AN ABSTRACT OF THE DISSERTATION OF

Madhusudhanan Srinivasan for the degree of Doctor of Philosophy in Computer Science presented on February 9, 2009.

Title: Behavior Graphs for Data-driven Animation of 3D Characters

Abstract approved: _____

Ronald A. Metoyer

In this dissertation, we present a user-in-the-loop method for the design of an interactive motion data structure that benefits from the advantages of both motion graphs and blend-based techniques. Our novel approach automatically analyzes a traditional motion graph built from labeled motion clips. The result is a more condensed, coarser graph which we call the Behavior Finite State Machine (BFSM). Each node of the BFSM represents a single behavior that may be continuously parameterized. An edge in the BFSM represents a valid transition between two behaviors, with the transition probability indicating the likelihood of such a transition. Our focus is on user-centered, semi-automatic methods for aiding in the construction and editing of such machines. Since the transitions and parameterized behavior spaces are based on constructing time-warps between motion clips, we present an intuitive process that allows the user to construct these data structures necessary for BFSM design. We present

the results of our approach using two dynamic interactive examples, locomotion and martial arts. We demonstrate the use of the BFSM to generate controllable motion in real-time, to synthesize motion offline using A* search, and to generate autonomous character navigation with obstacle avoidance in a virtual environment. We conclude with a discussion on the strengths and weaknesses of our approach.

Behavior Graphs for Data-driven Animation of 3D Characters

by

Madhusudhanan Srinivasan

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented February 9, 2009
Commencement June 2009

Doctor of Philosophy dissertation of Madhusudhanan Srinivasan presented on February 9, 2009.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Madhusudhanan Srinivasan, Author

ACKNOWLEDGEMENTS

I express my deepest gratitude to my father and mother who have always been a great source of encouragement for my graduate studies and an inspirational example. I thank them for their support and faith in me, and for all the sacrifices they made, so that I could achieve the educational goals that I had set for myself. I am deeply indebted to my sister, Anitha, whose unfailing support, sacrifice and love has made me worthy of being an example to her. I am thankful to my family for putting all their faith in me, and never doubting my abilities.

I would also like to thank Sandra for her un-ending support and motivation. She has always stood by me during the times I doubted myself, and offered courage and love. I certainly appreciate her long and patient wait in getting my graduate degree, and thank God with all humility for making her a part of my life.

I deeply thank my advisor Dr. Ron Metoyer for his constant encouragement, and guidance. He has been a great source of inspiration to me. His patient advice and counsel have always shaped and guided my capabilities in the right direction. I would also like to thank the faculty of the nVidia Graphics Lab, School of EECS - Dr. Eric Mortensen, Dr. Mike Bailey, and Dr. Eugene Zhang, whose insightful discussions and inputs have always been priceless. I express my gratitude to all my other committee members, for their time and efforts, to serve on my

committee. Thank you all very much.

I would also like to thank my friends at the nVidia Graphics Lab for making it a fun place to work for me, and for their valuable support and feedback. I would like to thank my life long friends in New York and Seattle for their enduring support. I would also like to thank all my friends in Corvallis and Portland who have always expected the best from me.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF APPENDIX FIGURES

DEDICATION

To Amma, Appa, Anitha and Sandra, whom I owe everything I am today.

Chapter 1 – Introduction

Generating realistic motion for character animation is a challenging problem in computer animation. Many synthetic scenes such as virtual environments, video games, and movie special effects involve digital characters that must be controlled and directed at a high-level. With the gain in popularity of 3D interactive environments, it is evident that real-time, controllable character motion is desirable. Of particular importance in these environments are human characters that move about the environment and react to user-control in a believable manner

Motion capture animation [3] is the process of recording motion data in real time from live actors and mapping it to computer characters (Figure 1.1). Motion capture technology has become more popular over the last decade because it can capture subtle human motion and thus add believability and realism to animated characters. However, it is a time-consuming and a labor-intensive process, and is not easily accessible to most users. In addition, it is a challenge to combine user control with captured examples without affecting the quality of motion it represents. Research in data-driven character animation aims to combine high-level control with existing motion data to overcome these challenges.

With the recent development of low-cost motion capture systems [40] and the need

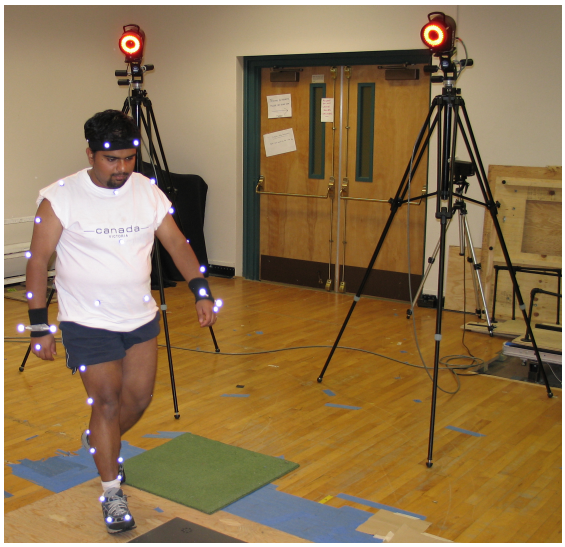


Figure 1.1: *Motion capture.*

for end-user 3D content such as that seen in SecondLife [36], it is becoming more and more evident that users should not only be able to customize the geometry and appearance of their 3D Avatars, but also their behavior and motion. This work is an attempt to present a "user-in-the-loop" motion design and generation process for interactive motion settings, using motion-capture data. The ability to quickly organize and design motion data structures that enable high-level control of animated characters could lead to advances in character motion for many application domains including training applications, games, educational content, and the recently popular 3D internet virtual worlds.

Our approach is to help the user leverage the available motion sequences to build a structure that leads to controllable and responsive character motion. To do this

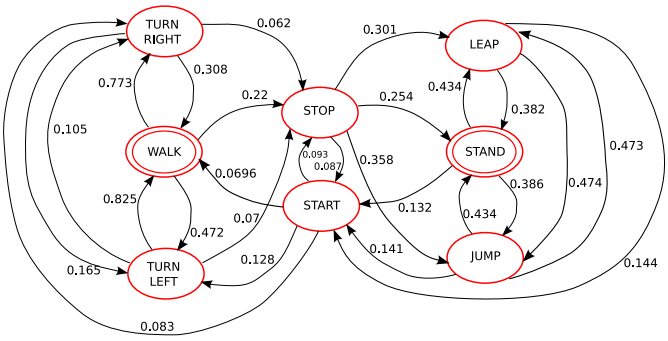


Figure 1.2: *An example of a Behavior Finite State Machine (BFSM). The weighted directed graph is made up of behavior states as nodes. The edges and their weights represent valid transitions between behaviors and their likelihoods. The structure of the BFSM and the weights are automatically calculated by analyzing a motion graph constructed from the motion clips representing the different behaviors.*

effectively, the user should be included in a tight loop with the motion design tools to facilitate experimentation with the sequences. In our approach, the user designs *behavior states* and transitions between these behavior states to create a *behavior finite state machine (BFSM)* (Figure 1.2). The result is a hybrid motion generation approach that combines the advantage of continuous control seen in blend-based approaches with natural transitions present in motion graph approaches [2, 21, 23, 26, 31, 27, 25, 43, 52, 67].

There are three important insights that lead to the solution presented in this dissertation.

- As is well known, **Motion Graph** based methods for motion generation are capable of creating natural motion transitions. However, these methods

sometimes suffer from "wandering" artifacts when motion sequences are not available to meet the desired control needs of the user [26].

- **Motion Blending** techniques are capable of creating highly controllable continuous behaviors [27]. A combination of Motion Graphs and Motion Blending is clearly a desirable goal as evidenced by the recent body of work [20, 57, 54, 43, 23, 42, 28] that is based on constructing hybrid graphs such as the one in Figure 1.2, for interactive motion synthesis.

- Users must be provided with high level methods for interfacing with motion data to quickly create and manipulate motion data structures such as the one shown in Figure 1.2. To this end, we attempt to automate parts of the process while giving the user opportunities to design the motion with intuitive mechanisms.

As Treuille et al. note in their recent work, building data-structures for hybrid approaches such as the BFSM, requires a great amount of skill and manual adjustment [65]. To aid the user in the design of the BFSM, we introduce our first contribution, a novel technique for "mining" the existing motion sequences to determine how to best connect the behavior states. We describe an approach that uses Q-learning to automatically infer the structure of a BFSM (connectivity and probabilities) from a labeled corpus of motion sequences. The structure is augmented by transition probabilities that indicate the likelihood of transitions between behavior states in the BFSM. We demonstrate the effectiveness of the

algorithm on several motion libraries.

Once the structure is automatically inferred, the user may prefer to fine-tune the BFSM by adjusting the transition probabilities. Modifying the weights directly on the BFSM edges may prove cumbersome, and lead to undesirable results. In order to facilitate intuitive editing of the BFSM transitions, we introduce our second contribution: a novel approach to modify transition probabilities using linear and quadratic optimization techniques [9, 8]. Our approach allows users to edit the BFSM using high-level inputs based on their perception of natural looking behavior transitions, as opposed to manually tweaking individual transition probabilities.

In order to synthesize natural motion with the BFSM, manual tweaking and low-level editing of motion-clips is generally required of the user. We attempt to address this problem by presenting our third contribution, a process that allows the designer to focus on designing motion at a high-level, as opposed to low level editing of motion clips to meet specific constraints for transitions. The underlying data structure in our approach is the registration curve introduced by Kovar and Gleicher [27]. Our contribution lies in our extensions to registration curves to assist in the creation of *cyclic* behavior states and *multi-point transition blends* between behavior states:

- **Cyclic Behaviors:** First, we introduce the self-transition warp as an extension to compute a self-warp within a registration curve. A self-warp eliminates the need to fine-tune and pre-process the clips (belonging to the same

behavior) to enforce constraints such as "the clips should begin and end at the same foot-step", or, "all clips should be of the same length".

- **Multi-point Transitions:** We also describe a novel approach to construct transitions between behaviors. Using our framework, users can define multiple transitions, each starting at a different time position along a behavior. Our multi-point transition approach allows for quick and responsive transitions between behaviors, without waiting for a transition that might otherwise only be available at the end of the motion segment. Multi-point transitions also allow behaviors to be represented using longer motion sequences, reducing artifacts during cyclic playback and allowing more variability.

These extensions allow one to author behavior states as continuous, and possibly parameterized spaces that will make up the behavior states of the BFSM. Our approach provides several advantages over previous methods. First, there is no need to 'force' motion to a particular "hub" motion sequence in order to transition between behavior states. In addition, the character does not have to transition between two existing sequences, rather, the character can transition between two interpolated sequences that are generated on the fly.

The above contributions facilitate the creation of BFSMs from unedited sequences of motion capture data. Our only assumption is that the sequences are segmented into somewhat atomic behaviors and that they are labelled. The assumption is reasonable since several approaches for motion segmenting and labeling have been

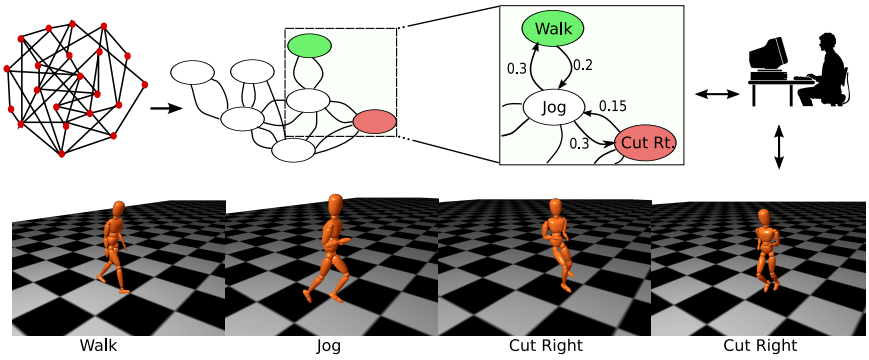


Figure 1.3: *An overview of our approach. We first analyze a motion graph to automatically build a coarse graph structure with weighted edges. The designer can populate the structure with labelled motion clips and design transitions. The result is a Behavior Finite State Machine (BFSM) that when combined with control input produces responsive and natural character motion.*

demonstrated in recent years [1, 25].

1.1 Overview

Our approach consists of three phases, the first one being automated, and the second and third phase involving both an automated and a user-driven component (Figure 1.3). We begin with the assumption of a library of motion with labeled sequences segmented into behaviors (e.g. running, walking, jumping, etc.). Our mining algorithm automatically analyzes the motion clips to produce a coarse graph called the Behavior Finite State Machine (BFSM). Next, we populate the BFSM using the labeled motion clips. We provide a time-warping tool that aids the user in the process of constructing data-structures necessary for synthesizing

behaviors and transitions between behaviors. At this point, the BFSM is capable of generating controllable motion and the user can interactively fine-tune and adjust the transition probabilities to generate a BFSM that meets the user's desires.

In Chapter 2, we review related work. We outline a novel approach to motion-graph mining that automatically discovers the structure of a BFSM from a set of labeled motion-clips in Chapter 3. Next, we present a heuristics-based optimization approach for semi-automatic editing of the BFSM (Chapter 4). We use this approach to fine-tune the BFSM we obtain from our graph-mining step. In Chapter 5, we describe our technique for building behavior states and transitions between the behavior states, to complete the BFSM design. Chapter 6 elaborates on different approaches to leverage our BFSM to synthesize locomotion that is controllable in real-time, plan motions off-line, and avoid obstacles interactively. We present our results in Chapter 7. We demonstrate the use of the BFSM to generate controllable motion in real-time, use the BFSM for offline motion synthesis using A* search, and we generate autonomous character navigation with obstacle avoidance in a virtual environment. A discussion of the strengths and weaknesses of our approach is presented in Chapter 8.

Chapter 2 – Related Work

There has been a great deal of interest in character motion synthesis from motion capture data in the past decade. For the purpose of this work, we broadly classify data-driven character motion synthesis into *Model-based* approaches and *Memory-based* approaches (Figure 2.1).

A Model-based approach aims at generating a motion model using motion examples. The model could be a *statistical* model, or a *physics-based* simulation model. The former approach strives to learn the statistical nature of a library of motion-capture data in order to generate new, continuous motion sequences with similar statistical properties [4, 5, 35, 39, 47]. A statistical model can generate new motion sequences with characteristics or style similar to that of the motion library. However, they do not provide an intuitive approach to incorporate user inputs. As a result, these models do not provide a convenient framework for controlling animated avatars. In a physics-based approach, forward-dynamics and control algorithms are used to simulate a physically modeled rigid-body human. Torques and forces calculated on the basis of motion-capture data are applied at joints and limbs to produce a desired movement [69, 70, 71]. Motion-capture driven simulations are desirable in applications where controllable and reactive human motion is required, because the generated motion remains close to and therefore retains the

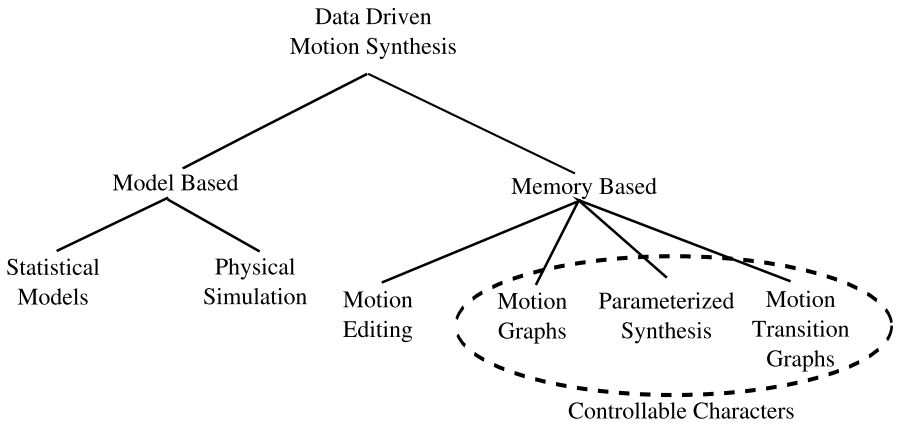


Figure 2.1: *A classification of data-driven motion synthesis techniques for a 3D character.*

important characteristics of original motion examples, while deviating sufficiently to accomplish a given task. However data-driven physical simulation has a long way to go before it can produce believable and controllable real-time locomotion. Numerical integrators slow down the simulation to below interactive rates. Parameters in a data-driven simulation require considerable offline experimentation before they can be incorporated into the simulation. Additionally, the realism of the simulation is tied closely to these parameters. Recent efforts have focussed on switching to physical-simulation only when a dynamic response to an input is needed [72, 37].

Unlike model-based techniques, our work is based on a memory-based approach that relies directly on the original motion example to synthesize motion. Motion frames are stored in memory and are either transformed, re-arranged or blended together to generate motion in response to user-input. Figure 2.1 shows different

memory-based approaches. *Motion editing* focusses on techniques for editing sequences of motion-capture data, in order to achieve user-constraints [6, 16, 17, 18, 34, 45, 68]. They are not capable of generating a continuous stream of motion, given a library of motion-capture data. In the following sections, we review three memory-based approaches that form the basis for the synthesis of controllable character animation-*Motion-graphs*, *Parameterized synthesis*, and *Motion Transition Graphs*.

