign each example motion a weight based on the distance between its parameters and the target parameters [76]. We adopt a similar approach, but offer several improvements over previous work:

1. **Interpolation Weight Constraints.** Interpolation can only reliably create new motions near the examples, which means only a finite region of parameter space is accessible. In particular, interpolation weights should be convex or nearly convex in order to prevent unreasonable amounts of extrapolation. Existing methods place no constraints on interpolation weights and can break down when specified parameters are far from those of the example motions. Our algorithm ensures that interpolation weights are "almost" convex, and the user can directly control the allowable deviation from strict convexity. This limits the amount of extrapolation performed during motion synthesis and effectively projects unattainable parameter requests to be projected back onto the accessible portion of parameter space.

[1]The phrases "scattered data interpolation" and "motion interpolation" should not be confused: the former is a general method for inferring function values in between measurements, and the latter is a motion synthesis technique (namely, blending with constant weights).

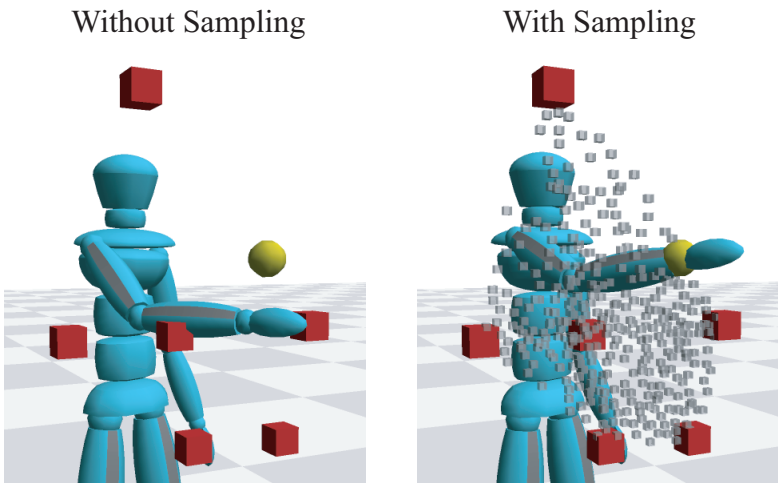

Figure 6.2: Left: Six example reaching motions create a sparse sampling of parameter space that leads to an inaccurate parameterization. The red cubes indicate parameters (i.e., wrist locations) of the example motions and the yellow sphere shows the desired location of the wrist. **Right:** We automatically generate a denser sampling that provides greater accuracy; grey cubes represent new samples.

2. **Accuracy.** If the examples are not sufficiently close in parameter space, direct application of scattered data interpolation may yield inaccurate results (Figure 6.2). We correct this by automatically sampling the set of valid interpolations in order to densely sample the accessible region of parameter space.

3. **Efficiency.** Large data sets may contain many example motions. We show how to automatically identify and remove redundant examples to reduce storage requirements. Also, while previous methods have used scattered data interpolation algorithms that require $O(n)$ time for n examples, our method's run time is nearly independent of the number of examples.

6.2 Searching For Motion

This section considers the problem of searching a motion data set for motion segments (called *matches*) that are similar to a query motion $\mathbf{M_q}$, which is itself assumed to be a segment of some motion in the data set. One reason this problem is challenging is because individual frames of motion are high-dimensional objects with non-Euclidean distance metrics (e.g., Equation 4.1 of

Chapter 4). As a result, traditional methods for organizing the data into a spatial hierarchy (such as a BSP-tree) can not be directly applied [12]. A second challenge is that logically similar motions may be numerically dissimilar in the sense that corresponding poses may have very different joint orientations and angular velocities (Figure 6.1). Traditional search algorithms implicitly equate numerical similarity with logical similarity, and as a result they have difficulty distinguishing motions that are unrelated from those that are different versions of the same kind of action.

Our search strategy is to find "close" matches that are numerically similar to the query and then use them as new queries to find more distant matches. Our algorithm for determining numerical similarity allows arbitrary metrics for comparing individual frames, and timing differences are factored out to allow matches to be of different duration than the query. A user executes a search by providing a query and a distance threshold. Numerically similar matches are identified based on this distance threshold, and these matches are automatically submitted as new queries. This process iterates until no new matches are found. Since each match spawns a new query, it is crucial that individual queries be processed quickly. In particular, we would like searching to be fast enough that the distance threshold can be tuned interactively. To make this feasible, the data set is preprocessed into a *match web*, which is a compact and efficiently searchable representation of all motion segments that, given a sufficiently large distance threshold, would be considered numerically similar.

One difficulty is that numerically similar motions might *not* be logically similar. For example, the overall structure of a man walking looks much like that of a woman walking, and reaching for an object appears quite similar to simply touching it. This problem is hard to correct because it involves high-level understanding of what motions mean. For an unlabelled data set, users must independently confirm that matches have the correct meaning. However, most large data sets contain some sort of descriptive labels — if nothing else, a filename or location in the directory hierarchy can serve as clues to a motion's contents. When labels are present, our algorithm restricts its search to semantically relevant parts of the data set.

The remainder of this section proposes criteria for determining whether two motion segments are numerically similar, explains how to build match webs and how to use them to quickly answer similarity queries, and presents experimental results.

6.2.1 Criteria for Numerical Similarity

Two criteria are used to determine whether two motion segments are numerically similar:

1. Corresponding frames should have similar skeleton poses.

2. Frame correspondences should be easy to identify. That is, related events in the motions should be clearly recognizable.

Frame correspondences are generated using the dynamic programming algorithm discussed in Chapter 5. As before, the set of frame correspondences is called a *time alignment*, and time alignments are required to be continuous, monotonic, and non-degenerate. Recall that if a grid is formed where cell (i, j) specifies the distance $D\left(\mathbf{M}_1(t_i), \mathbf{M}_2(t_j)\right)$ between frames $\mathbf{M_1}(t_i)$ and $\mathbf{M_2}(t_j)$, then the time alignment is a path on this grid from the lower left to the upper right such that the total cost of its cells is minimized.

To test the first criterion, the average value of the cells on the time alignment is compared against a user-specified threshold ϵ. The average value is used instead of the total in order to factor out differences in path length. The second criterion can be interpreted in terms of the *local* optimality of the time alignment. If a cell on the time alignment is a horizontal or vertical 1D local minimum, then the frame correspondence is strong in the sense that holding one frame fixed and varying the other only yields more dissimilar skeletal poses. To illustrate, consider the top of Figure 6.3, which shows the distance grid for two different walk cycles. The magenta path on the right is the calculated time alignment and the highlighted cells on the right show all of the horizontal and vertical 1D minima. Note that nearly every cell on the time alignment is one of these minima. This is because frames at the same point in the locomotion cycle are far more similar than frames that are out of phase. The bottom of Figure 6.3 repeats this analysis for two kicking motions. While the average distance between corresponding frames is about 5 times higher, frames

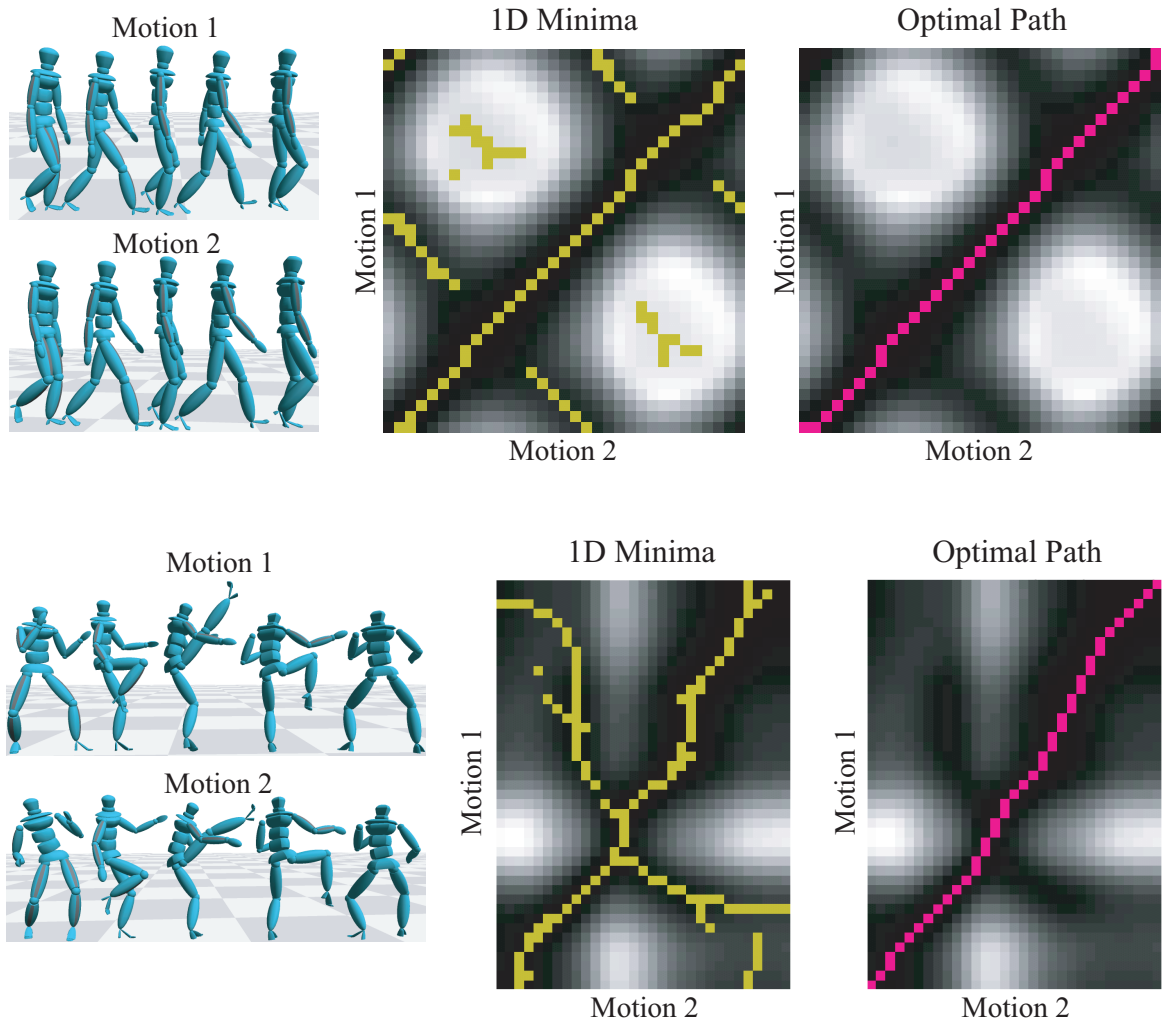

Figure 6.3: Comparing time alignments (yellow cells on the left) with local minima locations (magenta cells on the right). Darker pixels show smaller frame distances. Local minima locations were *not* used when computing the time alignments. **Top:** Two walk cycles. **Bottom:** Two kicks.

at related parts of the kick (chambering, extension, and retraction) are still more similar than pairs of unrelated frames. This is again reflected in the local minima of D: about 60% of the time alignment's cells are 1D minima, and the rest are close to 1D minima.

Ideally every cell on the time alignment would be a local minimum, since then each correspondence would be "obvious". In practice, however, this is overly restrictive. The finite sampling rate causes noise in the exact location of minima, and motions that have stretches of similar frames

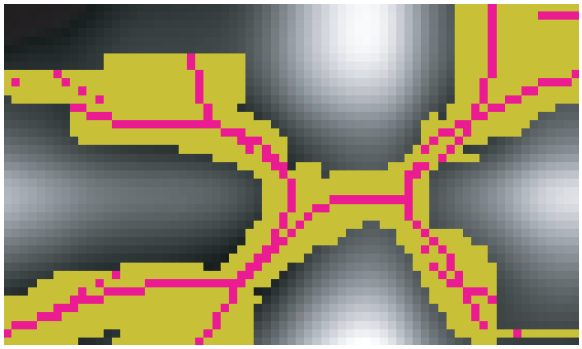

Figure 6.4: Local minima (magenta) are extended to form the valid region (yellow).