2.1 Controllable Characters

Synthesizing controllable character motion from motion capture data is an interesting research challenge. Solutions to this problem have evolved from blend-based techniques for motion data [67, 52, 27] to sequencing of motion data using graphs [26, 2, 31, 19, 32], to more recent work on combinations of the two approaches as well as optimal control methods [23, 43, 42, 30, 60, 28, 57, 10, 54, 20, 65].

2.1.1 Motion Graphs

The idea of using motion graphs for motion synthesis was inspired by the controllable sprites in the video textures work by Schödl et al. [55, 56]. Since this work,

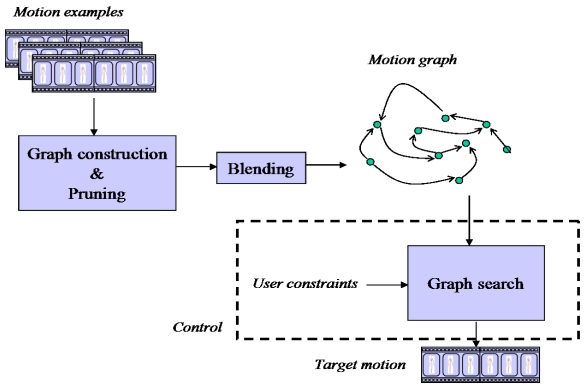


Figure 2.2: *Basic principles of motion graph approach.*

several researchers have applied similar techniques to motion data [2, 21, 23, 26, 31].

In a motion graph approach, a collection of motion sequences is represented as a weighted directed graph. Each node represents either a frame or sequence of frames. A directed edge between two nodes represents an acceptable transition between the frames representing the two nodes. Every edge is associated with a weight that measures the cost of transitioning between frames. A motion-graph is typically pruned to eliminate edges whose weights are above a certain threshold. An overview of the approach is shown in Figure 2.2.

The graph construction is aided by a similarity metric between frames (Appendix A.2). Synthesizing novel motion sequences is then reduced to the problem of finding a walk in this graph. Graph walks that satisfy certain properties can be extracted using algorithms from planning and graph theory, providing control over synthesized motion. A primitive form of a graph structure called *move trees* [38] has been used for online motion generation in video games. These tree structures also

represent connections between carefully chosen motion segments in the database. The transitions between segments are manually blended using interactive tools.

Kovar *et al.* [26] construct a graph with frames as nodes and a sequence of frames representing transitions between nodes. Examples are annotated manually with high-level information such as "walking", "running" etc. A similarity metric based on a point cloud representation of an articulated figure (Appendix A.2) is used to identify a pair of frames that are close up to a threshold. A transition is created by performing a blend over a window centered at these frames. Since the graph should be capable of generating motion indefinitely, transitions to nodes from where there are no outgoing arcs are pruned out. Graph walks satisfying user constraints are computed using a global branch-and-bound search technique. Every user specified constraint is associated with a cost function that measures how well the constraint is satisfied during the search. Some of the constraints include path-following and high-level keyframing.

Arikan and Forsyth [2] use a randomized global search over a hierarchy of motion graphs to search for motions satisfying a variety of constraints. A node of the graph at the finest level represents a motion sequence, and a set of edges between nodes is a set of all individual frame-transitions between the corresponding sequences. The edges are then clustered temporally based on where they originate from, and end at, in the connecting motion sequences. The graph at the highest level is formed by retaining all the nodes from the previous graph and using clusters to represent edges between nodes. Intermediate graphs are generated by a binary

partitioning of the edge clusters from the top-most graph. The search is initialized with random graph paths at the top level. Paths are mutated by combining and replacing multiple edges with an edge from a coarser graph, or refining a path using a finer graph. User constraints aid the search in evaluating different paths. The optimal path is output as a motion sequence satisfying user requirements. Their technique provides the user with the ability to specify a combination of posture and spatial constraints.

Lee et al. [31] also combine motion graphs with clustering. Frames represent nodes and a transition between frames is determined by a transition probability computed from the frame-similarity measure (Appendix A.2). Pruning is done on the basis of consistency in foot-plants, a transition threshold and removing dead-ends. They cluster similar frames together to generalize data, and capture the distribution of frames and transitions. A depth-limited *cluster tree* is built at each motion frame, based on transitions from that frame. Each cluster path in this tree from the root frame represents a collection of actions available to the avatar. From the constraints provided by the user, a cluster path is determined. Given this cluster path, the most probable sequence of frames is computed to synthesize the required motion. This approach enables real-time control of the avatar because the technique uses a local search to find the most probable motion.

While all of the above approaches automate the process of graph construction, they do not provide the ability to control the graph structure. Gleicher et al. address this issue in [19]. They note that the controllability of a character depends much

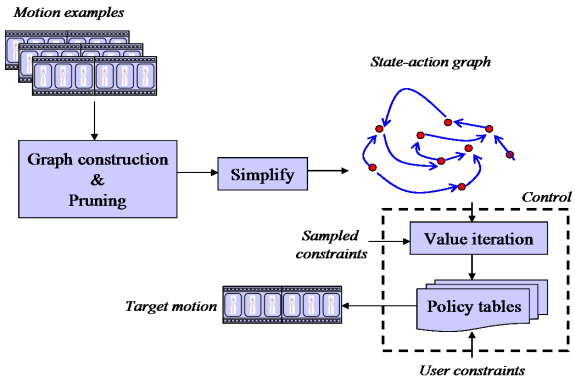


Figure 2.3: *Steps involved in precomputing control policy using motion graphs.*

on the connectivity of the graph. Their graph structure is similar to that used by Kovar et al. [26]. Similar frames with multiple incoming and outgoing edges, however, are grouped together into a *match set*. Frames in a match set are called *match frames*. For every match set, a common frame closest to all the match frames is identified. A displacement-map [6] is then constructed for every motion example (having the match frame) to pass through the common pose. These common poses, together with the motion transitions, are called *hubs* and a smooth transition is guaranteed across these hubs. This creates a more structured graph with multi-way smooth transitions, thus adding more controllability to the character. Internal constraints like foot-plants are identified in the original examples, and are enforced by carefully choosing a common pose so that changing the motion examples does not violate the constraints. A user can direct the character by specifying high-level actions like "punch", "dodge", "kick", "walk" etc. One of the drawbacks of the approach is that the transitions are constrained to occur only at the hubs, thus preventing fine-grained control over the synthesized motion.

In the above approaches, user control was mapped to the graph using a search method. Lee *et al.* [33] build upon their previous work in [31] and use the motion graph to precompute a control policy that indicates what the character should do in a given situation. The motion graph in [31] is further abstracted by collapsing successive frames with a single outgoing transition to represent a *state-action* model $\{(S, E), A\}$ for the character (Figure 2.3). Given the current state S representing a pose of the character, and a target state E representing user constraint in the form of a desired character location or a punch location, the control policy computes an optimal action A for the character. User constraints in the form of grid-sampled target positions are associated with every state S of the avatar. The state-action space is modeled as a first-order markovian process. A table indexed by (S, E) pair defines a control policy that returns a corresponding optimal action A. The control policy is learned by performing value-iteration, which is a standard reinforcement learning technique [61]. Value-iteration iteratively explores the state-action space to compute optimal actions A for every (S, E) pair. In their approach, each behavior is represented by a separate policy table, and multiple behaviors are composed by a weighted combination of the policy tables. The main drawback to this approach is that it is memory intensive. Each behavior requires an $O(NM)$ policy table, where M is the number of constraint samples, and N is the number of states. In addition, precomputed policies do not allow the user to change the optimization objectives at runtime.

Each of the above motion-graph based approaches creates very natural motion

that achieves user constraints such as path following, keyframes, or task performance. Each new constraint only adds another term to the cost function during the search, and does not affect the time-complexity of the search. Since a motion graph is based on a similarity metric between poses, it automatically encodes plausible transitions between different behaviors. For instance, a motion-graph would automatically rule out a transition between a walk and a sitting posture. In our work, we exploit this property in order to discover the structure of a Behavior Finite State Machine (BFSM).

Unfortunately motion graphs do not scale well with motion examples. Graph structures need $\theta(N^2)$ space for N examples. Searching for a suitable motion sequence in large graphs may take considerable time, and hence may not be suitable for real-time applications. To address this problem, the "snap-together" work presented by Gleicher et al. [19] relies on preprocessing the data for an efficient simple graph structure that can be queried at run-time to generate natural transitions [19]. Likewise, Srinivasan et al. build a similar preprocessed structure, the "Mobility Map", for synthesizing natural transitions [59]. The mobility map is augmented with spatial information to enable real-time path planning with natural-looking transitions.

In addition to scalability and efficiency issues, the performance of the technique depends on many parameters such as the examples selected, thresholds for pruning the graph, weighting different constraints etc. Nevertheless, the synthesized motion is very close to the original motion examples, and hence believable.

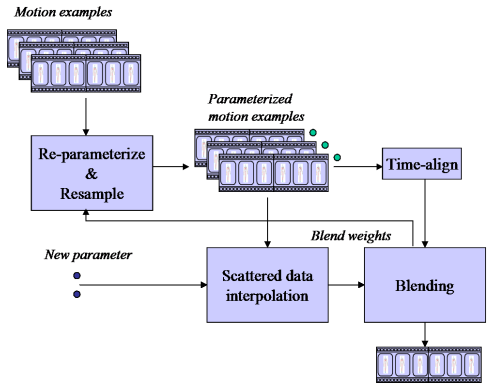


Figure 2.4: *Principles of parameterized motion synthesis.*

2.1.2 Parameterized Motion Synthesis

In parameterized motion synthesis, new motion clips are synthesized by interpolating and blending multiple motion examples to generate new sequences that meet the desired motion constraints [27, 25, 43, 52, 67]. Parameterized synthesis allows a user to control a motion task using relevant features of motion-examples, like speed and direction of a walk or the position of foot during a kick move. These relevant features, or parameters, can be computed for a given example motion with a user-provided function f. The input to the parameterization function f is the example motion, and the output is a meaningful parameter such as the position of the foot during a kick move.

The goal of parameterized motion generation is to compute the inverse of the parameterization function, f^{-1}. The input to such a function would be a parameter, such as the position of a kick, and the output would be a motion executing the

kick. Though, f^{-1} does not have a closed-form solution, it can be computed indirectly. Consider an example where a user desires to synthesize motion for a new kick location, using examples of different kicks. First, all motion examples are parameterized to associate a kick-location with each motion-clip. When a new kick location is supplied by the user, clips corresponding to nearby kick-locations can be identified and blended with appropriate weights to synthesize motion corresponding to the new location. The blend weight for each clip depends on how "far" its kick-location is from the specified kick-location in the parameter space. This technique for computing the blend weights is a form of *scattered data interpolation* [49]. Scattered data interpolation assigns a weight to each of the examples based on the distance between its parameters and the new parameter. The accuracy of the kick-location increases with the number of examples with nearby kick locations. For this reason, the parameterized examples are re-sampled to generate more motion examples (Figure 2.4).

Wiley and Hahn were among the earliest to introduce the notion of interpolated motion synthesis [67]. The parameter space is re-sampled to a regular grid and the blend weights are computed by performing a tri-linear interpolation, instead of scattered data interpolation. Their technique is applied to simple tasks like reaching, and periodic motion of the foot. A demonstration of interpolated synthesis of walking on different degrees of a slope is also presented. Blending is done by uniformly time-aligning the motion examples.

Rose et al. [52] find an interesting application for interpolation synthesis. Motion

examples are parameterized based on emotional parameters such as "happiness-sadness", "knowledgeable-clueless", etc. These parameters are called *adverbs*. The motion examples themselves are called *verbs*, and include walking, running, idling and reaching. Each set of verbs are structurally identical, and can be blended after a linear time-alignment. Blend weights are determined using a radial-basis [46] function for each motion example. A directed verb-graph is also constructed by the user that defines transitions between different sets of verbs. This provides the user with the ability to specify required actions in addition to the adverb parameter. For instance, the user could synthesize a moderately sad walk, followed by a totally clueless run. Park et al. [43] take a similar approach, except that they combine all the verbs into a single big verb: locomotion. Motion examples are parameterized based on style, speed, and turning angle. A novel *incremental time-warping* technique is used to blend motions with varying speeds. They also introduce a procedure to blend postures using quaternion representation.

One of the essential requirements for time-warping is to identify similar events across different motion examples. One approach is to manually identify and mark foot-plants in motion examples. Kovar and Gleicher [27] introduced an automated way to determine frame correspondences across multiple motion examples. This was done by first building a matrix of distances between a base example and each of the other examples. A standard dynamic time-warping algorithm is then applied to determine a set of corresponding frames. Fitting a spline through each set of corresponding frames across all the examples yields a *registration curve*. Every

point on this curve corresponds to a set of frames that can be blended together. An approach to automatically identify and enforce foot-plant constraints along the registration curve is also provided.

Structurally similar examples must be carefully selected to perform a blend. A walk motion, and a sit motion can be blended, but would not yield any meaningful result. In the above approaches, similar motion examples were hand-picked. In their more recent work, Kovar and Gleicher [25] propose a new technique to automatically identify structurally similar motion sequences, given a query motion sequence (Appendix A.3). Their technique is based on building a web of time-alignment curves by chaining individual time-warp curves together. They use this web to extract similar motion examples, and parameterize and interpolate using the above techniques.

An advantage of parameterized motion synthesis is that it provides an intuitive real-time interface to synthesize novel motion. Interpolated motion is believable as long as examples are carefully chosen. Quick and efficient generation of motion clips from few examples make it a preferred data-driven motion synthesis approach.

However, interpolated synthesis of character animation has several drawbacks. They require the user to carefully choose motion examples to interactively control via a specific parameter. For instance, if the animator wants to control the foot position in a particular kick, then interpolation techniques would require a family of motion sequences where the only difference in motion is the kick-location. Such a

data-set is hard to reproduce and laborious to gather. Moreover, the data required is exponential in the number of dimensions of the parameter. In addition, any two logically similar motion clips are not guaranteed to produce an artifact-free result on blending. For instance, two reaching examples having different foot-plants will not produce a convincing blend. Kovar and Gleicher [25] consider visual inspection to be the only reliable way to determine if two motion examples can be interpolated.

Interpolation techniques enrich a motion library by creating new motion sequences from existing ones. Motion graphs provide a generic framework for motion rearrangement. However, motion graphs suffer from the drawback that the quality of motion is sensitive to the amount of data used to construct the graph. Motion graphs require a large library to generate high-quality motion. Recently, there has been a growing interest in combining motion graph approaches with interpolation techniques to construct motion transition graphs for motion synthesis.

2.1.3 Motion Transition Graphs

The notion of a motion transition graph is built upon the ideas from verbs and adverbs introduced by Rose et al. [52]. As mentioned in Section 2.1.2, Park et al. build upon this work by creating one large "verb" of walks and runs, and parameterize the space on speed and turning direction [43]. They extend their work with the notion of a motion transition graph that has a set of labeled sequences

as nodes [42]. A node in a motion transition graph represents a group of basic motions of an identical structure, and an edge represents the transition from a blended motion to a blended motion (possibly including self-transition). Given a stream of motion specifications, the graph is traversed from node to node, while blending motions at nodes and making transitions at edges. Kim et al. present a similar technique [23] with rhythmic dance motions. Kwon and Shin introduced a technique to automatically identify behavior spaces by analyzing and grouping unlabeled sequences with similar footstep pattern [28]. A hierarchical motion transition graph is then constructed, that can generate controllable motion, given a stream of motion specifications. A conceptually similar idea has been used by Lai et al. for animating groups of discrete agents, such as flocks, herds, or small crowds [29].

Motion transition graphs have also been used for behavior planning to automatically generate realistic motions for animated characters [30]. Motion clips are abstracted as high-level behaviors and associated with a behavior finite-state machine (a motion transition graph) that defines the movement capabilities of a virtual character. Motion is generated by a planning algorithm that performs a global search on the Behavior Finite State Machine (BFSM), to reach a user-specified goal. Similar to their work, our approach combines Behavior FSMs and motion capture. While their framework assumes the existence of a BFSM constructed by the user, we provide a novel approach to automatically determine the structure and transitions of a BFSM from a motion graph constructed from annotated clips.

Unlike their BFSM, our BFSM edges are weighted, which allows us to implement an online, real-time planner based on shortest paths through the BFSM.

The motion transition graph approach is an attractive technique, especially for video games, because interesting animations can be generated using a relatively small amount of motion data. All of these approaches address the problem of generating artifact-free parameterized motion. However, they generally place fine-grained constraints on the set of motions that can form a behavior state. For instance [43] requires that their motion clips contain only two foot-steps and that they always start with the right foot forward. In [42], clips within a node are assumed to be very short and have identical footplants. Additionally, transitions between motions are constrained to occur at the beginning and at the end of a motion segment.

We build upon these ideas of constructing an abstract motion transition graph, however we do not place any hard constraints on the length of clips or on footplant patterns. Our framework allows for the user to quickly assemble labeled motion clips as well as register and connect them with transitions that originate and terminate at multiple-points along the clips. We use registration curves [27] to align motion clips within a behavior state (or space) and to construct transitions between behavior states.