(e.g., pauses) produce basins in the distance grid where the locations of minima are effectively arbitrary. We therefore instead require cells on the time alignment to be *near* 1D minima. To enforce this, we compute all 1D local minima and extend each one along the directions in which it is a minimum (horizontal, vertical, or both) until a cell is encountered whose value is at least α percent larger than the minimum's value, where α is defined by the user (see Figure 6.4; we set α to 15%). We call the resulting region on the grid the *valid region*. The time alignment is restricted to lie within the valid region, which can be implemented by modifying the dynamic programming algorithm so it only processes cells in the corresponding portion of the grid. If no time alignment can be created under this restriction, then the motions fail the second criterion and are considered dissimilar.

6.2.2 Match Webs

We now turn to the problem of precomputing potential matches for every motion segment in the database. To illustrate the issues involved, consider a brute force comparison of every possible pair of motion segments. Assume that the query is restricted to be at most m frames and that a distance threshold is provided in advance. The goal is to build a lookup table that lists all the matches for each possible query. If the data set has n frames total ($n \gg m$), then there are $O(mn)$ possible queries, since a query has n possible starting frames and m possible durations (neglecting boundary effects). Accounting for the slope limits used when computing time alignments, a $\tilde{m}$-frame query has $\left(W - \frac{1}{W}\right) \tilde{m}n$ candidate matches, and since the time needed to build a time alignment between

an r-frame motion and an s-frame motion is $O(rs)$, $O(\tilde{m}^3 n)$ work is needed to evaluate all of these candidate matches. The total amount of time needed to process all possible queries is therefore $O(m^4 n^2)$.

This brute force approach suffers from several problems. First, the computational cost is prohibitive, even for small data sets and short query clips. For example, using a data set with just 10 seconds of motion sampled at 60Hz and a maximum query length of 1.5s, a brute force comparison took over 16.5 hours on a 1.3GHz Athlon processor. Second, the results are redundant: if a particular motion segment is numerically similar to the query, then small perturbations of its start and end times will probably yield additional "similar" motion segments (indeed, this redundancy plagues existing efficient search algorithms [22, 92, 16, 44]). Finally, searches at run time are needlessly restricted to a maximum query length and a fixed distance threshold.

Abstractly, a brute force comparison of all possible motion segments is an analysis of the grid of distances formed by computing D for every pair of frames in the data set. Specifically, comparing any two motion segments amounts to solving a dynamic programming problem on a subsection of this grid. The problems mentioned in the previous paragraph stem from the highly redundant nature of this analysis: "nearby" pairs of motion segments are processed independently, even through the corresponding regions of the distance grid may overlap considerably. Through a more careful analysis of the distance grid, we will show that one can efficiently build a compact representation of all possibly similar motion segments, without placing any artificial restrictions on run-time queries.

Without loss of generality, we limit the discussion to finding segments in a single motion $\mathbf{M}_1$ that are numerically similar to another (possibly identical) motion $\mathbf{M}_2$; if a data set has many motions, then they are compared pairwise. Let $\mathbf{M}_i[q, r]$ denote the motion segment $\mathbf{M}_i(t_q), \ldots, \mathbf{M}_i(t_r)$. Two motion segments $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ are *potentially similar* if there is some distance threshold for which they are numerically similar under the criteria of Section 6.2.1. For this to hold, there must exist at least one time alignment starting at cell (a, c) and ending at cell (b, d) that is everywhere inside the valid region and obeys the continuity, monotonicity, and non-degeneracy restrictions. If there is no such time alignment, then $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ cannot be numerically

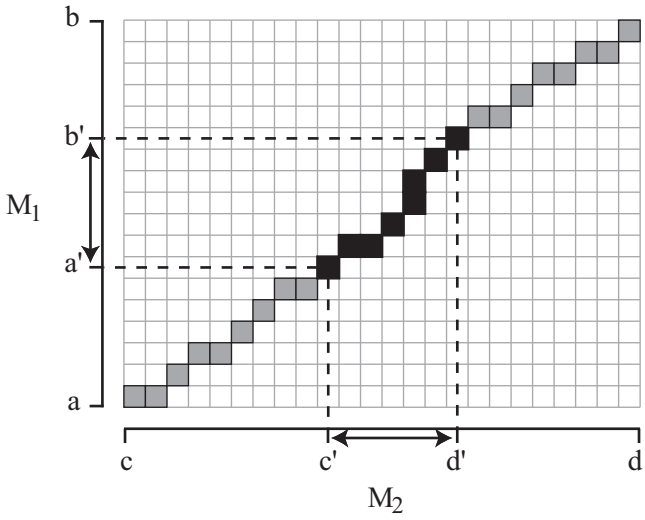

Figure 6.5: Any subregion of an optimal time alignment for $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ is an optimal time alignment for two shorter motion segments $\mathbf{M}_1[a', b']$ and $\mathbf{M}_2[c', d']$.

similar. Otherwise, dynamic programming provides the optimal time alignment between $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$.

Two observations can now be made that allow us to compactly represent *all* potentially similar motion segments. First, if cells (a, c) and (b, d) can be connected with a valid time alignment, then it is likely that nearby pairs $(a \pm \delta, c \pm \delta)$, $(b \pm \delta, d \pm \delta)$ can also be connected. In other words, the boundaries of $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ can be perturbed to find other potentially similar motion segments. It therefore makes sense to identify locally optimal pairs of motion segments where the time alignment has a locally minimal average cell value. Second, any subsection of the optimal time alignment for $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ is itself an optimal time alignment for motion segments inside $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ (Figure 6.5). An optimal time alignment therefore represents an entire family of potentially similar motion segments, not just a single pair.

In light of these observations, our algorithm searches for long paths on the distance grid that correspond to locally optimal time alignments. It starts by looking for chains of 1D minima that satisfy the continuity, monotonicity, and non-degeneracy restrictions. This is done by first locating all minima that have no neighbors to the left, bottom, or bottom-left, since these cannot be in the interior of a chain. Then for each of these minima a chain is formed by iteratively searching the

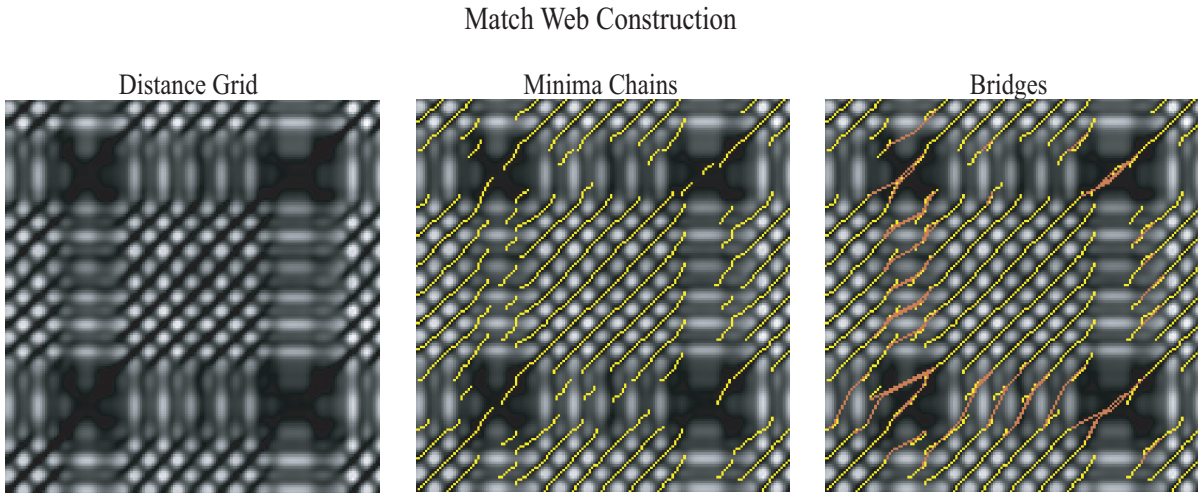

Figure 6.6: A match web is built by computing the distance between every pair of frames, finding chains of local 1D minima (yellow), and adding bridges that connect nearby chains (orange).

top, right, and top-right cells for other minima, making sure along the way that the non-degeneracy condition is not violated. Since some local minima are spurious — for example, walk cycles that are 180 degrees out of phase have local minima at frames where the legs are closest together — some of these chains will not be meaningful. As a heuristic, our algorithm simply removes chains below a threshold length ($0.25s$ in our implementation). Each remaining minima chain is a locally optimal time alignment since moving any cell increases the average distance.

Since the precise location of an individual minimum is somewhat arbitrary (Section 6.2.1), nearby minima chains that ought to be connected may be separate. To be conservative, our algorithm considers connecting any two chains as long as the connecting path is inside the valid region of the grid and has a length (measured via Manhattan distance) less than a threshold L, which was $2s$ in our implementation. For each chain $\mathbf{C}$, we identify other chains $\mathbf{C}'$ in the vicinity. We then compute for each cell C_i on $\mathbf{C}$ the optimal path to each cell on $\mathbf{C}'$ whose Manhattan distance to C_i is less than L. As usual, this path must be within the valid region and obey the other restrictions on time alignments. The path with the smallest average distance is retained as the final connection, or *bridge*, between $\mathbf{C}$ and $\mathbf{C}'$. Note that because bridges must be inside the valid region, nearby chains may not be connectable.

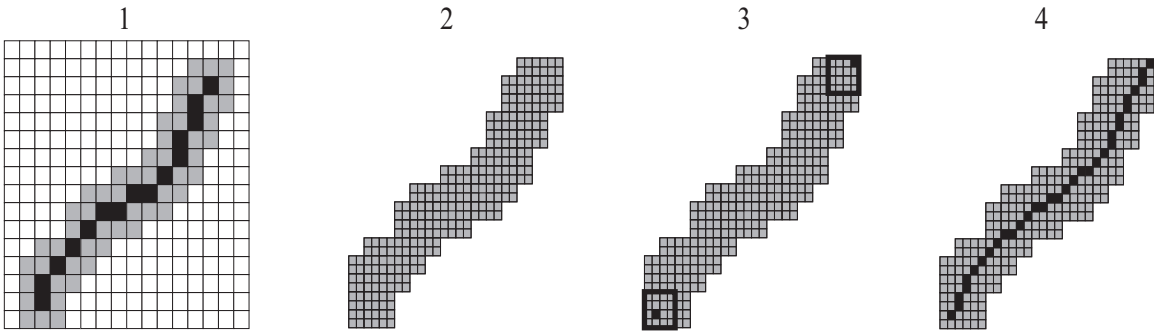

Figure 6.7: To refine a chain, it is first padded (1) and upsampled (2) to form a search region. New endpoints are then found via local search (3) and connected with an optimal path (4).

The result of this procedure is a network of paths on the distance grid, some of them minima chains and others bridges between chains (Figure 6.6). This network represents all potentially similar pairs of motion segments, and we refer to it as a match web. Starting from any cell on any path of the match web, a time alignment can be generated by travelling further down the path and possibly branching off onto connecting paths. While this time alignment is not necessarily optimal, it is close to optimal since every cell is either a local minimum or has a distance value close to a nearby minimum. Also, match webs can be stored compactly since all that must be retained are the grid position and value of each cell on each path.

Match webs can be constructed more efficiently by building them at a low resolution and then refining them at higher resolutions. The lowest resolution match web is built by downsampling $\mathbf{M_1}$ and $\mathbf{M_2}$ and running the algorithm described above. Each minima chain is then refined as shown in Figure 6.7: first, the chain is padded and upsampled to form a search region, then new endpoints are placed at the smallest-valued cells in the vicinity of the old endpoints, and finally these points are joined with an optimal path. Each bridge is handled similarly, except the endpoints are restricted to be on the refined versions of the chains it connects. Figure 6.8 shows an example of refining a low-resolution match web.

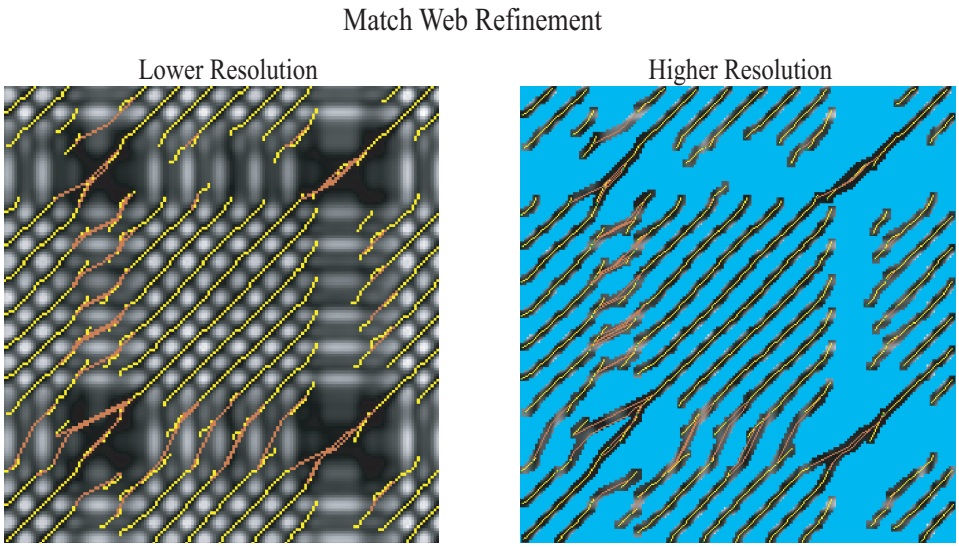

Figure 6.8: A match web may be built at a low resolution (left) and refined at a higher resolution (right). Cells shown in blue are not processed during refinement.

6.2.3 Searching With Match Webs

We now explain how to a use a match web for $\mathbf{M_1}$ and $\mathbf{M_2}$ (where possibly $\mathbf{M_1}$ and $\mathbf{M_2}$ are the same motion) to search for matches. We assume the user has provided a query motion segment $\mathbf{M_1}[a, b]$ and a distance threshold ϵ, and without loss of generality we assume that frames of $\mathbf{M_1}$ correspond to rows of the match web. The case where the query is in $\mathbf{M_2}$ (i.e., it spans columns of the match web rather than rows) is handled similarly. The search algorithm essentially intersects the match web with the rectangle defined by the region from row a to row b, as shown in Figure 6.9. Let a *match sequence* be a sequence of cells formed by selecting any cell of any path on the match web and then travelling down that path, possibly branching off onto connecting paths. The goal is to find all match sequences that span from row a to row b. The algorithm starts by finding every path (minima chain or bridge) that contains a cell in row a. For each such path, the leftmost (i.e., earliest) cell in row a serves as the initial cell of a match sequence. It then checks whether the path contains cells in row b, and if so, a new match sequence is formed by appending all cells up to the last one in row b. The algorithm next considers branching off onto connecting paths to search for additional match sequences. This is done by walking down the path, progressively adding cells

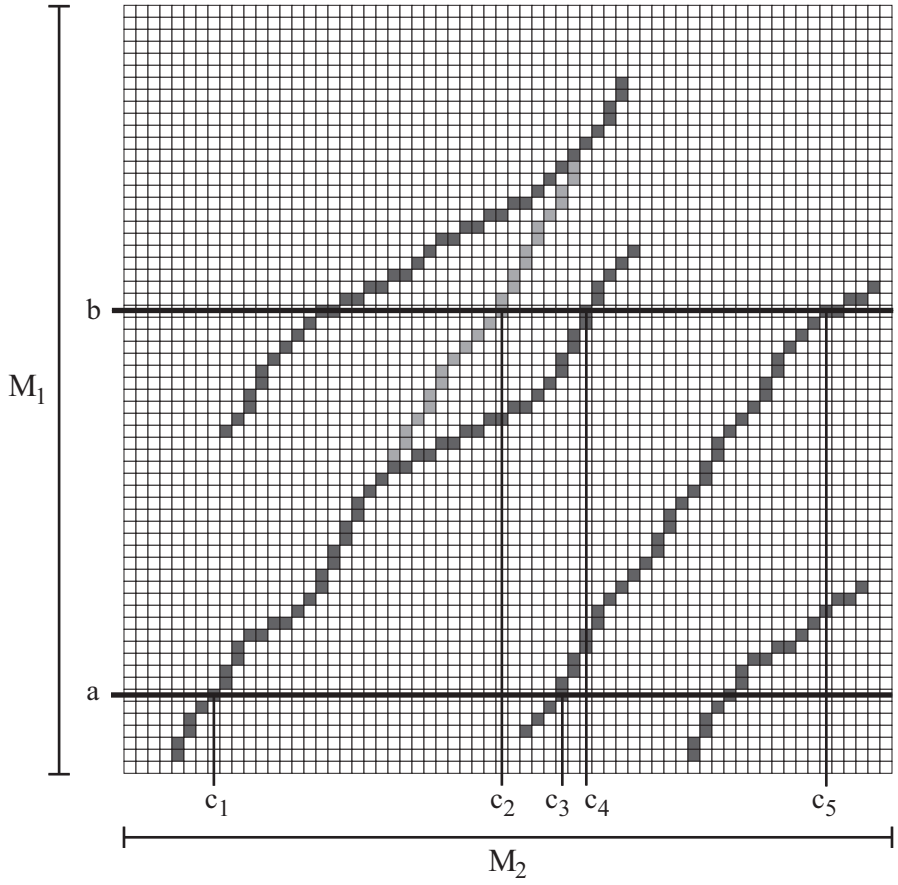

Figure 6.9: A simple search example for the query $\mathbf{M}_1[a, b]$. There are three potential matches: $\mathbf{M}_2[c_1, c_2]$, $\mathbf{M}_2[c_1, c_4]$, and $\mathbf{M}_2[c_3, c_5]$.

to the match sequence and recursively processing each connecting path, until either the end of the path is reached or a cell in row b is encountered.

Each match sequence is a time alignment between the query and a motion segment in $\mathbf{M_2}$ defined by the match sequence's first and last columns. Any match sequence whose average cell value is greater than ϵ is discarded. While each remaining match sequence may be viewed as a match to the query, some of these matches overlap significantly and hence are redundant. In Figure 6.9, for example, this is true of $\mathbf{M_2}[c_1, c_2]$ and $\mathbf{M_2}[c_1, c_4]$. To remove these redundancies, the matches are sorted in order of increasing average distance and placed in an array. The first element of this array is then returned as a match, and every other element that overlaps with it by more than a threshold percentage σ_0 is discarded. This procedure iterates until no matches are left.

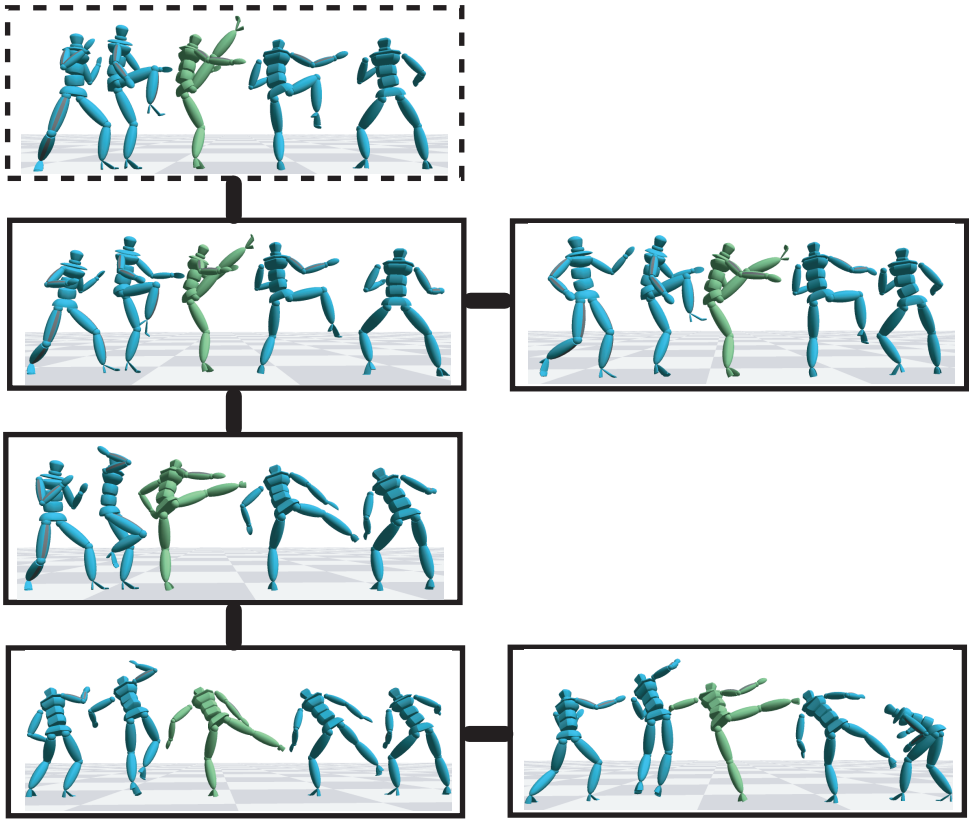

Figure 6.10: An example match graph. The query is in the dotted box and the other nodes are matches. Edges indicate numerical similarity.

The overlap $\sigma(\mathbf{M_2}[a,b], \mathbf{M_2}[c,d])$ of two candidate matches $\mathbf{M_2}[a,b]$ and $\mathbf{M_2}[c,d]$ is defined as

$$\sigma(\mathbf{M_2}[a,b], \mathbf{M_2}[c,d]) = \frac{\|[a,b] \cap [c,d]\|}{\min\left(\|a,b\|, \|c,d\|\right)}, \tag{6.1}$$

where $\|[t_1,t_2]\|$ is the size of the interval $[t_1,t_2]$.

So far we have described how to find the matches that are closest to the query, which we call first-tier matches. To find more distant matches, the search algorithm is repeated for each first-tier match, yielding second-tier matches. This process continues until no new matches are found. In this manner a graph is built where the nodes are motion segments and edges between nodes indicate that the segments are numerically similar. We call this data structure a *match graph* (Figure 6.10). Each edge in the match graph is associated with a time alignment and is assigned a cost equal to the average value of this time alignment's cells. The distance between two nodes is defined as the

cost of the minimal-cost connecting path on the match graph. The match graph can by itself be of interest since it depicts numerical similarity relationships in the matches. To illustrate, Figure 6.10 shows a match graph where the query was a front kick; note the progression from front kick to standing side kick to leaping side kick.

When processing queries other than the initial query, it must be determined whether each "new" match is a heretofore unseen motion or a duplicate of an existing match. When doing this, one must account for the fact that, in practice, duplicates will overlap a great deal but not span identical frame intervals. Let $\mathbf{M}$ be the current query and $\mathbf{M}'$ be a newly identified match. Our system starts by comparing $\mathbf{M}'$ against each node and recording the one with the greatest degree of overlap, $\mathbf{M_{max}}$, where the overlap is defined as in Equation 6.1. If this maximum overlap is less than a tolerance $\overline{\sigma}_0$, then $\mathbf{M}'$ is added to the match graph as a new node along with an edge connecting it to $\mathbf{M}$. If the maximum overlap is greater than σ_0, then the frame interval of $\mathbf{M_{max}}$ is averaged with that of $\mathbf{M}'$ and an edge is added between $\mathbf{M}$ and $\mathbf{M_{max}}$. Otherwise, if the maximum overlap is between $\overline{\sigma}_0$ and σ_0, then $\mathbf{M}'$ is discarded. Intuitively, this is done because an intermediate maximum overlap indicates that $\mathbf{M}'$ is neither a truly new match nor sufficiently similar to any existing match that it can be merged with it, and hence the safest procedure is to eliminate it. In our experience, this situation arises primarily when the distance threshold is sufficiently large that motion segments with only very rough similarity to the query are considered matches.