2.2 Structure from Motion Graphs

In a majority of the work above, the motion transition graph was assumed to be specified and constructed manually by the user. Automating the process of constructing a motion transition graph is an interesting problem that has received considerable attention recently [10, 20, 41].

Chiu et al. present a dual representation that encodes a BFSM within a motion graph [10]. The annotations in the motion graph correspond to the state labels. However, they assume a hand-built BFSM to begin with and construct the motion graph based on transitions allowed in the BFSM. Our approach is directly opposite - we assume that we initially only know the state-labels of the BFSM. The transitions between the states and their likelihoods are automatically inferred from a motion graph constructed with state-annotated motion data.

Beaudoin et al. recently presented a technique that distills a motion graph to bring out the underlying structure of a motion corpus [41]. The technique combines segmentation and clustering of motion clips into Motion Bundles. In this paper, we assume that our data is already segmented and labeled.

One recent approach that shares many of our goals is Parametric Motion Graphs [20]. In this technique, Heck and Gleicher use sampling methods to identify and represent good transitions between parameterized behavior spaces. Their sampling strategy addresses the issue of determining the range of parameters over which

a transition between two states is valid. However, their approach is limited to transitioning only at one point near the end of a clip. This constrains the length of the motion clips in order for the character to remain responsive to user inputs. We attempt to remove these restrictions on the design of BFSMs.

Similar to Kim et al., we also represent a transition in the BFSM as a "weighted" directed edge [23]. Kim et al. define a transition likelihood between behaviors as a weighted combination of a "kinematic" and a "behavioral" likelihood. The kinematic likelihood takes into account the pose transition cost and measures pose-similarity between the two behaviors. The behavioral likelihood is based on an "expected number of transitions" between the two behaviors. Our likelihood measure unifies the kinematic and behavioral continuity into a single measure. This eliminates an extra weighting parameter that must be specified by the user to combine the two components. In our approach, kinematic continuity is implicitly enforced by the underlying motion graph which is used to learn the structure of the BFSM. Behavioral continuity is then calculated based on the frequency of actual transitions that are observed between the two behaviors.

2.3 Navigation and Obstacle Avoidance

Creating characters that can navigate autonomously in 3D environments has been a popular area of research. Craig Reynolds broadly categorizes and discusses various obstacle avoidance approaches for autonomous agents [50, 51]. One approach

to incorporating obstacle avoidance into characters uses the Artificial Life model. Funge et al. proposed a framework for cognitive modeling of reasoning and planning for intelligent characters [14]. Terzopoulos et al. developed a cognitive model for motor control and perception in fishes [63, 64]. The artificial life model was capable of synthesizing flocking, herding, schooling etc., behaviors for fishes. Others like Julien et al. and Choi et al. have dealt with navigation and obstacle avoidance in a two-stage manner [44, 11]. The first stage involves path-planning, followed by a motion-synthesis stage to generate the desired character motion. Recently, Glardon et al. presented a technique to animate an autonomous character in a dynamic environment using a similar two-phase approach [15].

Much work has been presented recently in the area of learning real-time navigation and obstacle avoidance controllers for data-driven characters. Ikemoto et al. proposed a real-time character navigation controller based on reinforcement learning techniques [22]. Their approach is based on motion graphs, and thus suffers from responsiveness issues due to insufficient motion data. Lee et al. learn a controller for a boxing character using a similar reinforcement learning approach [32]. More recently, Treuille et al. [65] presented a technique to learn near-optimal controllers. They use a value-iteration technique with low-dimensional basis functions to learn controllers for navigation and obstacle-avoidance task. All of the above approaches eventually learn controllers that selects the best motion-clip to play from a discrete space of pre-recorded motion clips, given the environment around the character. Unfortunately, learning techniques do not scale very well to dy-

namic environments. Additionally, it is still not clear how reinforcement learning approaches can be extended to continuous behavior spaces like that of the BFSM.

In our work, we present a steering approach based on [50], and design a simple navigation controller and an obstacle avoidance controller on top of the BFSM.

Chapter 3 – Motion Graph Mining

A BFSM is a directed graph where nodes represent states and edges represent valid transitions between the states. A BFSM (or a motion transition graph) is usually hand-designed by the user whose goal is to generate controllable motion. For even a modest set of behaviors, the structure of the BFSM can become too complex and could involve a significant amount of design by an expert user. In the following sections, we present a "mining" approach to address this issue. Assuming that the user is supplied with labeled motion sequences that make up the states of the BFSM, our technique automatically discovers potential edges in the BFSM.

3.1 Motion Representation

We represent characters using a standard hierarchical skeleton format. A pose of a character is represented as a hierarchy of joints, each joint having three degrees of rotational freedom (Figure 3.1). Formally, a motion is defined as a continuous multi-dimensional function $\mathbf{m}(t) = (\mathbf{p}_0(t), \mathbf{q}_0(t), \mathbf{q}_{1\ldots n}(t))$ varying with time t, where $\mathbf{p}_0(t)$ and $\mathbf{q}_0(t)$ represent the global position and orientation of the root node respectively, and $\mathbf{q}_i(t)$ is the orientation of the i^{th} joint in its parent coordinate system. Orientations are represented with quaternions in our work.

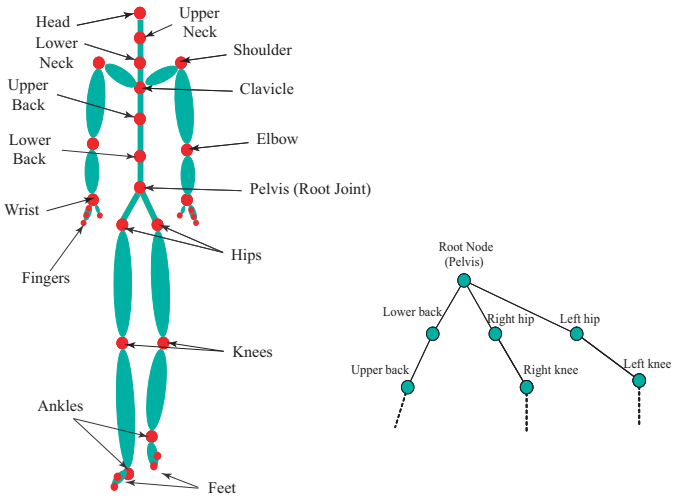


Figure 3.1: *An example articulated figure with 23 joints and its hierarchical representation. The joints have offsets and orientation with respect to their parent. The root node has a translation and orientation, that defines the position and orientation of the character.*

For the task of automatically mining the structure of the BFSM, we assume that we are provided with a set of K motion clips $\mathcal{M} = \{M_1, M_2, ...M_K\}$. Each clip M_i is an ordered collection of k_i frames discretely sampled over time from the corresponding motion function $\mathbf{m_i}(t)$. i.e. $M_i = \{\mathbf{m_i}(t_1), \mathbf{m_i}(t_2), ...\mathbf{m_i}(t_{k_i})\}$. For the sake of clarity, we denote a frame $\mathbf{m_i}(t_j)$ as $\mathbf{f}_j^i$, so that $M_i = \{\mathbf{f}_1^i, \mathbf{f}_2^i, ...\mathbf{f}_{k_i}^i\}$.

We also assume that each clip in $\mathcal{M}$ is assigned a unique label, using a labeling function $\mathcal{L}(.)$, from a set of N labels $\{S_1, S_2, ...S_N\}$ (e.g. running, walking, stopping, starting). We assert that $\mathcal{L}(M_i) = S_j \Leftrightarrow \mathcal{L}(\mathbf{f}_k^i) = S_j, \forall \mathbf{f}_k^i \in M_i$. In other words, a motion clip label is also applicable to all the frames of the clip.

3.2 Motion Graph Mining

The labels $\{S_1, S_2, ...S_N\}$ defined above denote "behavior states" and will correspond directly to the nodes in our high-level BFSM graph, $\mathcal{B}$ (Figure 3.3). Given a set of labeled clips $\mathcal{M}$ and the nodes in $\mathcal{B}$, our goal is to automatically find directed edges and their weights in $\mathcal{B}$ as depicted in Figure 3.3.

3.2.1 Motion Graph

We start by briefly describing our process of building a motion graph, $\mathcal{G}$, using the frames in $\mathcal{M}$ [26, 2, 31]. A key observation that motivates our approach is that $\mathcal{G}$ represents the space of possible transitions given $\mathcal{M}$. Some of these may be "new" transitions that were not actually captured in the original sequences. We will exploit this information to determine the transitions and their probabilities in $\mathcal{B}$.

Let $\mathbf{f_i}$ and $\mathbf{f_j}$ denote any two arbitrary frames from arbitrary sequences in $\mathcal{M}$. We use the approach presented by Lee et al. [31] to measure the difference, $d(\mathbf{f_i}, \mathbf{f_j})$, between frames $\mathbf{f_i}$ and $\mathbf{f_j}$ as

$$d(\mathbf{f_i}, \mathbf{f_j}) = \omega \|V_j - V_i\| + \sum_{k=0}^{N} w_k \|log(q_{j,k}^{-1} q_{i,k})\| \tag{3.1}$$

where $q_{i,k}$ is the k^{th} joint angle quaternion at frame $\mathbf{f_i}$, w_k is the weight for that

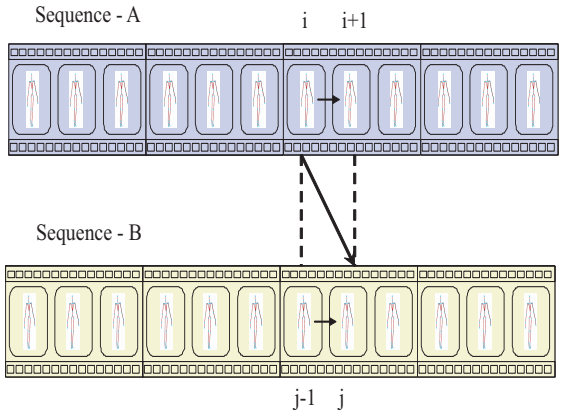


Figure 3.2: *The transition from a current pose in sequence A to a target pose in sequence B will only be smooth when the next pose in A is similar to the target pose in B and the previous pose for the target in B is similar to the current pose in A.*

particular joint and ω is a weighting for the term that measures root-node velocity difference. We use a joint weighting scheme suggested by Jin Wang and Bobby Bodenheimer [66].

To retain the dynamics of the motion when making transitions, we take an approach similar to Kovar et al. [26]. For a transition from one frame $\mathbf{f_i}$ to another frame $\mathbf{f_j}$ to appear smooth, both $d(\mathbf{f_i}, \mathbf{f_{j-1}})$ and $d(\mathbf{f_{i+1}}, \mathbf{f_j})$ should be small (Figure 3.2). Thus, we define the total cost to transition from frame $\mathbf{f_i}$ to frame $\mathbf{f_j}$ in terms of the differences between its neighboring frames.

$$D(\mathbf{f}_i, \mathbf{f}_j) = 0.5 * d(\mathbf{f_{i+1}}, \mathbf{f_j}) + 0.5 * d(\mathbf{f_i}, \mathbf{f_{j-1}}) \tag{3.2}$$

The resulting transition cost matrix is a weighted directed graph, $\mathcal{G}$, where the nodes are the frames, and the edges and their weights represent valid transitions and transition costs. As suggested by Lee *et al.* [31], this preliminary graph is pruned based on two rules. The first rule prunes out high cost transitions to reduce storage requirements and improve the quality of transitions. Pruning may introduce dead ends in the graph. The second rule eliminates dead ends by computing the largest strongly connected component of the graph using Tarjan's algorithm [62]. The remaining components are removed resulting in a single graph where any node can be reached from any other node.

3.2.2 Q-Learning

The utility of the above constructed motion-graph, $\mathcal{G}$, to generate natural-looking motion clips has been demonstrated by several researchers [26, 2, 31]. An optimal path between any two frames $(\mathbf{f}_i, \mathbf{f}_j)$ in $\mathcal{G}$ represents a smooth motion between the two frames and may include a transition not present in the original data-set $\mathcal{M}$.

To analyze the space of optimal transitions in $\mathcal{G}$, we run a Q-learning algorithm over $\mathcal{G}$ [61]. The goal is to learn a transition function $Q(\mathbf{f}_i, \mathbf{f}_j)$ given a frame transition cost $D(\mathbf{f}_i, \mathbf{f}_j)$ for a transition $\mathbf{f}_i \rightarrow \mathbf{f}_j$ in $\mathcal{G}$. $Q(\mathbf{f}_i, \mathbf{f}_j)$ scores an $\mathbf{f}_i \rightarrow \mathbf{f}_j$ transition with the objective of minimizing the "total discounted transition cost" of a motion sequence starting from frame $\mathbf{f}_i$, over an infinite horizon. We start by setting $Q(\mathbf{f}_i, \mathbf{f}_j) = D(\mathbf{f}_i, \mathbf{f}_j)$. We then iterate over every $\mathbf{f}_i \rightarrow \mathbf{f}_j$ edge in $\mathcal{G}$ and

update $Q(\mathbf{f}_i, \mathbf{f}_j)$ according to the following equation:

$$Q(\mathbf{f}_i, \mathbf{f}_j) \longleftarrow D(\mathbf{f}_i, \mathbf{f}_j) + \gamma * \min_{\forall \mathbf{f}_k \in \mathcal{N}(\mathbf{f}_j)} \{Q(\mathbf{f}_j, \mathbf{f}_k)\} \tag{3.3}$$

Here, $\gamma < 1.0$ denotes a discount factor that ensures convergence and $\mathcal{N}(\mathbf{f}_j)$ denotes a set of frames that are directly reachable from $\mathbf{f}_j$. We then replace each $D(\mathbf{f}_i, \mathbf{f}_j)$ in $\mathcal{G}$ with $Q(\mathbf{f}_i, \mathbf{f}_j)$, and call this new graph $\mathcal{Q}$.

Any path generated from $\mathcal{Q}$ by always picking a transition with the lowest Q-value at every frame will be a visually smooth sequence of frames. However, we have experienced in practice that the lowest Q-value differs from the second lowest Q-value by less than 1% of their values. This motivates us with the following scheme to expand the space of transitions represented by the graph, without restricting ourselves to the most optimal transitions. We allow transitions along edges having sub-optimal Q-values within Δ percentage of the optimal Q-value. A sub-optimal motion graph transition does not mean that the transition does not make logical sense, only that the motion does not support a smooth transition as defined by some transition threshold.

To allow sub-optimal transitions, we prune $\mathcal{Q}$, by computing the following at each frame $\mathbf{f}_i$ of the graph:

$$\begin{aligned} Q^i_{min} &= \min_{\forall \mathbf{f}_j \in \mathcal{N}(\mathbf{f}_i)} \{Q(\mathbf{f}_i, \mathbf{f}_j)\} \\ Q^i_{max} &= \max_{\forall \mathbf{f}_j \in \mathcal{N}(\mathbf{f}_i)} \{Q(\mathbf{f}_i, \mathbf{f}_j)\} \\ \Delta Q_i &= Q^i_{max} - Q^i_{min} \end{aligned} \tag{3.4}$$

We then remove all transition edges $\mathbf{f}_i \rightarrow \mathbf{f}_j$ such that

$$Q(\mathbf{f}_i, \mathbf{f}_j) > Q^i_{min} + \delta * \Delta Q_i$$

In all of our examples, we use $\delta = 0.70$.

3.2.3 Discovering BFSM Structure

It may be recalled from Section 3.1 that each clip in $\mathcal{M}$ is assigned a unique label from a set of N labels $\{S_1, S_2, ...S_N\}$ that represent the behavior states in a BFSM. We now construct an $N \times N$ matrix κ. The row and column headings of κ correspond to the labels $\{S_1, S_2, ...S_N\}$. Each entry $\kappa(S_p, S_q)$ is determined as follows. If $S_p = S_q$, then we set $\kappa(S_p, S_q) = 0$. This represents a scenario when a character can transition back into the same behavior (i.e. walking, running etc.). We require the user to directly identify such transitions. On the other hand, when $S_p \neq S_q$, $\kappa(S_p, S_q) = n$ where n is the number of edges of the form $\mathbf{f}_i \rightarrow \mathbf{f}_j$ in $\mathcal{Q}$ such that $\mathcal{L}(\mathbf{f}_i) = S_p$ and $\mathcal{L}(\mathbf{f}_j) = S_q$. It is important to note that some of the entries of the matrix κ could be zeroes and indicates the absence of transitions between the corresponding labels in $\mathcal{Q}$.