Appropriate values for σ_0 and $\overline{\sigma}_0$ depend on the query. In general, we have found $\sigma_0 \approx 80\%$ and $\overline{\sigma}_0 \approx 20\%$ to work well, but queries involving multiple periods of a cyclic motion (e.g, walking) require larger values of $\overline{\sigma}_0$ to allow greater overlap in distinct matches.

6.2.4 Experimental Results

We tested our match web implementation on a data set containing 37,000 frames, or a little over 10 minutes of motion sampled at 60Hz. The data was divided into thirty files ranging in length from 3s to 75s, and it included both motions where the actor performed a scripted sequence of specific moves and motions consisting of random variations of the same action. The former class of motions included picking up and putting back objects at predefined locations, walking/jogging in a spiral

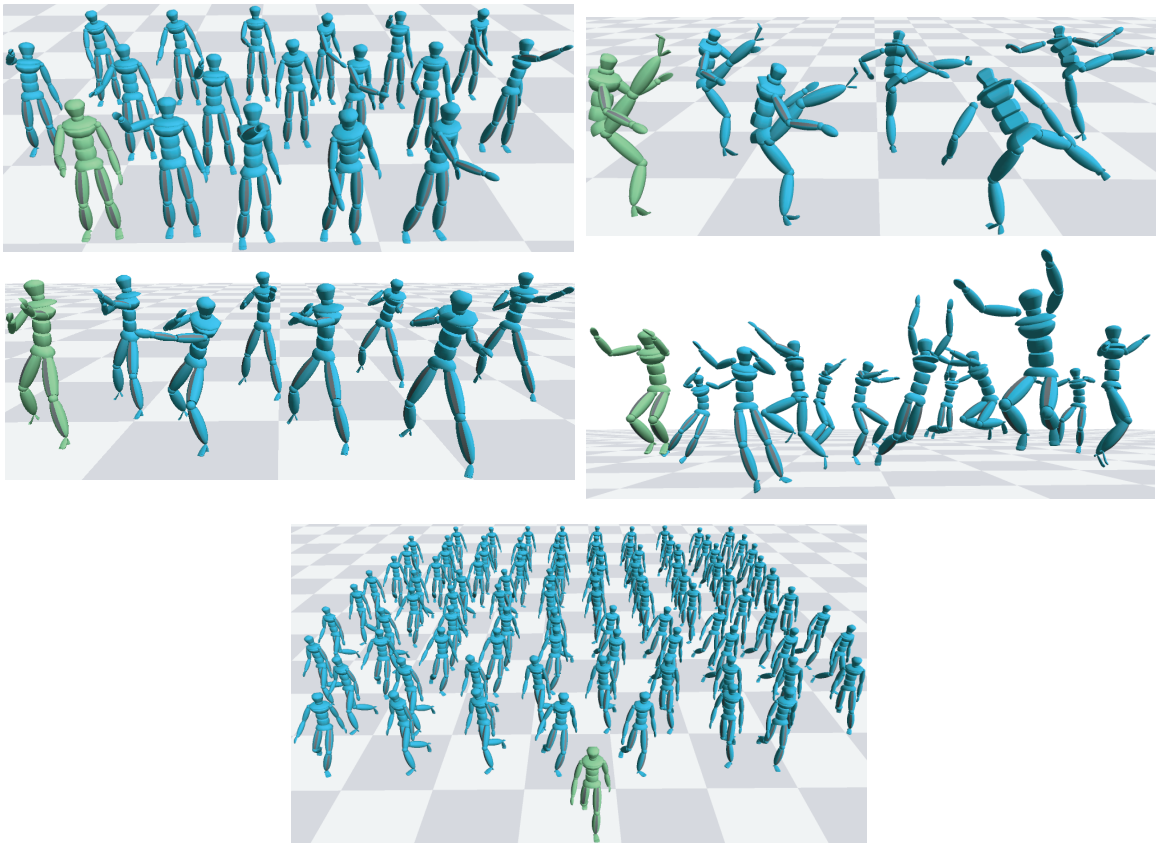

Figure 6.11: Search results for reaching, kicking, punching, jumping, and walking. Query motions are green and matches are blue.

at different speeds, stepping onto/off of platforms of various heights, and sitting down/standing up using chairs of various heights. The latter class of motions consisted of kicks, punches, cartwheels, jumping, and hopping on one foot. All experiments were ran on a machine with 1GB of memory and a 1.3GHz Athlon processor.

Figure 6.11 shows some query motions and the sets of the matches returned by our system. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/ParamMotion. In each case, the entire search process took less than half a second. Manually cropping matches from the data set with similar precision would be quite tedious, especially in the case of the walking query, where 95 matches were identified.

The remainder of this section provides details on the cost of building match webs and executing searches (in terms of both time and storage), presents results on the accuracy of the search, and briefly discusses some advantages and limitations of our approach.

6.2.4.1 Time and Storage Requirements

We initially constructed a match web for the entire data set by building a low resolution version at 10Hz and then refining it in two stages, first to 20Hz and then to 60Hz. The total computation time was 50.2 minutes. Without compression, the size of the match web on disk was 76.2MB. While this is three times the size of the original data, it nonetheless fit comfortably into main memory. Applying a standard compression algorithm (Lempel-Ziv encoding, as implemented in gzip) reduced the size by a factor of three to 24.6MB. Finding matches for a 1.5s query took on average 0.024s. Since each match is used as a new query, the time needed for a full search depends on how many matches are found — for example, finding 100 matches would take about 2.4s.

Using just the names of the data files, we next divided the data set into six categories: cartwheels, fighting, reaching, locomotion, jumping/hopping, and miscellaneous. Most individual data files, however, still contained multiple actions; for example, one "reaching" file contained six reaching motions, many walk cycles, and some motion of the actor readying himself. Separate match webs were built for each category, which took a total of 3.4 minutes and consumed 45MB of disk space without compression. Searching the reaching and locomotion match webs, which each comprised about a quarter of the data, took on average 0.006s for a 1.5s query, or about a quarter of the time needed in the unlabelled case. These results may be interpreted as follows. Building a match web requires $O(n^2)$ time for a data set of n frames, since the distance between every pair of frames must be computed (see Section 6.4 for more discussion on scalability). Hence, dividing the data set into six pieces should reduce computation time by roughly an order of magnitude. On the other hand, motions in different categories tend to be dissimilar and hence have sparse match webs, so dividing up the data set produces a more modest savings in storage than in computation time. The reduced search time stems from our search algorithm being approximately linear in the size of the data set, so smaller data sets yield proportionately faster searches.

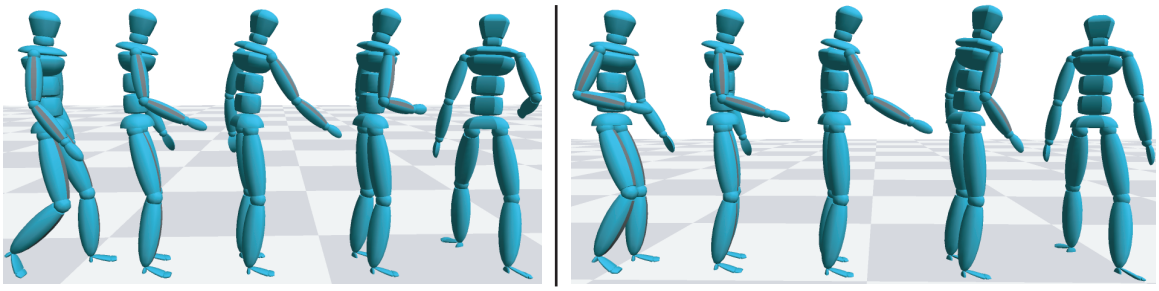

Figure 6.12: Due to their strong numerical similarity, our system confuses picking up an object (left) with putting it back to the same location (right).

6.2.4.2 Accuracy

To test the accuracy of the search results, we first restricted searches to data files in the same semantic category as the query. We entered queries of walking, jogging, jumping, hopping, punching, cartwheeling, sitting down/standing up, stepping onto/off of platforms, kicking, and picking up an object. For each query we attempted to find a distance threshold that would find all logically similar motion segments (identified manually) and nothing else. This was possible for all but the last two queries. For the kick query, we were able to find all kicks involving the same leg except for a spinning back kick. This kick was sufficiently different that it did not have strong time correspondences with any other kick (criterion 2 in Section 6.2.1). For the query of a character picking up an object from a shelf, the system returned every motion involving reaching to the shelf, but in half of the cases the character was putting the object back. A human can distinguish these motions because the reaching arm is initially hanging downwards when picking the object up and bent when putting it back (see Figure 6.12). However, this difference is sufficiently subtle relative to the rest of the motion that our system could not discern it.

We next ran the same experiments using the original, unlabelled data set. The results were identical, with two exceptions. First, more matches were returned for the walking query, since some motions not labelled as locomotion contained short segments of walking. Second, for the reaching query we found that any distance threshold large enough to return all of the data set's 17 reaching motions also included spurious matches. This is because some reaching motions were sufficiently different than the others that, in terms of our numerical similarity metrics, they

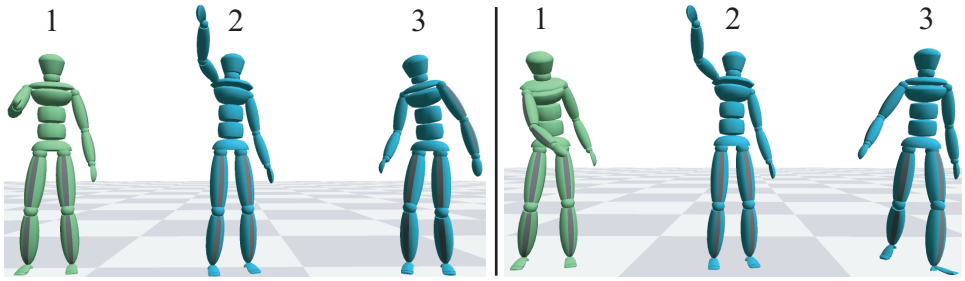

Figure 6.13: Left. Relative to a middle reach (1), a high reach's (2) numerical distance is comparable to looking over one's shoulder (3), but it is closer in terms of graph distance. **Right.** For a query of a lower-left reach (1), any threshold that returns an upper-right reach (2) as a first-tier match also returns spurious matches, such as walking (3).

were comparable to logically unrelated motions such as looking over one's shoulder (Figure 6.13). However, when sorted in order of increasing distance to the query, the first 17 matches were the true reaching motions.

6.2.4.3 Discussion

Our search algorithm defines matches as motion segments that are either close to the query or connected to it via a sequence of close intermediary matches. This allows one to find distant matches while using smaller, more reliable distance thresholds that prune unrelated motion segments. For example, given a query of someone reaching to the lower left, any distance threshold large enough for an upper-right reach to be considered close (i.e., a first-tier match) also produced many spurious matches, such as walking motions (Figure 6.13). On the other hand, using a lower threshold and multi-step search correctly identified the upper-right reach as closer to the query than any other non-reaching motion.

Nonetheless, some matches may be sufficiently far from the others that to find them one *must* use a threshold which also returns spurious matches. Since these spurious matches are used as new queries, they can lead to additional spurious matches. Two possible solutions are to incorporate semantic information, which prunes the space of candidate matches, and to sort the matches based on shortest-path distance to the query, on the assumption that spurious matches have a greater total distance than true matches. In our experiments, both of these solutions were successful.

If one logical match is very far from the others, then it may not be possible to generate clear time correspondences with any other match (that is, the second criterion of Section 6.2.1 would fail). In this case, that match will not be part of the match web and cannot be found by our system. We believe this is reasonable because our ultimate goal is to blend the matches to create parameterized motions, and a match that is this different from the others is unlikely to yield successful blends.

6.3 Building Parameterizations

Once example motions have been collected, they can be interpolated to create new motions using the methods of Chapter 5. While the synthesized motion can be controlled simply by varying the interpolation weights, in general these weights have no simple relationship to motion features. To provide more intuitive control, we parameterize the space of interpolations according to a user-supplied parameterization function $\mathbf{f}$ that computes relevant properties of the initial query motion $\mathbf{M_q}$. For the following discussion, we assume that the inputs to $\mathbf{f}$ are joint positions and orientations. This allows a wide range of properties to serve as the basis for the parameterization, including the location of an end effector at a point in time; the average, minimum, or maximum angular velocity of a joint; and features of aggregate quantities such as the center of mass. We assume that the parameterization function accounts for all meaningful differences between the input motions, and in particular we assume that motions with identical parameters could be used interchangeably. For example, if the examples are reaching motions and $\mathbf{f}$ provides the wrist location at a reach's apex, then it is implicitly assumed that the character performs each individual reach in a similar manner (e.g., it does not sometimes snatch objects and at other times casually pick them up).

Abstractly, given a set of input motions, $\mathbf{f}$ maps interpolation weights $\mathbf{w}$ to a parameter vector $\mathbf{p}$. Our goal is to invert this function: given a set of parameters, we want interpolation weights that produce the corresponding motion. Unfortunately, in general $\mathbf{f^{-1}}$ has no closed form representation. Moreover, since the number of examples is almost always greater than the dimensionality of the parameter space, $\mathbf{f}$ is a many-to-one function, and thus computing $\mathbf{f^{-1}}$ is an ill-posed

problem in the sense that it has no unique solution. In light of this, we use scattered data interpolation to construct a well-defined approximate representation of $\mathbf{f^{-1}}$ from a set of discrete samples $\{(\mathbf{p_1}, \mathbf{w_1}), \ldots, (\mathbf{p_m}, \mathbf{w_m})\}$. If there are n example motions, then initially there are n of these samples: the i^{th} example $\mathbf{M_i}$ is associated with a sample $(\mathbf{f}(\mathbf{M_i}), \delta_{\mathbf{ij}})$, where the i^{th} component of $\delta_{\mathbf{ij}}$ is 1 and the other components are 0. Given a set of target parameters $\widetilde{\mathbf{p}}$, the scattered data interpolation algorithm identifies nearby parameter samples $\{\mathbf{p_{i_1}} \ldots \mathbf{p_{i_k}}\}$ and averages the weights associated with each $\mathbf{p_{i_j}}$ according to the distance between $\mathbf{p_{i_j}}$ and $\widetilde{\mathbf{p}}$. Note that this makes $\mathbf{f^{-1}}$ well-posed by restricting the space of possible weights based on the samples $(\mathbf{p_i}, \mathbf{w_i})$.

The approximation of $\mathbf{f^{-1}}$ becomes better as the parameter space is sampled more densely, in the sense that the generated interpolation weights will produce motions that more accurately possess the desired parameters. Indeed, in the limit of arbitrarily many samples we effectively have a lookup table that directly maps motion parameters to interpolation weights. Uniform sampling also leads to better parameterizations, because clusters of samples can skew the approximation. In the extreme case where one set of sampled parameters $\mathbf{p_i}$ is duplicated several times, it will influence the final set of weights as if it were much closer to the target parameters $\widetilde{\mathbf{p}}$ than it truly is. Unfortunately, the sampling of parameter space provided by the example motions is not guaranteed to be either dense or uniform, and hence the approximation of $\mathbf{f^{-1}}$ may be inaccurate (Figure 6.2). To correct this, we synthesize interpolations of the examples to create additional samples of $\mathbf{f^{-1}}$, ensuring that the parameter space is sampled densely and uniformly.

The remainder of this section provides details on building parameterizations. The following issues are covered:

1. **Motion registration.** The methods of Chapter 5 are generalized so registration curves can be built from match graphs.

2. **Sampling strategy.** To sample interpolation weights when there are many example motions, our algorithm finds subsets of these examples that are nearby in parameter space. Weights are then randomly generated for this subset in a manner that explicitly limits the degree to which the data can be extrapolated.

3. **Fast interpolation that preserves constraints.** We use a k-nearest-neighbors technique that is efficient for large example sets and respects constraints on interpolation weights.

After discussing these matters, we conclude with some example results.

6.3.1 Registration

A registration curve is built for the example motions in a manner similar to what was used in Chapter 5: using timewarp and alignment curves between pairs of motions, "global" timewarp and alignment curves are built that encompass all of the motions. The procedure for determining constraint matches is then the same as before. The algorithm starts by using Dijkstra's algorithm to identify the shortest path from the query $\mathbf{M_q}$ to every other motion in the match graph. Any edge that is not on one of these shortest paths is discarded. For each remaining edge, the methods of Chapter 5 are used to build timewarp and alignment curves between the motions at the incident nodes. The timewarp curves are constructed so the initial and final frames of both motions are exactly interpolated. Now, given a frame of $\mathbf{M_q}$, the corresponding frame in any other motion can be found by walking down the shortest path on the match graph, using the timewarp curve at each edge to convert the frame of the current motion to the corresponding frame of the next motion. In this manner for any frame $\mathbf{M}_q(t_q)$ one can generate a frame correspondence $(\mathbf{M_q}(t_q), \mathbf{M_2}(t_2), \ldots, \mathbf{M_n}(t_n))$, where without loss of generality we have identified $\mathbf{M_q}$ with $\mathbf{M_1}$. To generate a timewarp curve for the entire match graph, frames of $\mathbf{M_q}$ are sampled to generate a dense set of these frame correspondences, and an n-dimensional, strictly increasing, endpoint interpolating quadratic B-spline is fit to them as in Chapter 5. The procedure for building the alignment curve is analogous: for any frame of $\mathbf{M_q}$, a transformation that aligns it with the corresponding frame of $\mathbf{M_i}$ can be found from the match graph, and so sets of these transformations are sampled and fit with a B-spline.

If any part of the parameterization function involves the skeletal configuration on a specific frame $\mathbf{M_q}(t_0)$, then this frame index t_0 is converted to the corresponding index u_0 on the timewarp curve. This allows the parameterization function to be computed for any other example motion and for any interpolation of the examples.

6.3.2 Sampling

While the goal is to produce a dense sampling of parameter space, parameter samples cannot be created directly; instead, they must be generated indirectly by sampling interpolation weights. Densely sampling these weights is infeasible for large example sets, because the dimensionality of the interpolation weight space is proportional to the number of examples, and so the number of samples needed to achieve a given sampling density grows exponentially with the number of examples. Moreover, such a sampling would be redundant because $\mathbf{f}$ is a many-to-one function — the same region of parameter space would be covered repeatedly by different sets of interpolation weights.

These difficulties can be avoided by limiting interpolations to subsets of examples that are nearby in parameter space. Intuitively, the goal of sampling is to fill in the gaps in parameter space, and the most natural strategy for filling any given gap is to combine the closest example motions. For example, imagine the examples are various reaching motions and we want to sample interpolated motions that reach near chest height. This could theoretically be done by combining motions that reach to the ground with ones that involve standing on tiptoes, but it is more sensible to instead combine example motions that already reach to points near the desired region.

This intuition can be turned into an algorithm as follows. First, the parameters of each example motion are computed. To approximate the accessible region of parameter space, a bounding box is generated for these parameters and expanded in each dimension by a fixed percentage about the central value (20% in our implementation). Points within this region are then randomly sampled, and for each point the algorithm locates the $d + 1$ example motions with the closest parameters, where d is the dimensionality of the parameter space. This number of neighbors is used because it is the minimum necessary to form a volume in parameter space. The weight of every other motion is set to zero, and a random set of weights is generated for the neighbors under the restriction that these weights must be nearly convex, or *valid*. Mathematically, this is defined as

$$-\delta \leq w_i \leq 1 + \delta \tag{6.2}$$

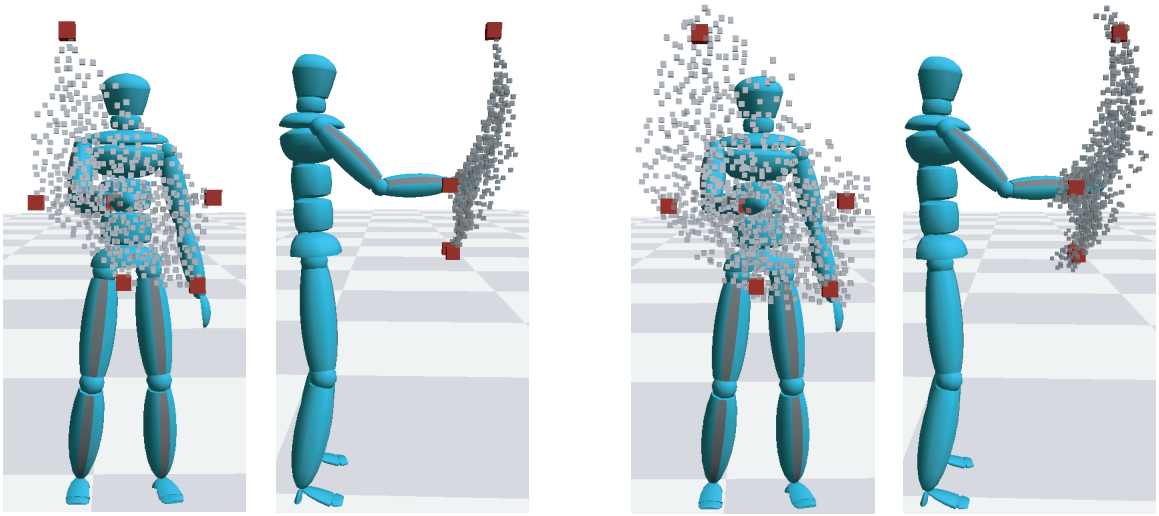

Figure 6.14: Samplings of parameter space for a set of reaching motions, using different values of δ.

and

$$\sum_i w_i = 1, \tag{6.3}$$

where δ controls the allowable degree of extrapolation. In particular, when $\delta = 0$ there is no extrapolation in the sense that the interpolation weights are strictly convex. A random set of valid interpolation weights can be calculated as follows. Let S be the sum of all weights that have been assigned values; initially $S = 0$. While unassigned weights exists, randomly select one of these weights w_i. If w_i is the last unassigned weight, set it to $1 - S$. Otherwise randomly assign it a value from the interval $[\max(-\delta, -\delta - S), \min(1 + \delta, 1 + \delta - S)]$. This ensures that both w_i and $S + w_i$ have a value in the range $[-\delta, 1 + \delta]$. The latter condition is important because it guarantees that the final weight is valid.

To prevent parameter samples from being too close, for each new sample the closest existing sample is found. If these samples are within a threshold distance of each other, then the new sample is discarded. In our experiments, which focused on joint positions, the threshold was half an inch.

By selecting different values for δ, a user can control the tradeoff between flexible synthesis (motions are generated from a larger space of possible interpolation weights) and quality guarantees (synthesized motions stay "near" the original data). Figure 6.14 shows samplings of parameter

space generated with different values for δ. In general, we have found that δ can be set to .1–.15 without much sacrifice of motion quality.

6.3.3 Interpolation

Given a new set of parameters $\widetilde{\mathbf{p}}$, a k-nearest-neighbors algorithm is used to find interpolation weights $\widetilde{\mathbf{w}}$ that produce those parameters. Let the k nearest neighbors be $\mathbf{p_1}, \ldots, \mathbf{p_k}$, in order of increasing distance, and let $\mathbf{w_i}$ be the interpolation weights associated with $\mathbf{p_i}$. $\widetilde{\mathbf{w}}$ is approximated as

$$\widetilde{\mathbf{w}} = \sum_{i=1}^{k} \alpha_i \mathbf{w_i}. \tag{6.4}$$

Following Allen et al [3], each α_i is initially assigned the value

$$\alpha_i = \frac{1}{D(\widetilde{\mathbf{p}}, \mathbf{p_i})} - \frac{1}{D(\widetilde{\mathbf{p}}, \mathbf{p_k})}, \tag{6.5}$$

where D computes the distance between two parameters (Euclidean distance in our implementation). These weights are then normalized so they sum to 1. Since the α_i are nonnegative and sum to 1, $\widetilde{\mathbf{w}}$ is inside the convex hull of the $\mathbf{w_i}$, and therefore $\widetilde{\mathbf{w}}$ satisfies the conditions in Equations 6.2 and 6.3. This ensures that any parameters specified by the user will produce a motion within the space of valid interpolations. In particular, parameters that are not attainable are effectively projected onto the accessible region of parameter space.