At this point, the matrix κ is an estimate of the frequency of transitions between different behaviors as observed in $\mathcal{Q}$. The estimates could be biased because some labels may be more prevalent than others in the original data. For example, if $\mathcal{M}$

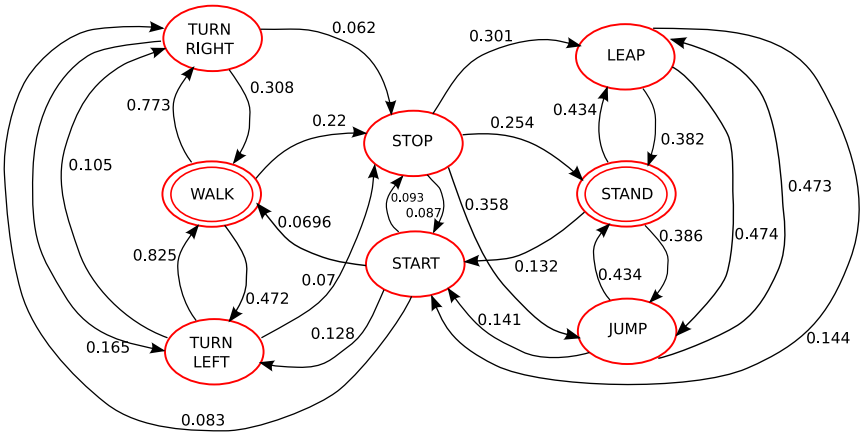


Figure 3.3: *An automatically discovered BFSM for a small locomotion dataset. Comparing with Figure 3.4, you'll notice that the mining algorithm discovered all of the appropriate transitions and assigned reasonable edge weights. States with double circles denote a cyclic behavior.*

contains many more walks than jumps, then the chances are higher that transitions from walks to other behavior states may dominate over jumps due to the bias. We account for this bias by updating the estimate $\kappa(S_p, S_q)$ as follows:

$$\kappa(S_p, S_q) \leftarrow \frac{\kappa(S_p, S_q)}{\eta(S_p) * \eta(S_q)} \tag{3.5}$$

where $\eta(S_p)$ represents the number of frames in $\mathcal{M}$ with label S_p. The denominator on the right side in the above equation represents an estimate of the total number of possible transitions between behaviors S_p and S_q, and thus normalizes $\kappa(S_p, S_q)$.

An additional pruning step sets entries of κ below a certain threshold (0.01 for all examples in our implementation) to zero. Each row (across columns) of the matrix is then normalized to sum to one, making the entries of κ valid probabilities.

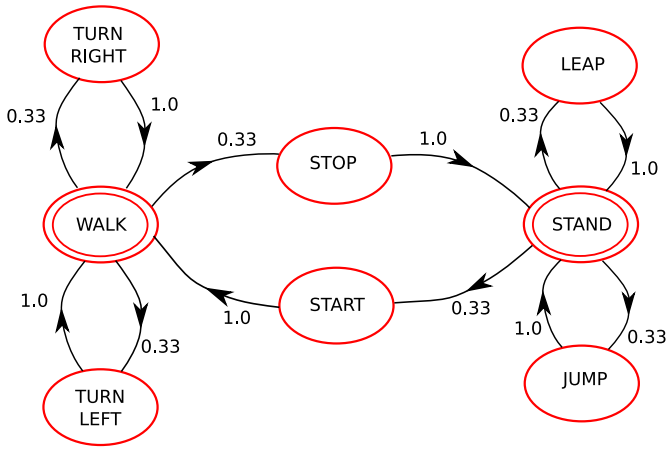


Figure 3.4: *A manually designed BFSM for the same small locomotion dataset. Notice that the manually designed graph is a subgraph of the automatically discovered BFSM but with different edge weights. All demos for this dissertation were generated from an automatically discovered graph with minor edits by a designer.*

Since the labels $\{S_1, S_2, ...S_N\}$ directly correspond to nodes in $\mathcal{B}$, each non-zero entry $\kappa(S_p, S_q)$ induces a directed edge $S_p \rightarrow S_q$ in $\mathcal{B}$, with a transition probability $\kappa(S_p, S_q)$. Figure 3.3 shows an example of such a graph $\mathcal{B}$ that was constructed by following the above steps for a set of 10 motion clips with labels shown in the figure. Comparing this graph to a manually designed BFSM, $\mathcal{B}_{manual}$, for the same dataset (Figure 3.4), we find that our mining approach discovered more connections compared to the manually designed graph. In Chapter 5, we shall see how the discovered graph is augmented with data-structures to synthesize controllable motion.

The most computationally expensive step in the above approach is our unoptimized Q-learning implementation. In the worst-case (i.e. a fully-connected graph), each iteration is $O(n^3)$ where n is the number of nodes in the motion graph. This step

can be optimized to $O(n^2 \log n)$ by using a heap to retrieve the edge with the minimum Q-value, in Equation 3.3.

3.2.4 Evaluating BFSM Structure

We compare the quality of a 'discovered' Behavior Finite State Machine $\mathcal{B}$ to a manual graph $\mathcal{B}_{manual}$ using two simple metrics. We define a '*connectivity factor*' (χ), that indicates the ratio of number of edges that were discovered in $\mathcal{B}$ to the number of edges in $\mathcal{B}_{manual}$ (Equation 3.6).

$$\chi = \frac{\mathbf{E}(\mathcal{B})}{\mathbf{E}(\mathcal{B}_{manual})} \tag{3.6}$$

where $\mathbf{E}(\mathcal{B})$ denotes the number of edges in the graph $\mathcal{B}$. The higher the value of χ, the more connected $\mathcal{B}$ is compared to $\mathcal{B}_{manual}$. Higher connectivity results in more responsive transitions between behaviors, and hence a more controllable character. More specifically, $\chi > 1.0$ implies that more edges were discovered by the mining approach, as compared to the edges manually designed by a user.

It is reasonable to expect the automated mining algorithm to reproduce the structure, at the very minimum, that the user designs while hand-constructing the BFSM. To this effect, we also define a '*reproducibility index*' (ρ) that measures the

ability of $\mathcal{B}$ to 'reproduce' transitions in $\mathcal{B}_{manual}$ (Equation 3.7).

$$\rho = \frac{\mathbf{E}(\mathcal{B} \cap \mathcal{B}_{manual})}{\mathbf{E}(\mathcal{B}_{manual})} \tag{3.7}$$

where $\mathcal{B} \cap \mathcal{B}_{manual}$ is a graph having edges that are common to $\mathcal{B}$ and $\mathcal{B}_{manual}$ (ignoring the edge weights). A value of $\rho = 1.0$ indicates $\mathcal{B}_{manual} \subseteq \mathcal{B}$, and every transition in $\mathcal{B}_{manual}$ is also present in $\mathcal{B}$.

While χ quantifies how good $\mathcal{B}$ is compared to $\mathcal{B}_{manual}$, ρ measures how faithfully user-designed transitions can be reproduced using the automatic approach. For the graphs in Figure 3.3 and 3.4, $\chi = 2.16$ and $\rho = 1.0$, indicating that our mining approach came up with a more connected graph structure that results in more responsive transitions.

Chapter 4 – Editing Behavior Finite State Machines

Automating the process of creating motion transition graphs has recently gained the attention of many researchers, as is evident from Chapter 2, and from our own mining approach described in Chapter 3. Such methods will eventually allow the user to quickly create complex motion transition graphs for character behaviors. However, a user may still need to edit and manipulate these high level behavior graphs. As a result, we forsee a need for tools to intuitively manipulate the structure of motion transition graphs. While simple operations such as adding or removing edges may be trivial, other type of modifications may not necessarily be as straightforward. In this chapter, we present an approach for editing the transition probabilities on the BFSM graph, based on a subjective criteria provided by the user.

Our solution for intuitive editing of a motion transition graph is based on the following observations:

- The transition probabilities in the BFSM encode the cost of transition between any two behaviors. i.e. If the transition probability between two behaviors i and j is p_{ij}, then the cost of transition can be calculated as $-\log p_{ij}$.

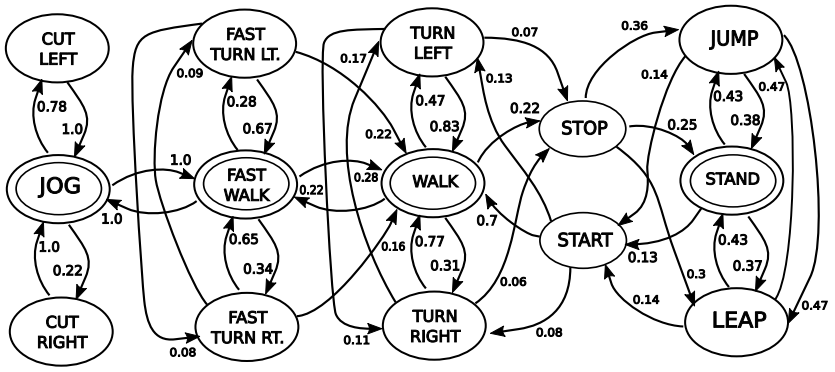


Figure 4.1: *A BFSM with 14 behaviors. The edges for this BFSM were automatically discovered by our mining algorithm described in Chapter 3*

- The weighted BFSM should encode a natural sequence of behavior transitions between any two behaviors. A natural sequence of behavior transitions between any two behaviors will correspond to an optimal path (or a shortest path) between those two behaviors in the graph.

- The end user that is using the BFSM graph is an expert at subjectively evaluating "natural looking" behavior transitions.

It is often the case that the end user's assessment of a sequence of natural looking transitions is complex and different from optimal paths that result from automated methods such as the one described in Chapter 3. While it is easy for a user to observe and identify behavior transitions that are natural looking, the sampled graph may not necessarily encode the most natural looking transitions as optimal paths in the graph. For example, Figure 4.1 shows a BFSM with 14 states constructed using our mining approach. Consider the optimal path between Stand

and Turn-Left behaviors. A user might consider (Stand : Start : Turn-Left) to be a more natural behavior than (Stand : Start : Walk : Turn-Left), which is optimal according to the BFSM in Figure 4.1.

Thus, there is a need for mechanisms that allow the user to change the optimal transition path between two behaviors in accordance with what the user specifies. One option would be to manually change the weights on the graph so that the user-specified path becomes the optimal path. This is not desirable, as it is not clear how other optimal paths in the graph change with local changes in weight. However, it would be more useful to have the user specify a sequence of transitions that looks natural according to some criteria. The graph can then be automatically optimized to change the relevant edge weights, without significantly affecting other parts of the graph. The result will be a graph with new weights such that the specified path is now the optimal path. This is an instance of the *Inverse Shortest Paths* problem.

In the following section, we formally state the Inverse Shortest Path problem as an optimization problem. We present a heuristics-based formulation that is applicable to our domain and compare it against a more general (and optimal) solution based on the Goldfarb-Idnani method for convex quadratic programming, presented by Burton et al. [9].

4.1 Inverse Shortest Path

Given a directed graph and a set of non-negative costs on its edges, the classical Inverse Shortest Path (ISP) problem seeks to modify these costs by as little as possible to ensure that a given path between two vertices in the graph is the new shortest path between those vertices [9].

Suppose $\bar{G} = (V, E, \bar{\mathbf{c}})$ is a weighted graph with n vertices and m edges. Let $V = \{v_k\}_{k=1}^{n}$ represent the set of vertices and $E = \{e_j\}_{j=1}^{m}$ be the set of edges. Each edge may also be represented as $e_j = (v_{s(j)}, v_{t(j)})$, where $s(j)$ is the index of the initial or source vertex of e_j and $t(j)$ is the index of its final or target vertex. $\bar{\mathbf{c}} = \{\bar{c}_j\}_{j=1}^{m}$, is a vector of non-negative costs associated with the edges.

Suppose a user specifies a path $\mathcal{P}_u$ in $\bar{G}$ such that $\mathcal{P}_u = (e_{j_1}, ...e_{j_l})$, where l is the number of edges in $\mathcal{P}_u$. The problem then is to determine $\mathbf{c}$, a new vector of edge costs such that

$$\min_{\mathbf{c}} ||\mathbf{c} - \bar{\mathbf{c}}|| \tag{4.1}$$

is achieved, under the constraints that

$$c_j > 0 \quad (j = 1, 2, ..., m) \tag{4.2}$$

and $\mathcal{P}_u = (e_{j_1}, ...e_{j_l})$ is the new shortest path in the graph $G = (V, E, \mathbf{c})$. Burton et al. use the above formulation (ISP) and a quadratic programming solver to achieve the given constraints [9].

It may be noted that the objective function in Equation 4.1 takes into account the cost vector for the entire graph. In other words, the solution method presented by Burton et al. allows the possibility of modifying the weights of edges that are not directly related to $\mathcal{P}_u$. However, while editing a BFSM, a user may prefer to "localize" the changes in edge weights to edges in $\mathcal{P}_u$. In addition, the user may have other constraints such as: minimize the number of edge weights that are modified, or already existing optimal paths should not be affected.

Keeping these requirements in mind, we formulate an alternate objective function and optimize using a Linear Least Squares approach [24]. Unlike the formulation above, our approach is more localized in the sense that it only modifies the cost on the edges along $\mathcal{P}_u$.

Let $\mathcal{S}$ represent the actual shortest path between vertices $v_{s(j_1)}$ and $v_{t(j_l)}$ in the path $\mathcal{P}_u$. Let $\bar{\mathbf{c}}(\mathcal{P}_u)$ and $\bar{\mathbf{c}}(\mathcal{S})$ each represent a vector of edge-costs along the paths $\mathcal{P}_u$ and $\mathcal{S}$ respectively. Then, the modified objective function can be stated as:

$$\min_{\mathbf{c}(\mathcal{P}_u)} ||\mathbf{c}(\mathcal{P}_u) - \bar{\mathbf{c}}(\mathcal{P}_u)|| \tag{4.3}$$

subject to the following constraints

$$\begin{aligned} & c_j(\mathcal{P}_u) > 0 \quad (j = 1, 2, ..., l) \\ & |\mathbf{c}(\mathcal{P}_u)| < |\bar{\mathbf{c}}(\mathcal{S})| \end{aligned} \tag{4.4}$$

It can be noted that the above formulation (LLS) seeks to only change the weights

on the edges of the user-specified path $\mathcal{P}_u = (e_{j_1}, ...e_{j_l})$. For a detailed discussion of the implementation of the problem as a linear least squares optimization, please see [24].

In the next section, we compare the performance of the above two approaches for the inverse shortest path problem, as applicable to our domain.

4.2 Results

The LLS and ISP based quadratic optimization were both implemented in Matlab [24]. The user can load a BFSM through a user-interface, and have the option of applying either of the above mentioned optimization methods to achieve the desired changes to the graph.

In order to compare the Quadratic optimization, and the LLS approach mentioned above, we randomly generate graphs of different sizes and densities. For each generated graph, we arbitrarily pick a number of pairs of vertices, and solve the inverse shortest path problem using each of the approaches. For each solution returned, we measure the following:

- *Edges Changed*: The number of edges whose weights have been modified. It is desirable to have a minimum number of edges affected during the optimization process.

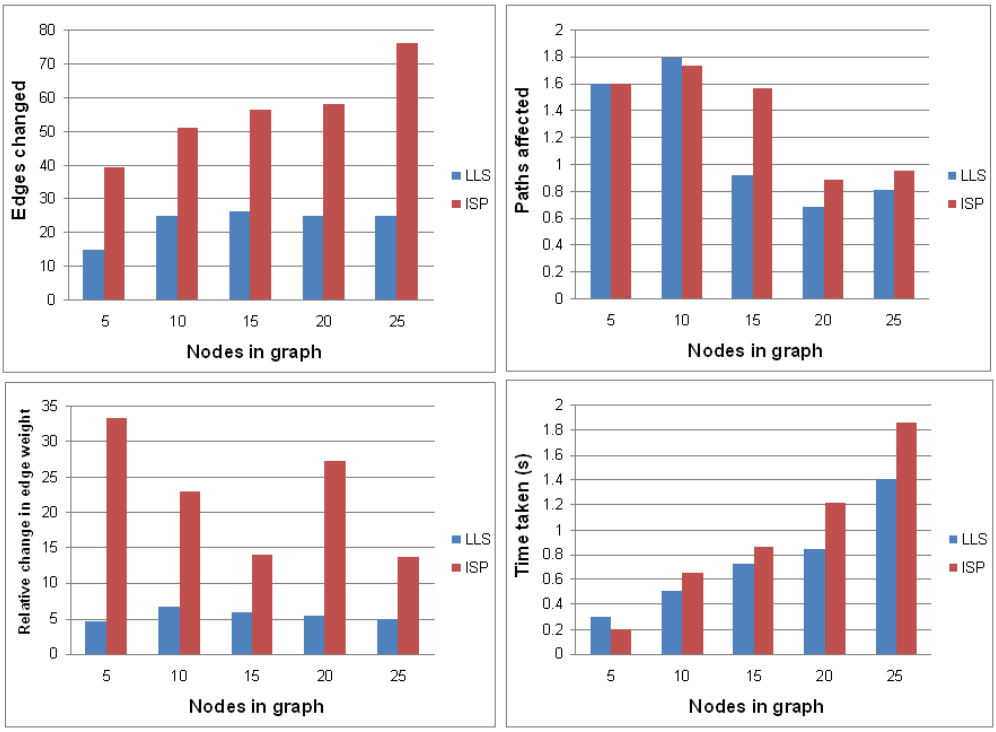


Figure 4.2: *Comparing LLS and ISP with increasing graph sizes. LLS consistently performs better in most cases.*

- *Paths Affected*: Number of existing optimal paths that have been changed. Again, it is desirable to minimize the changes in optimal paths in the BFSM graph.

- *Relative Change in Edge Weight*: Relative change in edge weights after the optimization step.

- *Time Taken*: Time taken to calculate the solution.

Figure 4.2 compares the performance of LLS and ISP with different graph sizes.