As a result of our sampling procedure, at most $d + 1$ elements of each $\mathbf{w_i}$ are nonzero. This implies that $\widetilde{\mathbf{w}}$ will in the worst case have $k(d+1)$ nonzero weights, and in practice there are fewer because nearby parameter samples tend to have the same set of nonzero weights. Since motions with zero weight can be ignored when creating a blend, the asymptotic run time of our algorithm is independent of the number of examples. This analysis neglects the cost of finding the k nearest neighbors, but we have found that even a brute force nearest neighbor calculation is negligible relative to the cost of interpolating motions.

Motion	# Examples	Time To Build	Size of Samples
reach	6	6.7s	4.5%
walk	96	4.6s	2.9%
kick	4	1.8s	1.4%
sit	2	3.9s	0.4%
step up	4	2.5s	0.4%
punch	7	1.7s	4.9%
hop	4	3.3s	1.8%
cartwheel	11	6.5s	8.2%

Table 6.1: Data about the parameterized motions built in our experiments. The storage needed for parameter samples is given as a percentage of the motion data's size.

6.3.4 Results and Applications

We have implemented the above algorithms and used them to create a variety of parameterized motions; see Table 6.1 for a summary. In each case we generated a thousand parameter samples using the method described in Section 6.3.2. The time needed to generate these samples varied from 1.7s to 6.7s, and after eliminating redundant samples the storage cost was 8.2% of the example motion data in the worst case and 3.2% on average. In all of our experiments new motions could be synthesized in real time, and these motions matched the user's target parameters to within visual tolerance as long as they were within the accessible region of parameter space. We set $k = 12$ when performing nearest neighbor interpolation. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/ParamMotion.

In addition to providing more accurate parameterizations, the sampled parameters can be used to visualize the range of synthesizable motions. Figure 6.15 shows some cases where this can be accomplished simply by drawing markers at the location of each parameter sample. Another application of our methods is automatic removal of redundant example motions, which can reduce the parameterized motion's memory footprint. For each example motion, we compute its parameters and see if they can be reproduced within a user-specified tolerance by interpolating nearby

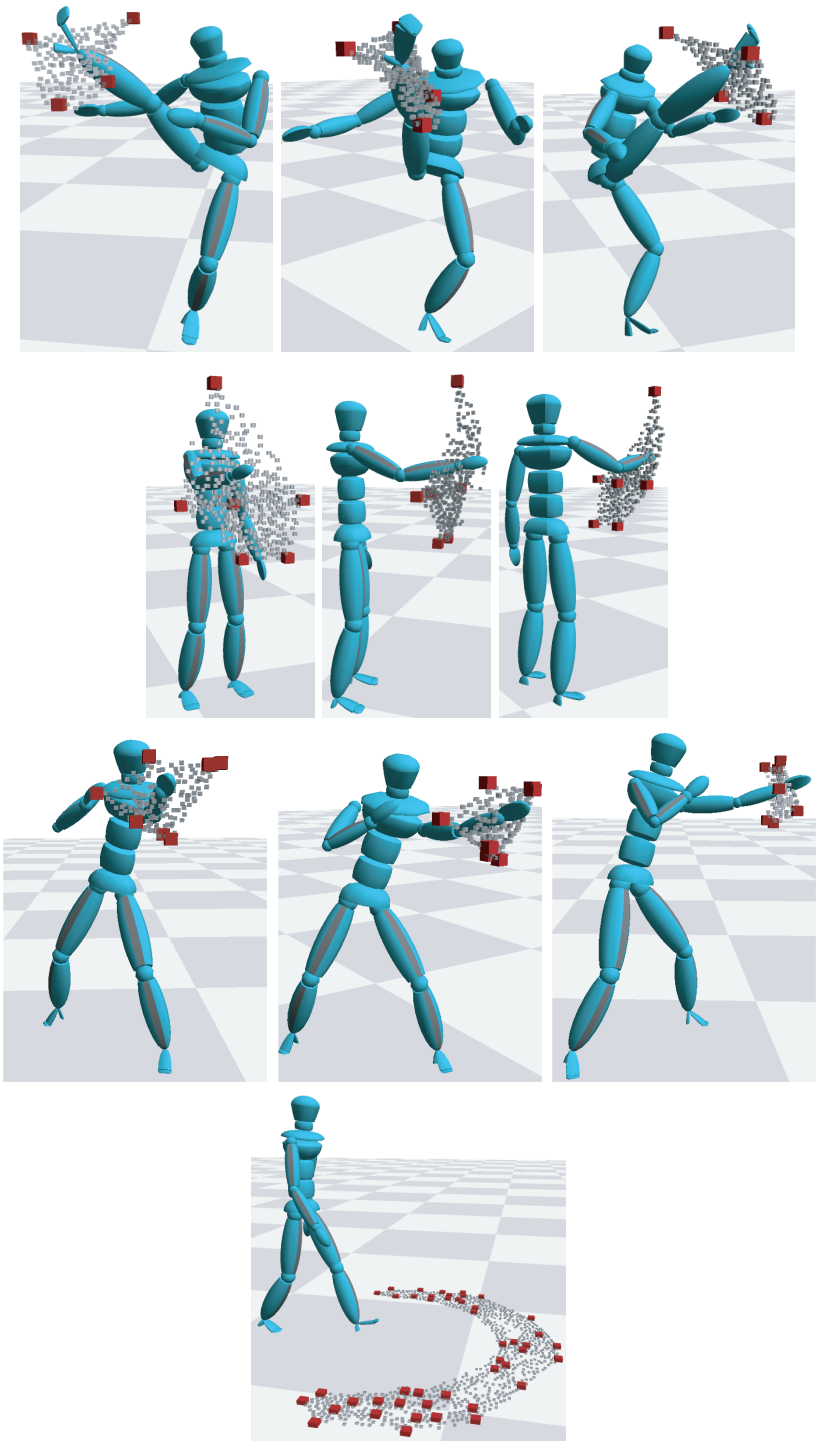

Figure 6.15: In order from top to bottom: visualization of the accessible locations for the ankle of a kick, the wrist of a reach, the wrist of a punch, and the final position of a walk cycle. Large red cubes show parameters of example motions; small grey cubes are sampled parameters.

examples. If so, it is discarded. For our parameterized walk, we removed all example motions where the final root location could be reproduced to within a quarter inch, reducing the number of motions from 96 to 46. The parameterized walk also shows the scalability of our scattered data interpolation method: including all 46 example motions in motion interpolation takes an order of magnitude longer than using our algorithm.

The automation provided by our system makes it feasible to experiment with unusual parameterized motions. Starting with 34s of cartwheel data, we built a parameterized motion where the user could control the final position of a sequence of cartwheels. The total amount of time needed to build the match web, identify a particular cartwheel, execute a search for other cartwheels, and generate parameter samples was less than thirty seconds.

6.4 Discussion

This chapter has presented automated methods for extracting logically related motions from a data set and converting them into an intuitively parameterized space of motions. One contribution of this work is a novel search method that uses numerically similar matches as intermediaries to find more distant matches, together with a precomputed representation of all possibly similar motion segments that makes this approach efficient. A second contribution is an automatic procedure for parameterizing a space of interpolations according to user-specified motion features. This algorithm samples interpolations to build an accurate approximation of the map from motion parameters to interpolation weights, and it incorporates a scalable scattered data interpolation method that ensures interpolation weights have reasonable values.

We conclude with a brief discussion of the scalability and generality of our methods.

6.4.1 Scalability

For our test data set, which is large relative to what is commonly used in current research, the time and space costs for building and storing a match web were quite manageable. However, a match web for a data set of n frames takes $O(n^2)$ time to construct and $O(n^2)$ space to store. The

time costs can be mitigated by using a multi-resolution construction method (as discussed in Section 6.2.2) and by computing match webs for different pairs of motions in parallel, and storage can be reduced through standard compression algorithms. Nonetheless, for massive databases it will not be feasible to build match webs for every pair of motions. A simple solution is to partition the database into independent modules based on semantic content and compute match webs separately for these modules. Since a very large database will contain many unrelated motions, such a division would be natural and is likely to already be reflected in the organization of the data files. The development of alternatives to match webs that are asymptotically more efficient yet have similar performance characteristics is left for future work.

Because our search algorithm tests every path in the match web that is contained within the rows spanned by the query, in general the time needed to execute a search is linear in the size of the database. However, we believe the paths in the match web may be organized into a hierarchical data structure that allows more efficient pruning of paths that exceed the distance threshold. Developing such a data structure is also left for future work.

While our parameterization algorithms apply in theory to parameter spaces of arbitrary dimensionality, in practice they are limited by the quantity of available data. Roughly speaking, we need at least enough examples to cover every combination of minimum/maximum values for individual parameters (the "corners" of the space), and more examples typically provide higher quality results. The number of necessary example motions is hence exponential in the number of parameters. This is a fundamental limitation of scattered data interpolation. In future work we intend to explore methods for easing the data requirements by identifying motion properties that are, for the purposes of sampling, uncorrelated.

6.4.2 Generality

While we have focused on parameterization functions involving joint positions and orientations, our methods allow parameterizations based on abstract properties like mood. For qualitative features like these where accuracy is less meaningful, we can simply skip the sampling step of Section 6.3.2 and apply scattered data interpolation directly to the example motions.

Motion sets found by our search engine are not guaranteed to be blendable. For example, one of our query motions consisted of a character stepping toward a shelf, picking up an object, and walking away. While our system correctly identified the eight other picking-up actions in the database, in three of these the initial step was with the wrong foot, and so these motions had to be discarded. More generally, the only reliable way of determining whether motions can be successfully blended is to create and look at specific blends. In light of this, one of the primary advantages of our system is that it greatly speeds and simplifies the process of experimentation.

Chapter 7

Discussion

Realistic human motion is difficult to animate — not only is the motion itself intrinsically complicated, but human observers are quite adept at discerning even subtle flaws. *Controllably* generating realistic motion is even more challenging, because one needs an entire repertoire of motion to draw from, along with control mechanisms for creating motions with desired properties. Data-driven synthesis is a powerful method for solving both of these problems: a discrete set of high-quality examples is converted into a generative model that can produce a wide range of new motions, and a high-level interface makes it possible to generate motions with specific properties. However, data-driven synthesis can only reliably produce motion that is similar to the original examples, and expressive motion models therefore require large data sets. While motion capture provides a practical means of obtaining large collections of examples, previous work has been limited to manual techniques that become awkward and laborious when applied to large data sets. This dissertation has introduced automated methods for building motion models, greatly reducing the time and labor needed to convert a finite set of examples into a richer object that can controllably synthesize new, realistic motion. We have focused on methods for building and using two popular models that offer complementary forms of control: motion graphs, which control sequences of actions, and motion blends, which control individual actions. Four primary contributions have been made:

1. **Efficient and accurate enforcement of kinematic end effector constraints.** Chapter 3 presented a new algorithm for precisely enforcing kinematic end effector constraints without introducing discontinuities into the original motion, regardless of how many constraints there are or when they turn on and off. Combined with methods for inferring constraints in synthesized motion from the original data, this algorithm is a crucial component of our graph-based and blending-based synthesis models, as it allows important interactions between the character and the environment to be automatically preserved.

2. **Automated construction and use of motion graphs.** Chapter 4 showed how motion graphs can be automatically constructed by identifying places where motions are similar and synthesizing special transition motions at the points. Search algorithms were then used to provide a high-level interface for extracting particular motions from the resulting graph, and this technique was demonstrated on the important problem of directing very general kinds of motion down arbitrary paths.

3. **Automated motion blending.** Chapter 5 introduced registration curves, a data structure that summarizes relationships between the timing, local coordinate frame, and constraint state of arbitrarily many motions. Registration curves make it possible to blend motions with no user intervention beyond specification of the blend weights themselves, providing an automated alternative for systems that use manual blending methods to create transitions, interpolations, and continuously controllable motion.

4. **Automated construction of parameterized motions.** Building off of the technology in Chapter 5, Chapter 6 provided automated techniques for building parameterized motions, which are an important application of motion blending. First, a search algorithm was introduced for automatically extracting from a data set motion segments that are similar to a query, greatly simplifying the task of collecting a set of related example motions. Second, a new method was provided for parameterizing the space of interpolations; this method is more efficient and scalable than those used previously and does not require the example motions to span an *a priori* range of variation.

We have discussed motion graphs and motion blends separately, but they could be used together for additional flexibility. For example, motion blending could be used to create a sampling of different variations of a motion in order to supply a motion graph with additional source material. A clear direction for future work is the development of tools for automatically constructing hybrid graph/blending synthesis models, such as a graph where each edge is a parameterized motion rather than a static motion clip. Indeed, this particular model has been investigated previously in the Verbs and Adverbs system of Rose et al. [76]. In this work each parameterized motion was constructed manually (example motions were manually cropped and time-aligned for blending, and parameterizations relied on a priori knowledge of the accessible range of variation), and the graph structure itself was also directly defined by the user (linear-blend transitions were added at specified frames). We believe the methods of this dissertation could be extended to considerably automate this process. Moreover, we believe more general models could be developed where, for example, the set of allowable transitions depends not just on what kind action has been taken but also on what the properties (parameters) of that action were.

The remainder of this chapter discusses applications of our models, limitations of our methods, and avenues for future work.

7.1 Applications

Our motion models are appropriate for a variety of applications. For movie special effects, blending-based synthesis could be used to adjust the motion of human or human-like characters (e.g., aliens or monsters) so they better interact with the environment or with other actors. For crowd animation, graph-based synthesis could be used to generate motions that follow predefined trajectories constructed so as to avoid inter-character collisions. For games, graph-based synthesis could be used for general animation of non-player characters (where some control delay would be better tolerated), and blending-based synthesis could be used to adjust on the fly actions that depend upon unpredictable circumstances (e.g., picking up an object that could be dropped anywhere within a region). More generally, both graph-based and blending-based synthesis are appropriate

for rapid construction of sets of motions for movies, commercials, scripted training simulations, or any other form of visualization.

One application for which our models need further development is the online animation of user-controlled characters, such as in a game. Not only does this application require motion to be continually generated and adaptable to changing circumstances, but this motion must also be highly responsive to user control. Supplying this responsiveness is challenging because one must be able to switch between actions on a very short timescale (a quarter of a second or less is typical for a game). This requires considerably more careful organization of the available motions than the methods used in Chapter 4, as one needs to guarantee that a large number of actions are possible at effectively any point in time (see Gleicher et al. [33] for some initial work in this direction). Moreover, motion quality is more difficult to preserve in highly responsive models. Real people cannot instantly change their motion, and momentum and balance constraints prevent people from arbitrarily switching between different actions. At the very least, this means that mechanisms must exist for selecting a tradeoff between realism and responsiveness. A preferable (but possibly more challenging) approach is to develop tools for specially designing motions such that rapid transitions can be made realistically. This would likely require example motions to be captured more carefully, and may even require a rethinking of how such capture can best be carried out.

Our technical presentation has implicitly assumed that the acquisition of a data set is independent of model construction, but our techniques are sufficiently fast that they could be used in conjunction with a live motion capture session. A performer's motion could be recorded for a few minutes, and within a comparable amount of time one could already be resequencing motion with a motion graph, experimenting with different blends, or assessing the range of variation provided by a parameterized motion. Based on this, one could then decide whether existing motions need to be performed differently or whether new motions need to be captured. This quick feedback loop would simplify the process of creating motion models tuned to the requirements of a specific application.

7.2 Limits to Automation

While our methods are automated, they are not fully automatic. In general, we rely on user intervention whenever the meaning of a motion is important. For example, if a user wants to restrict the style of motion generated from a motion graph, then it is up to them to add the appropriate descriptive labels to the original data. Similarly, the user must tell our system what aspect of a motion is the appropriate basis for a parameterization; it cannot, for instance, automatically infer that the heel position at full extension is the most important part of a kick. We believe that this sort of user intervention is ultimately unavoidable, since computationally achieving high-level understanding of something as subtle as human motion is likely to remain an unsolved problem for the foreseeable future. However, while user intervention probably cannot be eliminated, it likely can be reduced, possibly through the application of machine learning algorithms that extrapolate from information that has already been supplied by the user [6].

A second place where we rely on user intervention is when determining how much a motion model can modify the original data. Allowing larger changes gives the model more freedom at the expense of placing looser guarantees on motion quality. This tradeoff is difficult to balance automatically because the levels of fidelity and control needed in a motion model depend upon the intended application. At the same time, our mechanisms for selecting this tradeoff are relatively simple. For motion graphs, the tradeoff is controlled by the distance thresholds used when deciding where to place transitions. For parameterized motions, it is controlled by the search threshold (which effectively determines how different the closest examples can be) and by the amount of extrapolation allowed during blending. In the data sets used in this dissertation (which are large by current standards), once these values were set the time needed to complete model construction was on the order of tens of seconds, providing quick feedback for users attempting to adjust these values.

7.3 Limits to Data Acquisition

An important limitation of our models is that they may require prohibitive amounts of input data when motions have a large number of fundamental degrees of freedom that must all be controlled. For example, the path synthesis problem addressed in Chapter 4 is effectively two dimensional, since for control purposes the only relevant property of a clip is the change in position in the floor plane. However, one might want to be able to control many different features of a motion. For instance, for walking one might want to control the speed, curvature, and step length of the walk cycle; the mood and gender of the character; and logically independent actions of the upper body like carrying objects of different sizes and weights. If each of these attributes are to be independently controlled, then it is necessary to have enough data that one property can be varied while everything else is held constant. With our current methods, this necessitates a combinatorial explosion in the amount of input data, since one must effectively have a captured motion for every possible combination of motion features. We believe that this problem can be mitigated by developing mechanisms that identify "independent" features of captured motion and layer them into a composite motion.

We have focused on reducing the amount of labor needed to build motion models from a motion capture data set, but we have not addressed the labor that goes into acquiring the data set itself. Current methods for cleaning raw marker data and fitting it onto a skeleton can still require a significant amount of user intervention, and while there are no published accounts of exactly how much labor is typically needed, anecdotal evidence suggests that a minute of raw motion data can take hours or even days to fully process. Except for some early work on marker-to-skeleton data conversion [11, 67], this problem has largely been ignored by the research community. In the short term, our methods could always be applied to imperfectly cleaned data to give a rough sense of what sorts of new motion can be synthesized, and some our methods (such as identifying transition locations or sets of similar motion segments) can be applied directly to marker locations. As large data sets become increasingly common, however, it will become necessary to drastically reduce the time needed to process marker data into high-quality skeletal motion.

7.4 Incorporating Other Techniques

Our models do not guarantee that synthesized motions are physically accurate, which may result in characters that appear out of balance or that perform motions which are beyond the strength limits of real humans. Methods exist for adjusting motions to better satisfy physical laws [47, 86], and any one of these could be applied to our synthesized motions as a postprocess, much as we currently apply the algorithm of Chapter 3 to enforce kinematic end effector constraints. At present, however, methods for enforcing physical constraints are not sufficiently reliable to be used in an automatic setting. Moreover, the true physical constraints on synthesized motion are unknown. For example, the mass distribution of an actor is typically not acquired at capture time, and it is unclear how the physical accuracy of a motion is affected when it is fit onto a rigid skeleton. Incorporation of physics constraints into data-driven synthesis hence remains an important area for future work.

In general the proportions of an animated character will not match those of the live performer, especially if multiple actors are used for a single data set. Existing retargeting methods [31, 54] use optimization algorithms to adapt captured skeletal motions to different bodies while preserving contact constraints and avoiding self-intersection. These methods could trivially be applied to the captured motions prior to model construction, so as to tailor the models to the character(s) of interest.

7.5 Generalizing Current Models

This dissertation has used synthesis models that are simplified in several ways. One important simplification is that we only treat full-body motion, ignoring finer structures like the face and hands. With current technology, the motion of these body parts is typically captured independently of the rest of the body. In real life, however, facial and hand motion are not truly independent of full-body motion. For example, a character's gestures are typically tightly correlated with facial expression and hand configuration. In future work we plan to develop models that can synthesize facial and hand motion along with full-body motion while preserving relationships between these different types of movement.

An additional simplification used in this dissertation is that every degree of freedom in the body is synthesized simultaneously; there is no logical decoupling between, for example, the upper and lower bodies. To illustrate, imagine that we have a motion of someone walking and a motion of someone waving while standing still. It seems natural to transfer the wave onto the walk so they occur in concert, and yet the models developed in this thesis provide no mechanism for accomplishing this. Executing this style of motion combination is nontrivial, and in particular simply attaching the upper body of one motion onto the lower body of another will not yield a realistic result, because subtle but important correlations between the movement of different body parts are ignored. Providing support for combining motions of different body parts onto a single body is also left for future work.

LIST OF REFERENCES

[1] Y. Abe, C. K. Liu, and Z. Popović. Momentum-based parameterization of dynamic character motion. *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*, August 2004.

[2] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organizations and Algorithms (FODO)*, pages 69–84. Springer Verlag, 1993.

[3] B. Allen, B. Curless, and Z. Popović. Articulated body deformation from range scan data. *ACM Transactions on Graphics*, 21(3):612–619, 2002.

[4] K. Amaya, A. Bruderlin, and T. Calvert. Emotion from motion. In *Proceedings of Graphics Interface (GI'96)*, pages 222–229, May 1996.

[5] O. Arikan and D. A. Forsyth. Interactive motion generation from examples. *ACM Transactions on Graphics*, 21(3):483–490, 2002.

[6] O. Arikan, D. A. Forsyth, and J. O'Brien. Motion synthesis from annotations. *ACM Transactions on Graphics*, 22(3):402–408, 2003.

[7] G. Ashraf and K. C. Wong. Generating consistent motion transition via decoupled framespace interpolation. *Computer Graphics Forum*, 19(3), August 2000.

[8] G. Ashraf and K. C. Wong. Constrained framespace interpolation. In *Computer Animation 2001*, pages 61–72, August 2001.

[9] J. Barbic, A. Safonova, J.-Y. Pan, C. Faloutsos, J. Hodgins, and N. Pollard. Segmenting motion capture data into distinct behaviors. In *Proceedings of Graphics Interface (GI 2004)*, 1994.

[10] R. Bindiganavale and N. Badler. Motion abstraction and mapping with spatial constraints. In *Modelling and Motion Capture Techniques for Virtual Environments, CAPTECH'98*, pages 70–82, November 1998.

[11] B. Bodenheimer, C. Rose, S. Rosenthal, and J. Pella. The process of motion capture: dealing with the data. In *Computer Animation and Simulation '97*, pages 3–18, 1997.

[12] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[13] R. Bowden. Learning statistical models of human motion. In *IEEE Workshop on Human Modelling, Analysis, and Synthesis, CVPR 2000*. IEEE Computer Society, June 2000.

[14] M. Brand and A. Hertzmann. Style machines. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 183–192. ACM SIGGRAPH, July 2000.

[15] A. Bruderlin and L. Williams. Motion signal processing. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 97–104. ACM SIGGRAPH, August 1995.

[16] M. Cardle, M. Vlachos, S. Brooks, E. Keogh, and D. Gunopulos. Fast motion capture matching with replicated motion editing. In *Proceedings of SIGGRAPH 2003 Technical Sketches & Applications*, 2003.

[17] V. Castelli and L. Bergman. *Image Databases: Search and Retrieval of Digital Imagery*. John Wiley & Sons, 2001.

[18] K. Chan and W. Fu. Efficient time series matching by wavelets. In *Proceedings of the 15th IEEE International Conference on Data Engineering*, pages 126–133, 1999.

[19] D. Chi, M. Costa, L. Zhao, and N. Badler. The emote model for effort and shape. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 173–182. ACM SIGGRAPH, August 2000.