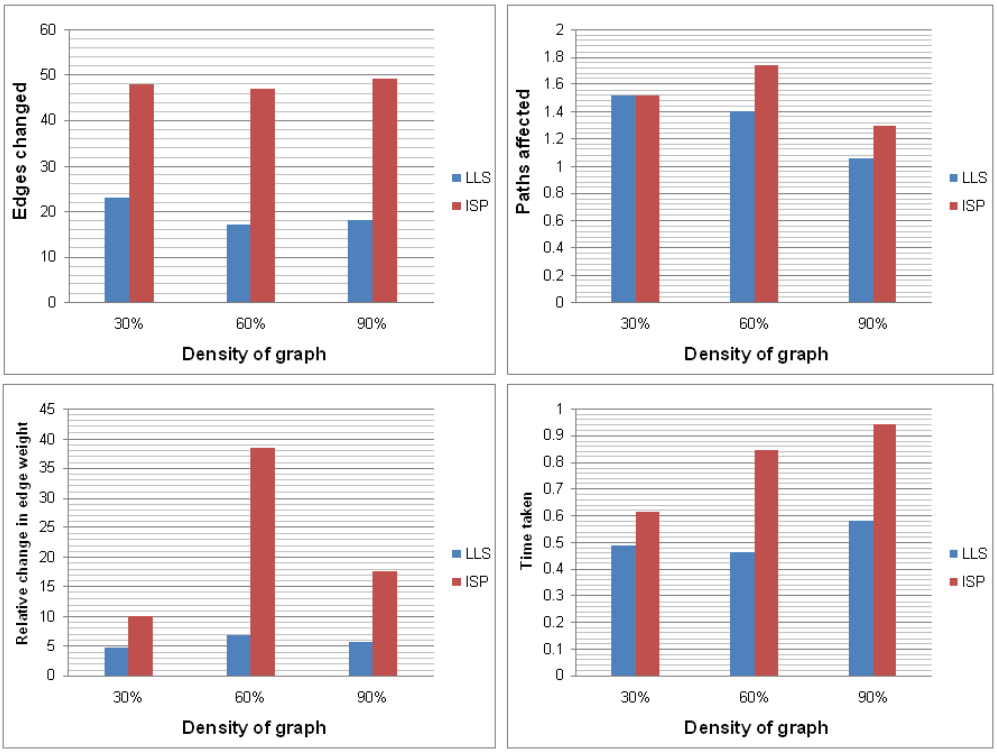


Figure 4.3: *Comparing LLS and ISP with different graph densities.*

The LLS formulation consistently performs better than ISP, in terms of the number of edges affected , and the change in weights. In most cases, the number of optimal paths affected is also fewer, as compared to the ISP formulation. It is also interesting to note that when compared to LLS, ISP affects more edges on larger graphs, than on smaller graphs. This is due to the fact that in the worst case, LLS can modify only the weights on the user-specified path $\mathcal{P}_u$. Figure 4.3 compares the performance of LLS and ISP with different graph densities. We measure the graph density for a graph with n vertices and m edges as $\frac{m}{n*(n-1)}$. It can be clearly seen that LLS performs much better than ISP. Fewer edges and optimal paths are

affected, and there is a smaller relative change in edge weights. From the above figures, it is also clear that LLS is also more efficient, in terms of computation time.

4.3 Conclusion

In the above sections, we make an attempt to address an important issue of providing an intuitive approach to edit an automatically generated BFSM. To this effect, we provide the end-user with a method to easily incorporate subjective evaluations of behavior transitions into the BFSM. The transition probabilities in the BFSM can be tuned using few high-level inputs from the user. We formulate the problem as an instance of the Inverse Shortest Path problem. We provide a "relaxed" version of the problem using linear least squares approximation, and compare it with the optimal quadratic programming approach presented by Burton et al. [9].

Our formulation of the Inverse Shortest Path problem as LLS works well in most of the scenarios as shown by the above results. However, it is not guaranteed to provide optimal results. In our experience, in some cases where the length of the user-specified path $\mathcal{P}_u = (e_{j_1}, ...e_{j_l})$ between the vertices $v_{s(j_1)}$ and $v_{t(j_l)}$ is greater than the actual shortest path $\mathcal{S}$ between the same vertices, the Quadratic solver performed better than LLS. Our user-interface provides the option of running both the versions of the problem. Consequently, the user can choose to apply each of the optimization process, and select the one that provides a better result in terms

of edges modified, optimal paths changed, or relative change in weights.

Chapter 5 – Building Behavior States and Transitions

The states of a BFSM represent behaviors that can be performed by the character. These behaviors, together with transitions between them, define the motion capabilities of the character being animated. While some behaviors may consist of a single motion clip (i.e Stand etc.), others may represent a continuous space of motion-clips parameterized by some relevant parameter. For example, the Walk behavior may be parameterized by the degree of turn. Some behaviors like "Walking" may generate a continuous stream of motion, while others, like "Stopping", may have a fixed duration. A BFSM should be capable of generating motion from these behaviors and handle transitions between them.

In the following sections, we discuss an approach to classify behavior states. Our classification aids us in formulating data structures necessary to represent different types of behaviors. We then describe how these data-structures are used to synthesize motion within and transitions between different behaviors.

Table 5.1: *Taxonomy of Behavior States and their Requirements*

	Registration Curve	**Self Warp**	**Transition Warp**
Single Sequence-Cyclic		✔	✔
Single Sequence-Acyclic			✔
Multi-Sequence-Cyclic	✔	✔	✔
Multi-Sequence-Acyclic	✔		✔

5.1 Behavior State Taxonomy

Recall that the nodes of the graph in Figure 3.3 represent behaviors and the edges with assigned probabilities represent valid transitions and their likelihoods. Behaviors like walk, run etc. may consist of multiple motion examples that vary according to some parameter such as turning angle, speed etc. We classify these behavior states as *multi-sequence* behaviors. Other behaviors like cut-right, jump etc. may consist of only a single motion clip. We call these behaviors *single-sequence* behaviors. In addition, behaviors like walking, running, standing etc. can be used to create motion cycles, while other behaviors like start walking, stopping etc. are not cyclic in nature. Accordingly, we can also classify behavior states as being *cyclic* or *acyclic*. As a result, behavior states may require up to four components:

- **Labeled motion clips** that are representative of the behavior, and are used in generating the desired behavior.

- An **Alignment curve** that is used to align multiple motions in a multi-sequence behavior state. The alignment curve also facilitates motion interpolation, to form a continuous space of motion clips for that behavior.

- A **Self-warp curve** to aid in generating a continuous stream of motion for cyclic behavior states.

- A **Transition warp curve** to generate a transition from one behavior state to another.

Table 5.1 summarizes the various data-structures needed for different types of behaviors.

Our approach for motion generation builds upon the idea of Kovar et al.'s registration curves [27]. As a result, time-alignment of motion clips is a fundamental operation required while defining behaviors and transitions. In the following sections, we first present an interactive time-warping tool that allows a user to quickly align and define time-warps between clips. An approach for implementing behaviors and transitions between different behaviors is then presented.

5.2 Time-alignment Tool

Figure 5.1 shows our time-warping tool. The tool displays a cost-matrix between two motion clips computed using a standard pose-distance measure [31]. An op-

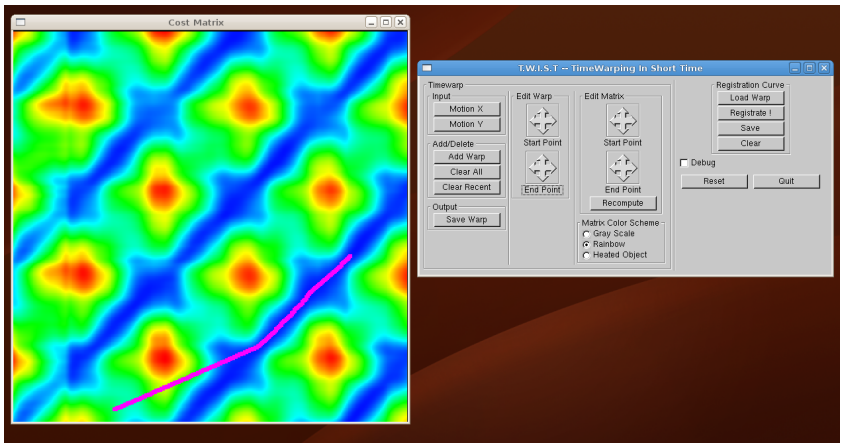


Figure 5.1: *Our interactive tool to construct time-warps between motion clips. The tool displays an optimal warp-path on top of a cost matrix between the motion clips. The matrix is color-coded so that blue represents regions of low cost, and red represents regions of high cost.*

timal time-warp between the clips is computed using dynamic time-warping techniques and is displayed as a line superimposed on the matrix [48, 6].

The primary goal of this tool is to automate the process of aligning motion clips and constructing time-warps. The user can specify a particular time-warp by interactively choosing a path in the matrix. The path defines a set of corresponding frames from each clip, and therefore aligns the motion clips in the best possible manner. The path is continuously updated in real-time as the user moves its starting location over the cost matrix. The user can then choose to anchor the start of the path and modify its length to produce the final warp. The user can also choose to define multiple time-warps between the motion clips.

5.3 Interpolated Behavior Space

Multiple motion clips may be included in a behavior state to create a parameterized space of the motions. For instance, a walking behavior could consist of sequences with varying degrees of left and right turning. The parameterized behavior state would then allow a user to control the amount of turning while walking. In this section, we describe the process of generating motion from a multi-sequence behavior state.

Following the notations introduced in Section 3.1, suppose a multi-sequence behavior $\mathbf{b}$ consists of k motion clips:

$$\mathbf{b} = \{\mathbf{m}_1^b(t), \mathbf{m}_2^b(t)..., \mathbf{m}_k^b(t)\}$$

We define a time-warp function $\mathbf{A}^b(u)$ for $\mathbf{b}$ that simultaneously aligns the time for each of its k motion-clips as $\mathbf{A}^b(u) = [A_1^b(u), A_2^b(u)..., A_k^b(u)]$, where $A_i^b(u) = t_i^b$ denotes the time position for motion $\mathbf{m}_i^b(t)$. We follow the approach used by Kovar et al. to construct $\mathbf{A}^b(u)$, where $u \in [0, 1]$ is a normalized global time parameter [27].

To describe the construction of $\mathbf{A}^b(u)$ in brief, we first pick a reference motion in $\mathbf{b}$ that is closest in terms of average time-warp cost, to all other motions in $\mathbf{b}$. In our notation, we assume $\mathbf{m}_1^b(t)$ denotes the reference motion. A time-warp between each motion in $\mathbf{b} - \{\mathbf{m}_1^b(t)\}$ and $\mathbf{m}_1^b(t)$ is then constructed and combined

to compute $\mathbf{A}^b(u)$. The tool shown in figure 5.1 is used to compute $\mathbf{A}^b(u)$. The user is required to construct each timewarp between $\mathbf{m}_1^b(t)$ and rest of the clips in $\mathbf{b}$. The tool automatically computes $\mathbf{A}^b(u)$ using the individual timewarps.

Evaluating the curve $\mathbf{A}^b(u)$ at some u gives a set of aligned time instants $\mathbf{A}^b(u) = [t_1^b, t_2^b..., t_k^b]$ for each of the motion clips in $\mathbf{b}$. Each of the k motion clips in $\mathbf{b}$ can then be evaluated at these time instants to result in a set of frames

$$\{\mathbf{m}_1^b(t_1^b), \mathbf{m}_2^b(t_2^b)..., \mathbf{m}_k^b(t_k^b)\}$$

In other words, given $\mathbf{b}$, $\mathbf{A}^b(u)$ and some u, we can define a function

$$\mathbf{F}_b(u) = [\mathbf{m}_1^b(t_1^b), \mathbf{m}_2^b(t_2^b)..., \mathbf{m}_k^b(t_k^b)]$$

that returns a vector of corresponding frames in $\mathbf{b}$, at time u.

Suppose we are given a k-dimensional vector of weights $\mathbf{w}^b = [w_1^b, w_2^b..., w_k^b]$ such that $w_i^b \in [0, 1]$, $\sum_i w_i^b = 1.0$ and w_i^b is the weight of clip $\mathbf{m}_i^b(t)$ in $\mathbf{b}$. Now, given $\mathbf{w}^b$ and a normalized global time u, we can compute an interpolated frame $\mathcal{F}_b(u, \mathbf{w^b})$ for behavior b as follows:

$$\mathcal{F}_b(u, \mathbf{w^b}) = \mathbf{w}^b(\mathbf{F}_b(u))^T \tag{5.1}$$

Using equation 5.1, one can compute $\mathcal{F}_b(u, \mathbf{w^b})$ at each normalized global time-instant u, as we advance along the curve $\mathbf{A}^b(u)$. The resulting sequence of frames

would represent an interpolated motion clip that is a weighted combination of all the clips in $\mathbf{b}$, according to a user-given weight vector $\mathbf{w}^b$. We advance along the curve $\mathbf{A}^b(u)$ using the voting scheme suggested by Kovar et al. in [27].

The weight vector $\mathbf{w}^b$ can also be mapped to intuitive parameters (turning angle, speed etc.) using nearest neighbor or scattered data interpolation [25, 43]. We use radial basis functions to map our parameters [7].

The alignment curve $\mathbf{A}^b(u)$ is implemented as a spline through the points:

$$[A_1^b(u), A_2^b(u)..., A_k^b(u)] \sim [t_1^b, t_2^b..., t_k^b]$$

As Kovar et al. note, each function $A_i^b(u) = t_i^b$ is invertible [27]. Consequently, one can compute an inverse function $(A_i^b)^{-1}(t_i^b) = u$ for each $A_i^b(u)$. This means that given a time-instant t_i^b in motion $\mathbf{m}_i^b(t)$, we can uniquely determine a point u on the alignment curve $\mathbf{A}^b(u)$. Of particular interest is the function

$$(A_1^b)^{-1}(t_1^b) = u \tag{5.2}$$

that relates a position in the reference motion $\mathbf{m}_1^b(t)$ to $\mathbf{A}^b(u)$. In the following sections, we describe the process of designing transitions between (possibly identical) behaviors. We shall use the property described by Equation 5.2 in synthesizing transitions between two multi-sequence behaviors.

As a special case, suppose $\mathbf{b}$ is a single-sequence behavior, i.e. $\mathbf{b} = \{\mathbf{m}_1^b(t)\}$. Then

$\mathbf{A}^b(u) = [t_1^b] = u * len(\mathbf{m}_1^b(t))$. $len(\mathbf{m}_1^b(t))$ represents the length of the clip $\mathbf{m}_1^b(t)$, and has the same units as t_1^b. The weight vector trivially becomes $\mathbf{w}^b = [1.0]$, and Equation 5.1 is simplified to

$$\mathcal{F}_b(u, 1.0) = 1.0 * \mathbf{F}_b(u) = \mathbf{m}_1^b(t_1^b) = \mathbf{m}_1^b(\mathbf{A}^b(u)) \tag{5.3}$$

Further, Equation 5.2 can also be simplified to

$$(A_1^b)^{-1}(t_1^b) = \frac{t_1^b}{len(\mathbf{m}_1^b(t))} \tag{5.4}$$

5.4 Transitions

In this section, we describe our process of synthesizing a transition from a source behavior **a** to a target behavior **b** in a BFSM. If **a** and **b** are single-sequence behaviors, computing a transition is straightforward [6, 68]. Once we time-align the two clips, a uniformly varying weighting function from 0 to 1.0 can be used to synthesize the transition between **a** and **b**.

However, in the most general case, **a** and **b** may both represent multi-sequence behaviors. In such scenarios, we are presented with the problem of computing a transition between two interpolated motions $\mathcal{F}_a(u^a, \mathbf{w^a})$ and $\mathcal{F}_b(u^b, \mathbf{w^b})$ on the fly (Equation 5.1).

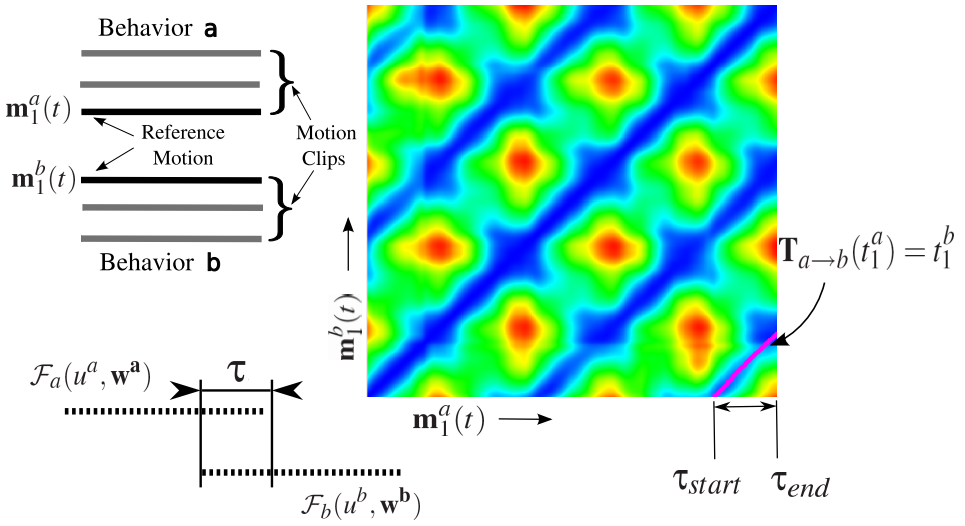


Figure 5.2: *Transition between multi-sequence behaviors* **a** *and* **b** *(top, left) is based on a time-warp defined between their reference motions (right). As we advance along $\mathcal{F}_a$ in the transition window, we can blend into $\mathcal{F}_b$.*

5.4.1 Transitions between Multi-sequence Behaviors

Consider two multi-sequence behaviors $\mathbf{a} = \{\mathbf{m}_1^a(t), \mathbf{m}_2^a(t)..., \mathbf{m}_{k_a}^a(t)\}$ and $\mathbf{b} = \{\mathbf{m}_1^b(t), \mathbf{m}_2^b(t)..., \mathbf{m}_{k_b}^b(t)\}$, with k_a and k_b motion clips respectively. Let $\mathbf{A}^a(u)$ and $\mathbf{A}^b(u)$ denote the alignment curves for each of these behaviors. A transition warp from **a** to **b** is constructed using the reference motions $\mathbf{m}_1^a(t)$ and $\mathbf{m}_1^b(t)$ respectively.

More specifically, we define $\mathbf{T}_{a\to b}(t_1^a)$ to be a time-warp function that maps a time instant t_1^a in $\mathbf{m}_1^a(t)$ to a time instant t_1^b in $\mathbf{m}_1^b(t)$. $\mathbf{T}_{a\to b}(t_1^a)$ is given by Equation 5.5.

$$\mathbf{T}_{a\to b}(t_1^a) = t_1^b, \quad t_1^a \in [\tau_{start}, \tau_{end}] \tag{5.5}$$

where $\mathbf{m}_1^a(t)$ and $\mathbf{m}_1^b(t)$ are reference motions for **a** and **b** respectively. Figure 5.2

shows an example of such a time-warp. The duration of transition, τ, is user-specified and $\tau \leq \tau_{end} - \tau_{start}$. $\mathbf{T}_{a \to b}(t_1^a)$ is easily constructed using our tool in Figure 5.1.

We now describe the process of synthesizing a transition from **a** to **b**. Let $\mathcal{F}_a(u^a, \mathbf{w^a})$ represent an interpolated frame for **a**, where u^a represents a normalized time-instant and $\mathbf{w^a}$ is the weight vector for clips in **a**. We know from Section 5.3 that $\mathbf{A}^a(u)$ can be evaluated at u^a to compute the corresponding time instant t_1^a in reference motion $\mathbf{m}_1^a(t)$. When $t_1^a \in [\tau_{start}, \tau_{end}]$, the function in Equation 5.5 is defined and we can compute the corresponding position t_1^b in reference motion $\mathbf{m}_1^b(t)$ for behavior **b**. Further, from Equation 5.2, t_1^b can be used to compute the corresponding position u^b on $\mathbf{A}^b(u)$ for behavior **b**

$$u^b = (A_1^b)^{-1}(t_1^b) \tag{5.6}$$

In this manner, given u^a, we can use Equations 5.5 and 5.6 to find the corresponding u^b. As a result, a corresponding interpolated frame $\mathcal{F}_b(u^b, \mathbf{w^b})$ in behavior **b** can be computed.

To summarize, as we playback interpolated frames $\mathcal{F}_a(u^a, \mathbf{w^a})$, we can compute corresponding interpolated frames $\mathcal{F}_b(u^b, \mathbf{w^b})$ in behavior **b** using the steps outlined above. We then synthesize a transition blend between the corresponding frames $\mathcal{F}_a(u^a, \mathbf{w^a})$ and $\mathcal{F}_b(u^b, \mathbf{w^b})$ using a uniformly varying weight over the transition

window $\tau \in [\tau_{start}, \tau_{end}]$ as follows:

$$\mathcal{F}_{ab}(u^a, w(t)) = (1 - w(t)) * \mathcal{F}(u^a, \mathbf{w^a}) + w(t) * \mathcal{F}(u^b, \mathbf{w^b}) \tag{5.7}$$

where $w(t)$ is an ease-in ease-out function uniformly varying from 0 to 1.0, as t varies across the transition window τ.

We assume a default set of weights $\mathbf{w^b}$ in the target space $\mathbf{b}$, while synthesizing the transition using Equation 5.7. While this may not be a reasonable assumption to make in every situation, it works well for our examples. Heck et al. address the problem of finding the right set of weights in the target space during a motion transition in their recent work [20].

Note that applying the definition of $\mathcal{F}(u^a, \mathbf{w^a})$ and $\mathcal{F}(u^b, \mathbf{w^b})$ from Equation 5.1 to Equation 5.7, $\mathcal{F}_{ab}(u^a, w(t))$ may alternatively be represented as

$$\mathcal{F}_{ab}(u^a, w(t)) = \begin{bmatrix} \tilde{\mathbf{w}}^a(t) & \tilde{\mathbf{w}}^b(t) \end{bmatrix} * \begin{bmatrix} (\mathbf{F}_a(u))^T \\ (\mathbf{F}_b(u))^T \end{bmatrix} \tag{5.8}$$

where $\tilde{\mathbf{w}}^a(t) = (1 - w(t)) * \mathbf{w}^a(t)$ and $\tilde{\mathbf{w}}^b(t) = w(t) * \mathbf{w}^b(t)$.

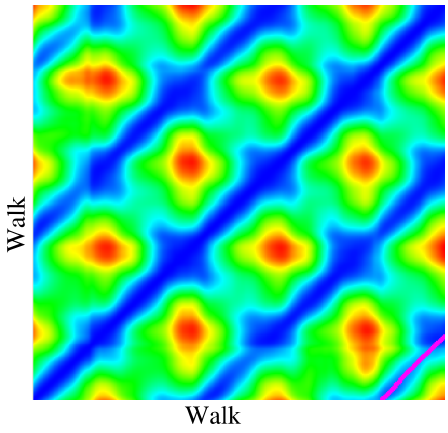


Figure 5.3: *Designing a self-transition for a Walk behavior. The background is a visualization of the cost matrix constructed using the reference motion for the Walk behavior. A self-transition is specified by a transition from the end of the reference clip to the beginning of the reference clip (i.e. lower-right corner of the cost-matrix).*

5.4.2 Self Transitions

Multi-sequence behaviors that are cyclic (eg. walking, running etc.) require a "self-transition" in order to synthesize a motion cycle. Self-transitions are a special case of a transition between multi-sequence behaviors. Our transition scheme described in Section 5.4.1 can be used for self-transitions without any modification by replacing the target behavior **b** with behavior **a**. Equation 5.5 would now represent a mapping from a time instant near the end of $\mathbf{m}_1^a(t)$ to a time instant at the beginning of $\mathbf{m}_1^a(t)$. Figure 5.3 shows an example of a self-transition designed by the user for the Walk behavior. The transition-warp is specified at the lower-right corner of the cost-matrix computed using the reference motion for the behavior.

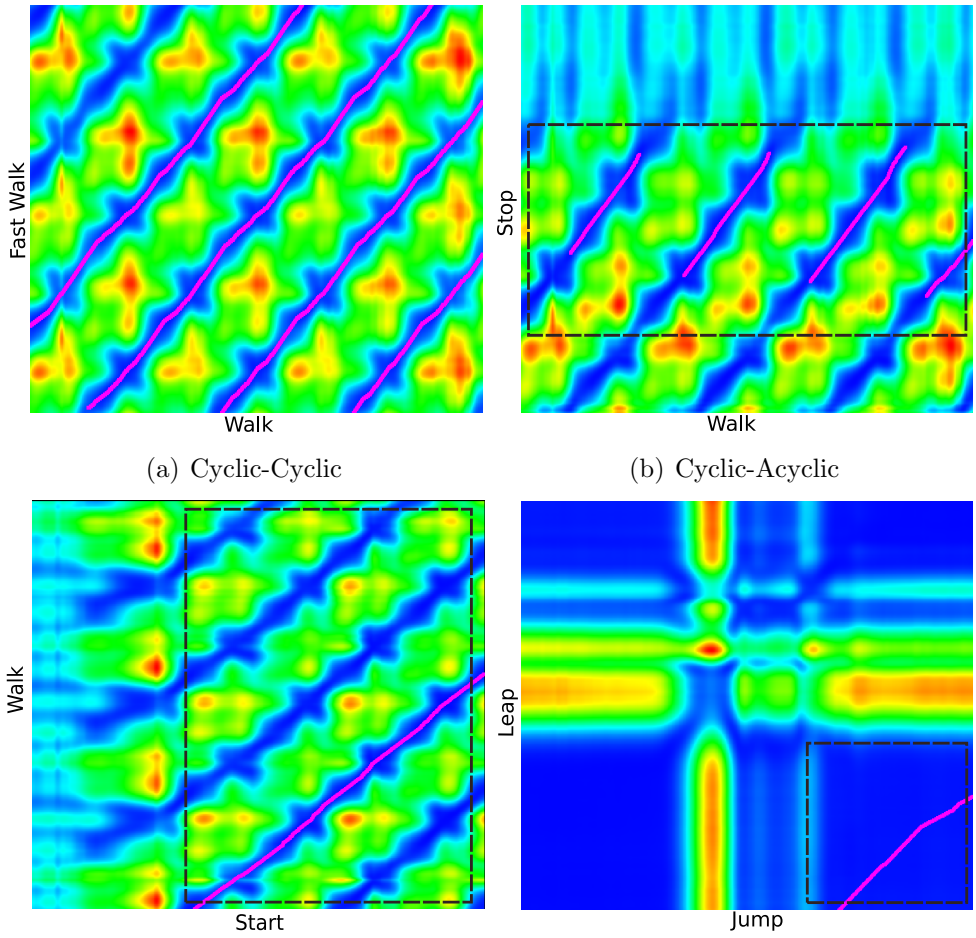
5.4.3 Multi-point Transitions

Most previous work to date has assumed short clips [43, 42], requiring that the end of a source motion match well to the beginning of the target motion, and that transitions occur at these precise locations. This results in quick and seamless transitions, but requires carefully crafted motion sequences. In addition, the motion may suffer from lack of variation. A walk with several steps will naturally contain more variation than a clip that includes a single cycle.

We support multi-point transitions to ease these restrictions (Figures 5.4(a) and 5.4(b)). Arbitrarily long sequences can be used as long as multiple transition warps are identified, requiring less 'fine tuning' of the individual motion sequences and only a small amount of additional specification in our alignment tool. Each of these warps correspond to a mapping defined by Equation 5.5, but over a different $[\tau_{start}, \tau_{end}]$ interval. Thus, even with long sequences that include natural variation, responsiveness to control input is not hindered when incorporating multi-point transitions.

5.4.4 Transition Design Guidelines

We found that the following guidelines were helpful in general to aid the design of transitions between various types of behavior states (cyclic, acyclic).



(a) Cyclic-Cyclic (b) Cyclic-Acyclic

(c) Acyclic-Cyclic (d) Acyclic-Acyclic

Figure 5.4: *The background of each image is a visualization of the cost matrix between the reference motions for each pair of behavior states. Blue represents regions of low cost while red represents regions of high cost. The magenta paths within the cost matrix are the designed timewarp curves. In (a) and (b) we see examples of multi-point transition warps between reference motions for the* ***walk*** *and* ***fast walk*** *behavior states and* ***walk*** *and* ***stop*** *behavior states respectively. Both (c) and (d) show single transition warps. The black boxes highlight the regions where the warps should be placed. The warps are designed by interactively anchoring the desired start position and modifying the warp length.*

- ***Cyclic-to-cyclic*** transitions should not constrain the character to transition at only the ends of the source and target sequences. Therefore, the user should design multiple transitions in order to provide the character with multiple opportunities to transition between the respective behavior states. Multi-point transitions ensure maximum responsiveness to user input as they guarantee that the motion generated from the source behavior can start transitioning into the target behavior state within a finite (generally small) delay.

  See Figure 5.4(a) for an example of a responsive transition between a *Walk* behavior state and a *Fast* Walk behavior state.

- ***Cyclic-to-acyclic*** transitions from a cyclic behavior state such as *walking* to an acyclic behavior state such as *coming to a stop* should allow the character to make the transition from any point in the cyclic motion to an acceptable location in the acyclic motion. The user is advised, therefore, to design transitions over the cost-matrix as shown in Figure 5.4(b). Here again, multi-point transitions are designed for a quick transition into the target acyclic state with a small delay. Notice that by designing the transition warp, the user also has control over the maximum duration of the target acyclic motion. In Figure 5.4(b) for instance, motion frames in the *Stop* behavior that do not correspond to any transition warps (i.e. the beginning portion of the Stop sequence) are discarded automatically. This results in an effectively shorter target motion clip, and a much quicker "Stopping"

response.

- ***Acyclic-to-cyclic*** transitions require a warp from the latter portion of the acyclic behavior to any location in the cyclic behavior. Because the source is acyclic, transition can occur only at the end and therefore multi-point transitions do not provide any advantage. Figure 5.4(c) shows an example where the user has specified a transition warp from the *Start* behavior to the *Walk* behavior.

- ***Acyclic-to-acyclic*** transitions should be allowed from the end of the source to the beginning of the target. This corresponds to the lower right region of the cost matrix as shown in Figure 5.4(d) where a transition warp is defined from a *Jump* (in-place) behavior to a *Leap* (forward) behavior. While the lower left portion represents a reasonable warp in terms of cost (the beginning of the two sequences is similar), it would not make logical sense to design a transition at this location.

The above guidelines allow a user to design a transition without requiring carefully crafted sequences for the source and target behaviors. It may be pointed out that the region shown in Figure 5.4(d) would be used in all the above cases, if the source and target behaviors were to be constructed with carefully crafted sequences that overlap at the end and the beginning respectively.

5.5 Interpolating Root Transformations

Equations 5.1 and 5.7 both represent the synthesis of an interpolated frame from a set of corresponding frames. It may be recalled from Section 3.1 that a frame of a motion $\mathbf{m}(t)$, at time t is given by a set of joint angles, and root-node transformations. Thus, the synthesis of a new frame from a set of frames involves interpolation of joint positions and root-node transformations in some manner. Joint angles can be easily interpolated using spherical linear interpolation [58]. However, as Kovar et al. note [27], interpolating root-node positions and orientations needs careful attention.

For the sake of this discussion, suppose a set of k motions $\{\mathbf{m}_1(t), \mathbf{m}_2(t)..., \mathbf{m}_k(t)\}$ are required to be interpolated to synthesize a new motion $\mathbf{m}(t)$. For a motion $\mathbf{m}_i(t)$, let $p_i(t)$ and $R_i(t)$ represent the root-node position and orientation at time t, respectively.

In order to facilitate the synthesis of root-node transformations, $p(t)$ and $R(t)$, for $\mathbf{m}(t)$, we pre-process each of the k motion clips in $\{\mathbf{m}_1(t), \mathbf{m}_2(t)..., \mathbf{m}_k(t)\}$. More specifically, for each motion $\mathbf{m}_i(t)$, we precompute the relative transformations at each time-step t.

Thus at any time t, the relative position of the root-node, $\Delta p_i(t)$, with respect to the previous frame $t-1$, for motion $\mathbf{m}_i(t)$ is given by Equation 5.9

$$\Delta p_i(t) = R_i(t-1)^{-1} * (p_i(t) - p_i(t-1)) \tag{5.9}$$

Similarly, the relative orientation of the root-node at t, $\Delta R_i(t)$, with respect to the previous frame $t-1$ is given by Equation 5.10

$$\Delta R_i(t) = (R_i(t-1))^{-1} * R_i(t) \tag{5.10}$$

At run time, we synthesize $p(t)$ and $R(t)$, for $\mathbf{m}(t)$, as follows: Let $\mathbf{\Delta P}(t)$ represent a row-vector of relative position offsets for the given k motion clips, i.e.

$$\mathbf{\Delta P}(t) = \begin{bmatrix} \Delta p_1(t) & \Delta p_2(t) & ... & \Delta p_k(t) \end{bmatrix}$$

If $\mathbf{w}$ represents a vector of blend-weights for the given set of motions, then we calculate the weighted-average of the relative position change, $\Delta\bar{p}(t)$, at time t, as follows:

$$\Delta\bar{p}(t) = \mathbf{w} * (\mathbf{\Delta P}(t))^T \tag{5.11}$$

Given the root transformations, $p(t-1)$ and $R(t-1)$, for the frame $\mathbf{m}(t-1)$, the position of frame $\mathbf{m}(t)$ is then computed as

$$p(t) = p(t-1) + R(t-1) * \Delta\bar{p}(t) \tag{5.12}$$

We extend the same idea to compute the orientation of the root node. i.e. If $\Delta\bar{R}(t)$

is defined as

$$\Delta\bar{R}(t) = \left[\begin{array}{cccc} \Delta R_1(t) & \Delta R_2(t) & ... & \Delta R_k(t) \end{array} \right]$$

then we use spherical linear interpolation (in quaternion space) to compute a weighted-average of the relative orientation change, $\Delta\bar{R}(t)$, as follows:

$$\Delta\bar{R}(t) = slerp(\mathbf{w}, \mathbf{\Delta R}(t)) \tag{5.13}$$

The orientation $R(t)$ for motion frame $\mathbf{m}(t)$ is then calculated as

$$R(t) = R(t-1) * \Delta\bar{R}(t) \tag{5.14}$$

In summary, we preprocess motion-clips to extract relative transformation offsets at every frame. At run time, we interpolate these offsets to synthesize the correct root-node transformations for the interpolated frame. Once the character is initialized with a starting position, and orientation, successive frames can be drawn at the correct location with the correct orientation using Equations (5.12 , 5.14).