[20] K.-J. Choi and H.-S. Ko. On-line motion retargeting. *Journal of Visualization and Computer Animation*, 11:223–243, 2000.

[21] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2000.

[22] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, pages 419–429, 1994.

[23] P. Faloutsos, M. van de Panne, and D. Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH 2001*, Annual Conference Series, pages 251–260. ACM SIGGRAPH, July 2001.

[24] P. Faloutsos, M. van de Panne, and D. Terzopoulos. The virtual stuntman: dynamic characters with a repetoire of autonomous motor skills. *Computers and Graphics*, 25(6):933–953, 2001.

[25] A. Fang and N. Pollard. Efficient synthesis of physically valid human motion. *ACM Transactions on Graphics*, 22(3):417–426, 2003.

[26] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1996.

[27] T. Funkhouser, P. Min, M. Kazhdan, J. Chen, A. Halderman, and D. Dobkin. A search engine for 3d models. *ACM Transactions on Graphics*, 22(1):83–105, 2003.

[28] A. Galata, N. Jognson, and D. Hogg. Learning variable-length markov models of behavior. *Computer Vision and Image Understanding Journal*, 81(3):398–413, 2001.

[29] M. Girard and A. Maciejewski. Computational modeling for the computer animation of legged figures. In *Proceedings of ACM SIGGRAPH 1985*, pages 263–270. ACM SIGGRAPH, 1985.

[30] M. Gleicher. Motion editing with spacetime constraints. In Michael Cohen and David Zeltzer, editors, *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 139–148. ACM, apr 1997.

[31] M. Gleicher. Retargeting motion to new characters. In *Proceedings 0f ACM SIGGRAPH 98*, Annual Conference Series, pages 33–42. ACM SIGGRAPH, July 1998.

[32] M. Gleicher. Motion path editing. In *Proceedings 2001 ACM Symposium on Interactive 3D Graphics*. ACM, March 2001.

[33] M. Gleicher, H. J. Shin, L. Kovar, and A. Jepsen. Snap together motion: assembling run-time animations. In *Proceedings 2003 ACM Symposium on Interactive 3D Graphics*. ACM, April 2003.

[34] F. Sebastian Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.

[35] S. Guo and J. Roberge. A high-level control mechanism for human locomotion based on parametric frame space interpolation. In *Proceedings of Eurographics Workshop on Computer Animation and Simulation '96*, pages 95–107, August 1996.

[36] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[37] J. Harrison, R. Rensink, and M. van de Panne. Obscuring length changes during animated motion. *ACM Transactions on Graphics*, 23(3):569–573, 2004.

[38] J. Hodgins and N. Pollard. Adapting simulated behaviors for new characters. In *Proceedings of ACM SIGGRAPH 97*, Annual Conference Series, pages 153–162. ACM SIGGRAPH, August 1997.

[39] J. Hodgins, W. Wooten, D. Brogan, and J. O'Brien. Animating human athletics. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 71–78. ACM SIGGRAPH, August 1995.

[40] E. Hsu, S. Gentry, and J. Popović. Example-based control of human motion. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*, August 2004.

[41] D. James and K. Fatahalian. Precomputing interactive dynamic deformable scenes. *ACM Transactions on Graphics*, 22(3):879–887, 2003.

[42] O. C. Jenkins and M. Matarić. Deriving action and behavior primitives from human motion data. In *Proceedings of 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-2002)*, pages 2551–2556, 2002.

[43] E. Keogh, K. Chakrabarti, M. Pazzani, and Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proceedings of 2001 ACM SIGMOD International Conference on Management of Data*, pages 151–162, 2001.

[44] E. Keogh, T. Palpanas, V. Zordan, D. Gunopulos, and M. Cardle. Indexing large human-motion databases. In *Proceedings of the 30th International Converence on Very Large Data Bases*, 2004.

[45] M.-J. Kim, M.-S. Kim, and S. Y. Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *Proceedings of ACM SIGGRAPH 1996*, Annual Conference Series, pages 369–376, August 1996.

[46] T. Kim, S. Park, and S. Shin. Rhythmic-motion synthesis base on motion-beat analysis. *ACM Transactions on Graphics*, 22(3):392–401, 2003.

[47] H. Ko and N. Badler. Animating human locomotion with inverse dynamics. *IEEE Computer Graphics and Application*, 16(2):50–59, 1996.

[48] J. Korein and N. Badler. Techniques for generating the goal-directed animation of articulated structures. *IEEE Computer Graphics & Applications*, 2(9):71–81, November 1982.

[49] A. Lamouret and M. van de Panne. Motion synthesis by example. *7th Eurographics Workshop on Animation and Simulation*, pages 199–212, 1996.

[50] J. Lander. Working with motion capture file formats. *Game Developer*, 5(1):30–37, January 1998.

[51] J. Laszlo, M. van de Panne, and E. Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of ACM SIGGRAPH 96*, Annual Conference Series, pages 155–162. ACM SIGGRAPH, August 1996.

[52] J. Lee, J. Chai, P. Reitsma, J. Hodgins, and N. Pollard. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics*, 21(3):491–500, 2002.

[53] J. Lee and K. H. Lee. Precomputing avatar behavior from human motion data. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*, August 2004.

[54] J. Lee and S. Y. Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of ACM SIGGRAPH 99*, Annual Conference Series, pages 39–48. ACM SIGGRAPH, August 1999.

[55] J. Lee and S. Y. Shin. General construction of time-domain filters for orientation data. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):119–128, 2002.

[56] J. P. Lewis, M. Cordner, and N. Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 165–172. ACM SIGGRAPH, August 2000.

[57] Y. Li, T. Wang, and H.-Y. Shum. Motion texture: A two-level statistical model for character motion synthesis. *ACM Transactions on Graphics*, 21(3):465–472, 2002.

[58] C. K. Liu and Z. Popović. Synthesis of complex dynamic character motion from simple animations. *ACM Transactions on Graphics*, 21(3):408–416, 2002.

[59] F. Liu, Y. Zhuan, F. Wu, and Y. Pan. 3d motion retrieval with motoin index tree. *Computer Vision and Image Understanding*, 92(2-3):265–284, 2003.

[60] Anthony A. Maciejewski. Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE computer graphics & applications*, 10(3):63–71, May 1990.

[61] A. Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Academic Press, 2000.

[62] M. Mizuguchi, J. Buchanan, and T. Calvert. Data driven motion transitions for interactive games. In *Eurographics 2001 Short Presentations*, September 2001.

[63] L. Molina-Tanco and A. Hilton. Realistic synthesis of novel human movements from a database of motion capture examples. In *Proceedings of the Workshop on Human Motion*, pages 137–142. IEEE Computer Society, December 2000.

[64] J.-S. Monzani, P. Baerlocher, R. Boulic, and D. Thalmann. Using an intermediate skeleton and inverse kinematics for motion retargeting. *Computer Graphics Forum*, 19, August 2000.

[65] M. Neff and E. Fiume. Modeling tension and relaxation for computer animation. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, pages 81–88, July 2002.

[66] M. Neff and E. Fiume. Aesthetic edits for character animation. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003*, pages 239–244, July 2003.

[67] J. O'Brien, B. Bodenheimer, G. Brostow, and J. Hodgins. Automatic joint parameter estimation from magnetic motion capture data. In *Proceedings of Graphics Interface (GI'97)*, pages 53–60, May 1997.

[68] S. I. Park, H. J. Shin, and S. Y. Shin. On-line locomotion generation based on motion blending. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, July 2002.

[69] K. Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, March 1995.

[70] K. Perlin and A. Goldberg. Improv: a system for scripting interactive actors in virtual worlds. In *Proceedings of ACM SIGGRAPH 96*, pages 205–216. ACM SIGGRAPH, August 1996.

[71] Z. Popović and A. Witkin. Physically based motion transformation. In *Proceedings of ACM SIGGRAPH 99*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, August 1999.

[72] K. Pullen and C. Bregler. Motion capture assisted animation: texturing and synthesis. *ACM Transactions on Graphics*, 22(3):501–508, 2003.

[73] L. Rabiner and B.H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, Englewood Cliffs, NJ 07632, 1993.

[74] P. Reitsma and N. Pollard. Perceptual metrics for character animation: sensitivity to errors in ballistic motion. *ACM Transactions on Graphics*, 22(3):537–542, 2003.

[75] P. Reitsma and N. Pollard. Evaluating motion graphs for character navigation. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*, August 2004.

[76] C. Rose, M. Cohen, and B. Bodenheimer. Verbs and adverbs: multidimensional motion interpolation. *IEEE Computer Graphics and Application*, 18(5):32–40, 1998.

[77] C. Rose, B. Guenter, B. Bodenheimer, and M. Cohen. Efficient generation of motion transitions using spacetime constraints. In *Proceedings of ACM SIGGRAPH 1996*, Annual Conference Series, pages 147–154. ACM SIGGRAPH, August 1996.

[78] C. Rose, P. Sloan, and M. Cohen. Artist-directed inverse-kinematics using radial basis function interpolation. *Computer Graphics Forum*, 20(3), 2001.

[79] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[80] A. Safonova, J. Hodgins, and N. Pollard. Synthesizing physically realistic motion in low-dimensional, behavior-specific spaces. *ACM Transactions on Graphics*, 23(3), 2004.

[81] A. Schödl and I. Essa. Controlled animation of video sprites. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, August 2002.

[82] A. Schödl, R. Szeliski, D. Salesin, and I. Essa. Video textures. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 489–498. ACM SIGGRAPH, July 2000.

[83] H. J. Shin, J. Lee, M. Gleicher, and S. Y. Shin. Computer puppetry: an importance-based approach. *ACM Transactions on Graphics*, 20(2):67–94, April 2001.

[84] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of ACM SIGGRAPH 1985*, pages 245–254. ACM SIGGRAPH, July 1985.

[85] D. Sturman. A brief history of motion capture for computer character animation. In *"Character Motion Systems", Notes for SIGGRAPH '94, Course 9*, 1994.

[86] S. Tak, O.-Y. Song, and H.-S. Ko. Spacetime sweeping: an interactive dynamic constraints solver. In *Computer Animation 2002*, pages 261–270, June 2002.

[87] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[88] S. Theodore. Understanding animation blending. *Game Developer*, pages 30–35, May 2002.

[89] D. Tolani, A. Goswanmi, and N. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62:353–388, 2000.

[90] M. Unuma, K. Anjyo, and T. Tekeuchi. Fourier principles for emotion-based human figure animation. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 91–96. ACM SIGGRAPH, 1995.

[91] R. C. Veltkamp, H. Burkhardt, and H. P. Kriegel. *State-of-the-Art in Content-Based Image and Video Retrieval*. Kluwer Academic Publishers, 2001.

[92] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 216–225, 2003.

[93] J. Wang and B. Bodenheimer. An evaluation of a cost metric for selecting transitions between motion segments. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003*, July 2003.

[94] D. Washburn. The quest for pure motion capture. *Game Developer*, December 2001.

[95] C. Welman. Inverse kinematics and geometric constriants for articulated figure manipulation. Master's thesis, Simon Fraser University, 1989.

[96] D. Wiley and J. Hahn. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Application*, 17(6):39–45, 1997.

[97] D. Winter. *Biomechanics and Motor Control of Human Movement*. Wiley, New York, 1990.

[98] A. Witkin and Z. Popović. Motion warping. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 105–108. ACM SIGGRAPH, August 1995.

[99] W. Wooten and J. Hodgins. Dynamic simulation of human diving. In *Proceedings of Graphics Interface (GI'95)*, pages 1–9, May 1995.

[100] W. Wooten and J. Hodgins. Simulating leaping, tumbling, landing, and balancing humans. In *IEEE International Conference on Robotics and Animation*, volume 1, pages 656–662, 2000.

[101] J. Zhao and N. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):313–336, October 1994.

[102] V. Zordan and J. Hodgins. Motion capture-driven simulations that hit and react. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, July 2002.

[103] V. Zordan and N. Horst. Mapping optical motion capture data to skeletal motion using a physical model. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003*, August 2003.