Chapter 6 – Motion Synthesis using BFSM

In Chapters 3-5, we presented a user-in-the-loop "semi-automated" process for designing a BFSM. In the following sections, we present the various ways in which the BFSM design can be exploited to synthesize character motion. Specifically, we present a run-time motion generation framework that leverages multi-point transitions between behavior states in our BFSM. We also present a scheme for incorporating global planning, similar to the one presented by Lau et al. [30]. We conclude with an approach for reactive character navigation with obstacle avoidance, using the BFSM.

6.1 Run-Time Motion Generation - The CFSM

Motion is generated at run-time by simply playing the motion represented by a behavior state. This motion may consist of a single sequence playback or a sequence synthesized from multiple motions through blending as discussed in Section 5.3. A behavior change request forces the computation of a path from the source behavior to the target behavior. Once this path is known, the character must go through a series of transitions that move from the source to the target through the intermediate behavior states. These transitions are generated as discussed in Section 5.4.

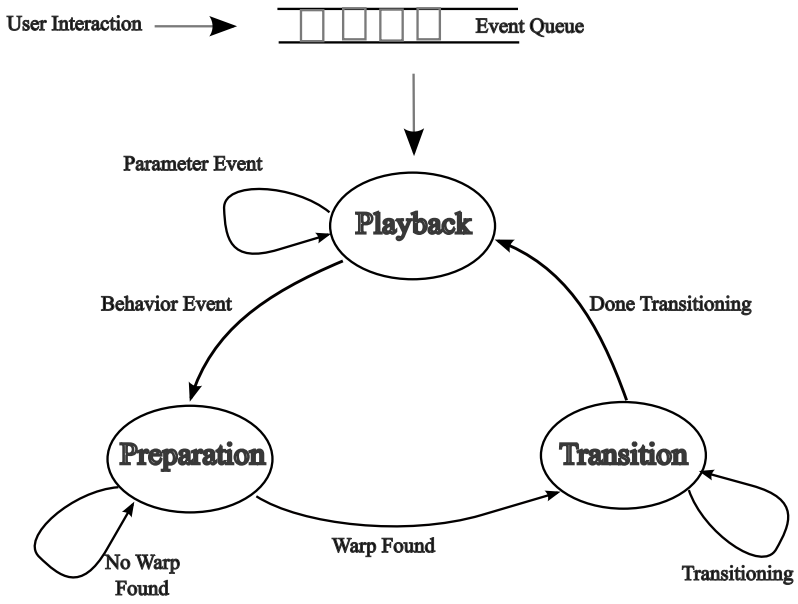


Figure 6.1: *A Control Finite State Machine consisting of three modes. Control inputs are recorded as events, and are stored in an event queue. The events are processed by the CFSM in the order they were requested.*

In this section, we will discuss how user requests are processed online to control parameterized behaviors and trigger transitions.

When in a behavior state, motion is generated by playing a single sequence, or, in the case of a multiple sequence behavior, using the approach in Section 5.3. Our multi-point transition scheme allows transitions between states starting from multiple locations. Triggering a transition upon request by the user requires careful attention with our scheme. For instance, consider a transition request from a multiple-sequence, cyclic walk behavior to a multiple-sequence, cyclic jog behavior.

The character could be in the middle of a self-transition in the walk behavior when the request was made. Or, the character could be beyond the last transition point of the walk behavior when the request was made, and the character can only transition when it cycles back to the beginning of the behavior.

Our approach is to use a control finite state machine (CFSM) to facilitate transitions (Figure 6.1). The states of this CFSM corresponds to the different states the character can be in, while synthesizing motion from the BFSM. For the sake of clarity, we distinguish the CFSM-states from the BFSM-states by calling the CFSM states "modes". We identify three such modes as shown in Figure 6.1.

The character starts in some behavior state of the BFSM. The CFSM starts in the *playback* mode. The event queue is initialized to be empty. All user requests are put into an event queue in the order they are received from the user. While in the *playback* mode, the motion for the current behavior state of the BFSM is played until an event is found on the event queue.

We identify and process three kinds of events: a parameter change event, a user behavior change event, and an internal behavior change event. A parameter change event (e.g. turn more to the right) causes an update to the parameters of the current behavior state in the BFSM and thus a change in the synthesized motion within that behavior. The CFSM continues in the playback mode, but the behavior parameters are updated to reflect a change in weights of motion clips that are currently being blended. If the behavior is cyclic, the character continues to remain

in the same behavior space.

When the character is in an acyclic behavior, then the question arises as to what happens when it reaches the end of the motion clip. We handle such situations by generating an internal behavior change event. In this case, the CFSM places a behavior state change event on the queue to move from the acyclic behavior state to a default cyclic behavior state. This default transition from an acyclic state is specified by the designer during the BFSM design phase.

A behavior state change event can also be directly specified by the user through a user interface. A behavior change event causes two things to happen. First, a shortest path is computed from the source behavior state to the target behavior state in the BFSM using the negative log likelihood for the cost function,

$$c_{ij} = -\log p_{ij} \tag{6.1}$$

where p_{ij} represents the probability of a transition between behaviors i and j. The resulting sequence of behavior states is added to the event queue as behavior state change events. For example if in the walking state, the user requests a cut right, the path from walking to cut right may include fast walking and jog between them. Therefore the behavior state change request to cut right is replaced with three requests: walk, jog, and cut right.

Upon processing a behavior change event, generated either internally or by the user, the CFSM transitions to the *preparation* mode. In this mode, playback of

the motion in the current behavior state continues until the next suitable transition point is reached (there may be multiple possible transitions due to our multi-point transition scheme) for the target behavior state. Once a transition point is reached, the CFSM moves to the *transition* mode, and handles the transitioning from the current behavior to the requested behavior over the user-defined time window. Once the transition is complete, the CFSM moves back to the *playback* mode and continues to process available events.

The scheme presented above ensures that transitions and parameter change requests are brought into effect immediately, depending on the underlying BFSM design. When the character is in a parameterized multiple sequence behavior state, the effect of a user-controlled parameter is immediately visible. In addition, the CFSM ensures a transition as soon as one is available, once a behavior change event has been issued. Our motion generation approach benefits from blends to produce continuous and controllable motion within a state. Additionally, since the BFSM essentially represents a coarse weighted motion graph, transitions between behavior states will occur as a natural looking sequence through possibly multiple behaviors.

6.2 Motion Planning

We also present a global planning scheme with a BFSM. The approach is similar to the one presented by Lau et al. [30], but has been adapted to our continuous

behavior spaces.

We summarize Lau et al.'s approach before we explain how we adapt their planner to work with our BFSM. The planner is based on A* search [53] that uses two related data-structures: a tree, "Tree", that records the explored states in the BFSM that are visited during the search; and a priority queue, "Queue", that orders a list of potential states to be explored during the next search iteration, according to a cost value. A node in the search tree stores an action a, a motion-clip, that was executed at that state, the position, orientation, time and cost.

The cost of each action, representing a single motion-clip, is computed as the distance the character's root position (projected on the floor) travels. The following information for each action a is computed automatically:

1. Relative change in character's root position and orientation $(x(a), y(a), \theta(a))$.

2. Change in time represented by the number of frames in the clip $t(a)$.

3. The cost of this action $cost(a)$

Thus the position, orientation and time at some node n of the tree represents the global position, orientation, and playback time for a character, if it follows a sequence of actions along a path from the root-node of the tree, to the node n. The total cost at node n is the sum of the cost of actions along this path and the expected cost to reach the goal (DistToGoal).

Algorithm 6.1: Lau et al.'s Original Behavior Planner [30]

```
Tree.Initialize(s_init);
Queue.Insert(s_init,DistToGoal(s_init,s_goal));
while !Queue.Empty() do
    s_best ← Queue.RemoveMin();
    if GoalReached(s_best,s_goal) then
        return s_best ;
    end
    e ← E(s_best.time);
    A ← F(s_best, e);
    foreach a ∈ A do
        s_next ← T(s_best, a) ;
        if G(s_next,s_best,e) then
            Tree.Expand(s_next,s_best) ;
            Queue.Insert(s_next,DistToGoal(s_next,s_goal)) ;
        end
    end
end
return no possible path found ;
```

The pseudocode for Lau's behavior planner is shown in Algorithm 1. The planner initializes the root of the tree with s_{init}, which represents the initial configuration of the character. The lowest cost node s_{best} from the queue is iteratively expanded until a solution is found, or queue is empty (i.e. no solution exists). If the character has reached the goal, then s_{best} will be equal to s_{goal} (within some tolerance ϵ), and the planner will return a sequence of actions (or motion clips) from s_{init} to s_{best}. The function F determines the set of actions A that the character can take from a particular state. This set is determined by the transitions in the BFSM. The function $E()$ updates the environment e according to the current global time. The transition function T computes the "next" state, s_{next} resulting from taking

an action a from the state s_{best} as follows:

$$\begin{aligned} s_{next}.pos &= s_{best}.pos + f(s_{best}.ori, x(a), y(a), \theta(a)) \\ s_{next}.ori &= s_{best}.ori + \theta(a) \\ s_{next}.time &= s_{best}.time + t(a) \\ s_{next}.cost &= s_{best}.cost + cost(a) \end{aligned} \tag{6.2}$$

Finally, the function G checks if s_{next} should be expanded as a child node of s_{best} in the tree. This function performs collision checking along the path from s_{best} to s_{next}. G also checks to see if s_{next} is a previously visited location in the environment. Lau et al. maintain a state-indexed table to keep track of locations visited, in order to avoid re-visiting already explored states during the search.

Once the search algorithm returns a sequence of motion-clips, the clips are joined together using interpolation to smooth out any discontinuities. The result is an actual character motion in the environment, that reaches the specified goal.

Our approach for global planning is based on the above technique, but with two important differences. First, any global planning algorithm would be intractable to plan over a continuous state space. Our BFSM is a continuous space motion machine in the sense that a behavior state could potentially represent a continuous space of motion-clips that may be parameterized in some way. Consequently, we need to sample each of our multi-sequence behavior states to generate "example" motion clips that are representative of that state. This results in a new BFSM that is identical to the Finite State Machine in Lau et al's work, with a discrete

set of motion examples at each behavior state. While this sampling restricts the range of clips that can be synthesized, it provides the user with the flexibility to trade-off the number of motion clips to be used, for time taken by the planner to come up with a reasonable plan.

Additionally, Equation 6.2 assumes that the motion clips will be playedback one after the other in sequence, without any overlaps. To synthesize a smooth motion out of a series of clips returned by the planner, the clips need to be overlapped over a transition window to effect a smooth transition. This means that a transition from a state s_{in} to a state s_{out} involves a transition between two overlapping clips, and the actual final position, orientation and duration of playback for the character in $states_{out}$ will be slightly different from that estimated by Equation 6.2. In other words, the planner in Algorithm 1 does not take into account the actual transitions between motion clips.

In our experiments we have observed that if this error build-up is not taken into account, it leads to the planner returning a motion sequence that deviates from its goal. This deviation from the specified goal is even more significant if the error accumulates early on in the solution path returned by the planner. We remedy this issue by using a function $S(s_{best}, a, e)$ as shown in the modified version of the planner shown in Algorithm 2. Given a current state s_{best} that the character is in, an action (i.e. a motion clip) a that the character needs to execute next, and the environment e, the function $S(s_{best}, a, e)$ does the following:

Algorithm 6.2: Our Modified Behavior Planner

```
Sample BFSM states to discretize the BFSM;
Tree.Initialize(s_init);
Queue.Insert(s_init,DistToGoal(s_init,s_goal));
while !Queue.Empty() do
    s_best ← Queue.RemoveMin();
    if GoalReached(s_best,s_goal) then
        return s_best ;
    end
    e ← E(s_best.time);
    A ← F(s_best, e);
    foreach a ∈ A do
        s_next ← S(s_best, a, e) ;
        if s_next ≠ null then
            Tree.Expand(s_next,s_best) ;
            Queue.Insert(s_next,DistToGoal(s_next,s_goal)) ;
        end
    end
end
return no possible path found ;
```

1. "Simulates" the transition from action $S_{best}.a$ to action a using the BFSM, and computes the resulting state s_{next}. i.e. It computes the final position, orientation and duration of motion, by actually simulating a transition from $S_{best}.a$ to a.

2. Performs collision checking with the environment e during the simulated transition above.

3. Checks to make sure that s_{next} is not an already visited state.

If there is a collision or if s_{next} is an already visited location in the environment,

then $S(s_{best}, a, e)$ returns **null**, otherwise a valid state s_{next} is returned.

There is a time overhead in using the above technique, as every transition is simulated to determine the correct state. However, it results in an accurate sequence returned by the planner that reaches the goal location. We simulate transitions with rendering turned off. In Chapter 7, we demonstrate our approach with a simple walking character that plans paths around obstacles in the environment.

6.3 Autonomous Navigation and Obstacle Avoidance

Based on the run-time motion generation scheme presented in Section 6.1, we present a navigation controller that will allow the user to direct the character to move in a virtual environment while avoiding obstacles. For simplicity, consider a single-state BFSM with parameterized walking behavior. As is typical with parameterized behaviors, the walk direction is controlled by a weight vector $\mathbf{W}$ for this behavior space.

An overview of our navigation model is shown in Figure 6.2. A *steering model* takes as input (at any time instant $\mathbf{t}$), a global heading direction $\vec{v}_g(t)$ provided by the user. Given the current location and orientation of the character at the current instant $\mathbf{t}$, it generates an appropriate weight vector $\mathbf{W}(\mathbf{t}+\mathbf{1})$ at the next time-step. This steering model basically performs the task of correcting the heading direction of the character, given its position and orientation, and the desired

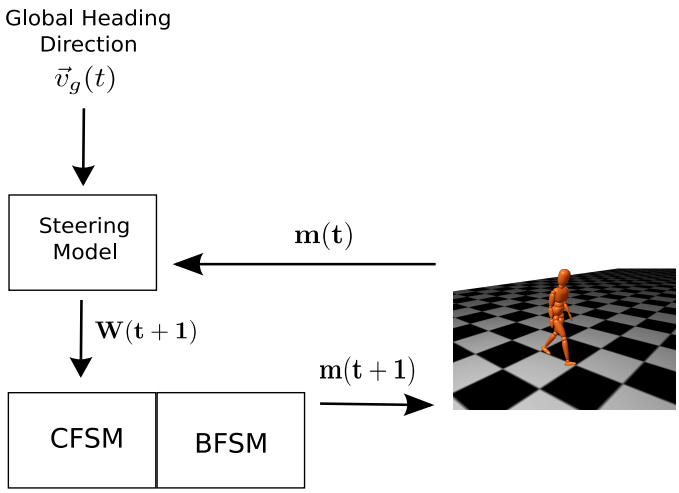


Figure 6.2: *An overview of our character navigation system. At each time-step t, the steering model generates the set of weights to steer the character, given the heading direction and the current position and orientation of the character.*

heading direction.

We now briefly describe the steering model in detail. First, we sample our behavior state with a random set of weights (i.e. $\tilde{\mathbf{W}}$ for each sample). Each motion sample is of a fixed duration (2 seconds). As shown in Figure 6.3, we measure the relative change in the character's orientation, $\tilde{\alpha_f}$ for each corresponding weight vector $\tilde{\mathbf{W}}$. We then train a neural network [7] on the training set $\langle \tilde{\mathbf{W}}, \tilde{\alpha_f} \rangle$ to give us a generalized function $\mathcal{N}(\alpha_f(t))$ that estimates the weight vector that will result in the character turning by $\alpha_f(t))$ degrees, over the 2 second interval. The details of our steering model are shown in Figure 6.4. First, the heading direction is

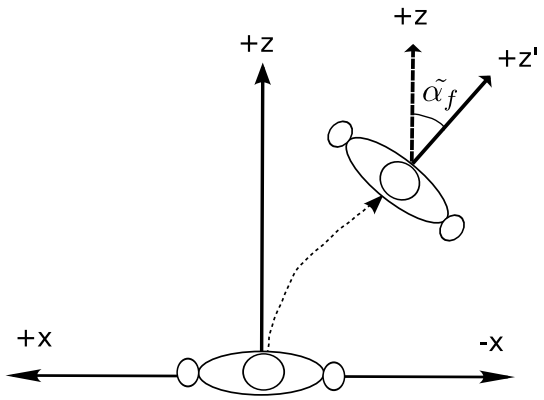


Figure 6.3: *Calculating the deviation $\tilde{\alpha_f}$ for a motion sample generated with weights $\tilde{\mathbf{W}}$. Each motion-clip is about 2 seconds long.*

converted into the character's local coordinates using the following equation.

$$\vec{v_l}(t) = (R_0(t))^{-1} * \vec{v_g}(t) \tag{6.3}$$

where $\vec{v_l}(t)$ is the heading direction in local coordinates, and $R_0(t)$ is the orientation of the character in the world. An angle of deviation, $\alpha_f(t)$, is then computed w.r.t the character's facing direction (i.e. positive Z).

$$\alpha_f(t) = \angle(\vec{v_l}(t), Z) \tag{6.4}$$

This deviation at the current time instant t can be fed into the neural network

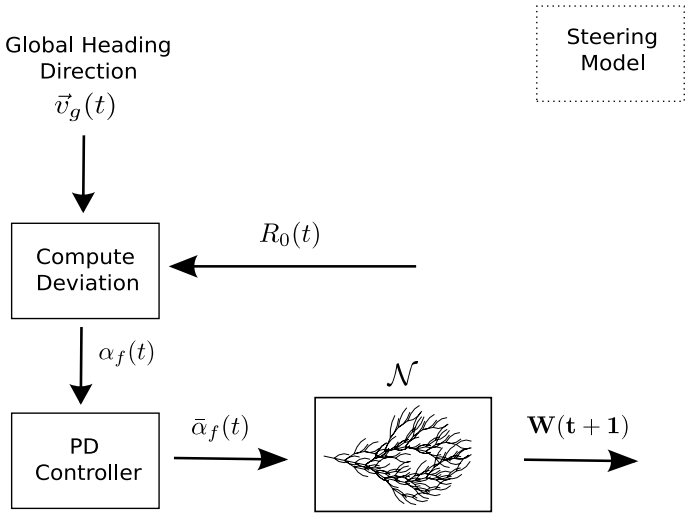


Figure 6.4: *A detailed overview of our steering model. A deviation in character's local coordinate-system is first calculated, given the character's orientation and the requested global heading direction. A PD Controller filters the deviation before using a trained neural network to estimate the weights required to steer the character towards the requested direction.*

$\mathcal{N}$ to obtain a set of weights, $\mathbf{W}(t+1)$, that generates an interpolated motion in response to the deviation.

$$\mathbf{W}(t+1) = \mathcal{N}(\alpha_f(t)) \tag{6.5}$$

It is important to note that our motion-synthesis framework (Section 6.1) allows for real-time updates to the generated motion. Hence, the steering model is consulted for an updated set of weights at every frame. As a consequence, sudden changes in the global heading direction $\vec{v}_g(t)$ (and hence $\alpha_f(t)$) may cause sudden changes

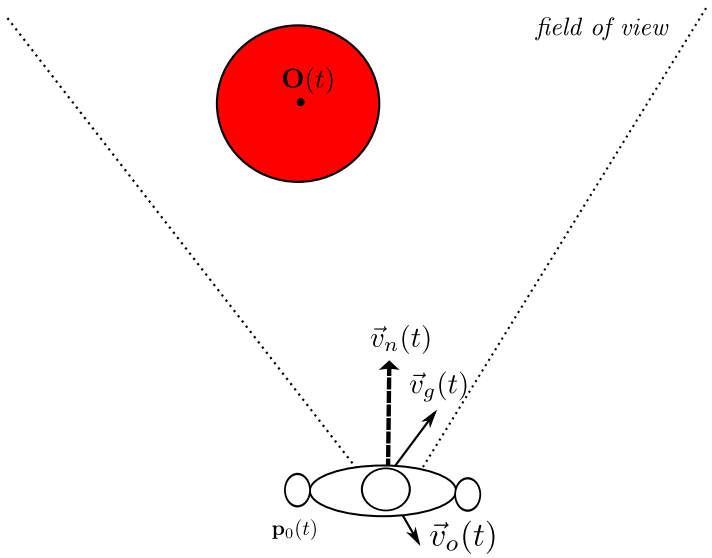


Figure 6.5: *A steer-to-avoid approach for obstacle avoidance. An obstacle in the character's* field-of-view *exerts a force which is combined with the navigation direction to calculate a new heading direction* $\vec{v}_g(t)$ *for the character.*

in weights, $\mathbf{W}(t+1)$, resulting in undesirable motion artifacts. To avoid a noisy estimate of $\alpha_f(t)$, we filter out the deviation using a critically damped PD controller (Equation 6.6).

$$\ddot{\bar{\alpha}}_f(t) = k_s * (\bar{\alpha}_f(t) - \alpha_f(t)) - k_d * \dot{\bar{\alpha}}_f(t) \tag{6.6}$$

$\bar{\alpha}_f(t)$ represents the filtered deviation that is now fed into the neural network in Equation 6.5. We demonstrate character navigation using our steering mode in Chapter 7.

To implement obstacle avoidance, we use the *steer-to-avoid* approach from Craig

Reynolds' work [50]. We assume that the obstacles are cylindrical, and define a field-of-view and a distance threshold beyond which the character is not influenced by the obstacle(Figure 6.5). As shown in the figure, an "obstacle-force", $\vec{v}_o(t)$, is calculated as soon as the obstacle is in the character's area of influence. The direction and magnitude of this force depends on the position of the obstacle with respect to the character, as shown by the following equation:

$$\vec{v}_o(t) = \frac{-1}{(\|\mathbf{O}(t) - \mathbf{p}_0(t)\| - r)^2} * \frac{\mathbf{O}(t) - \mathbf{p}_0(t)}{\|\mathbf{O}(t) - \mathbf{p}_0(t)\|} \tag{6.7}$$

where $\mathbf{O}(t)$ is the obstacle position and $\mathbf{p}_0(t)$ is the character position, and r is the radius of the obstacle. Given a navigation direction $\vec{v}_n(t)$ for the character, we compute the final heading direction, $\vec{v}_g(t)$, as:

$$\vec{v}_g(t) = \vec{v}_n(t) + w * \vec{v}_o(t) \tag{6.8}$$

where w is a weighting term. In Chapter 7, we demonstrate obstacle avoidance with both static and moving obstacles using the above formulation.

Chapter 7 – Results

For the examples presented in the following sections, we use various sources of motion capture data consisting of locomotion behaviors and martial-arts [12, 13]. All of the examples presented in this chapter were generated in real-time with no post-process. First, we present two BFSMs constructed using our mining approach described in Chapter 3, and demonstrate interactive character control using these behavior machines. We then present results from applying offline global planning to a simple BFSM (Section 6.2). Finally, we demonstrate reactive character navigation with obstacle avoidance with a locomoting character using techniques described in Section 7.3.

7.1 Interactive Character Control

We now present results of our approach for two domains - locomotion and martial arts. We demonstrate by showing the resulting BFSMs as well as animated motion (please see the accompanying video). All demonstrations were created on an Intel 3.0 GHz machine with 2GB of RAM.

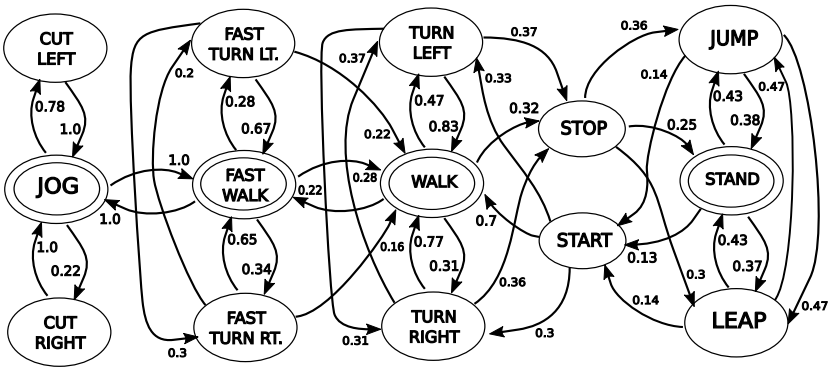


Figure 7.1: *The BFSM for our Locomotion Demonstration described in Section 7.1.1. The BFSM has been edited by fine-tuning the transition probabilities in Figure 4.1, using semi-automatic techniques described in Chapter 4.*

7.1.1 Locomotion

Our first demonstration consists of a BFSM built from a set of locomotion sequences from the CMU motion capture database [12]. The example makes use of 20 motion clips consisting of a total of 1631 frames of motion. The sequences were not edited in any way. We do, however, synthesize a useful 'idle' behavior by borrowing frames from a jump sequence.

The first step in the BFSM construction pipeline involves the application of our mining technique to the motion library (Chapter 3). The computation of the Q-graph took approximately 60 minutes using a discount factor of 0.95 for Q-learning. Extracting the BFSM structure and transition probabilities from the Q-graph took approximately 5 seconds. The resulting graph is shown in Figure 4.1. The transition probabilities were then fine-tuned using semi-automatic methods

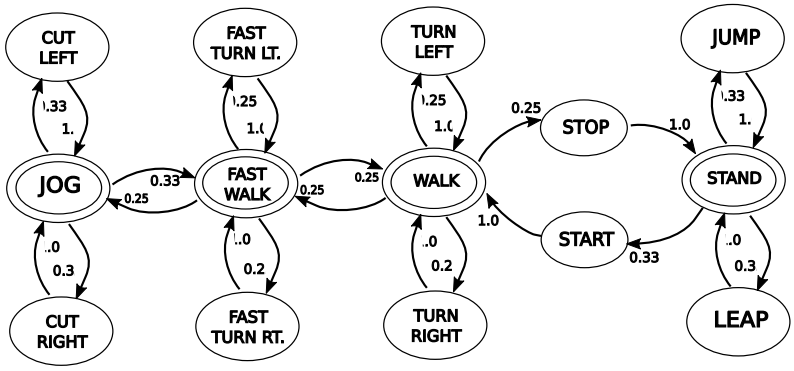


Figure 7.2: *A manually designed version of the BFSM shown in Figure 7.1. The transitions and their probabilities are hand-designed by a user.*

described in Chapter 4. The BFSM, with final transition probabilities, is shown in Figure 7.1.

The next step involves designing and populating the BFSM with data-structures for transitions and motion interpolation (Chapter 5). The behavior state transition warps were designed in less than a minute on average for each state transition. The time-warping tool in Section 5.2 aids in the transition design process. Storage for the BFSM is proportional to the number of behaviors and the motions that make up the behaviors. For each behavior state, we must store the functions outlined in Section 5.1 as well as all original motion sequences. For the locomotion demo, the BFSM (including motion, warps, and graph structure) requires approximately 3.1MB of disk space.

Compared to a BFSM designed manually by the user, the BFSM from our mining procedure had $\chi = 1.92$ and $\rho = 0.92$. According to the terminology introduced

in Equation 3.7, a value of $\rho < 1.0$ suggests that our mining approach is not able to reproduce all of the transitions manually specified by a user in Figure 7.2. The reason is as follows: the BFSM resulting from the automated procedure does not find a transition edge between the 'jog' or 'cut' behaviors and the remaining states because our dataset does not contain any motion clips exemplifying transitions between jog and walk behaviors. To remedy this, we simply force a transition from jog to fast-walk. The resulting motion is visually satisfactory.

A similar artifact occurs between the "cut" (both left and right) behaviors and "jog". Our dataset suffers from inadequate overlapping frames for a transition edge from a left or a right-cut to a jog. As a consequence, transitions from a left-cut or a right-cut behavior to a jog behavior get pruned out during the BFSM mining process. Again, to remedy this anomaly, we force these transitions in our final BFSM.

Using the locomotion BFSM, we can interactively control the character in real-time to navigate a complex obstacle course. The character is driven to navigate a short maze, jump over a pitfall, follow a trajectory on the ground and negotiate cones in an agility drill (Figure 7.3). This domain demonstrates all aspects of BFSMs including parameterized and cyclic behavior states(walking and jogging), acyclic behavior states (leaping), and multi-point transitions (walk to jog). On average, the delay between a user request and start of the transition is 0.44 seconds. All of the presented motion was generated in real time with no additional post processing.

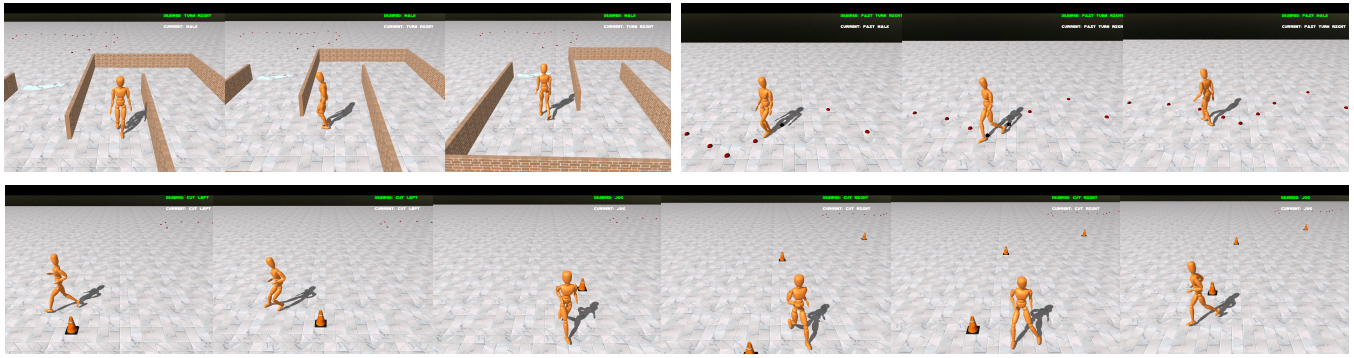


Figure 7.3: *Images from a real-time capture of interactive character control using a BFSM. Images flow left to right and top to bottom.*

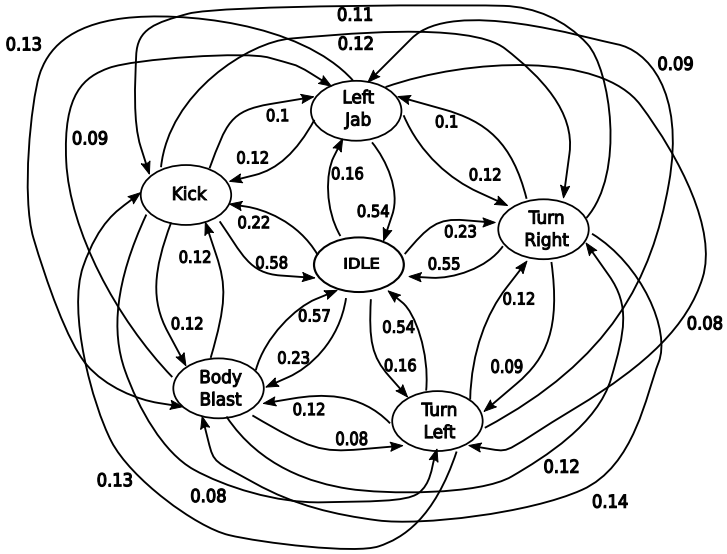


Figure 7.4: *A BFSM for the martial-arts domain. The BFSM has 6 behavior states, and is completely connected.*

7.1.2 Martial Arts

Our second demonstration is taken from the martial arts domain. We use only 6 motion clips consisting of 400 frames of animation. The final BFSM structure was computed in less than 5 minutes using our graph mining algorithm. Again, we used a discount factor of 0.95 for Q-learning. The resulting BFSM consists of 6 behavior states and 25 edges and is completely connected (see accompanying video). The connectivity factor (χ) and reproducibility index (ρ) were 2.5 and 1.0 respectively. The user is able to throw jabs, punch combinations and kicks as well as turn the character left and right.

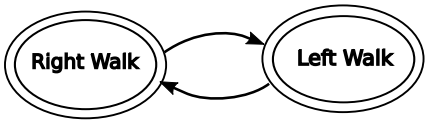


Figure 7.5: *A simplified two-state BFSM for a walking character. The parameterized turn behaviors are separated out into "Left Walk" and "Right Walk" behaviors in order to avoid motion artifacts during blending. We use this BFSM to demonstrate principles of global planning and reactive navigation with obstacle avoidance (Section 7.2 and 7.3).*

7.2 Global Planning

Figure 7.5 shows a simple two-state BFSM that was designed for a locomoting character. Each behavior, Left-walk and Right-walk, is cyclic and consists of three motion-clips that turn at 0, 45 and 90 degrees in each of the direction. Each motion-clip is approximately 2 seconds long. It was a design choice by the user to represent a parameterized walk as a combination of two separate turning behaviors. Incorporating the left and right-turning motion clips into a single "walk" behavior results in motion artifacts because each behavior consists of sharp (eg. 90 degree) turns that fail to produce artifact-free motion, when blended together.

In order to apply the global planner described in Algorithm 2, we discretize each of the behaviors in Figure 7.5. Each discrete sample is a motion-clip generated as a result of using a random vector of convex blend-weights. The number of samples generated for each behavior is a parameter that can be chosen by the user and affects the size of the search space.

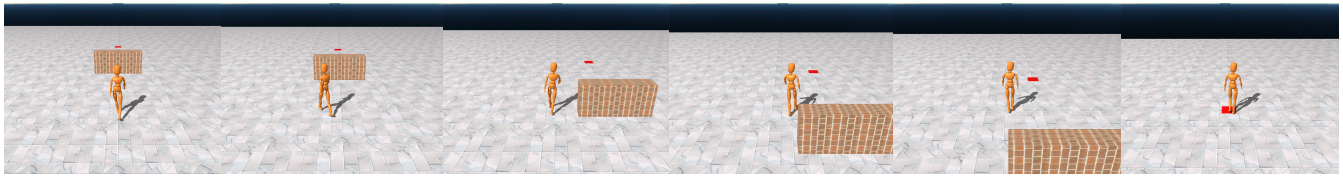


Figure 7.6: *Offline global planning. The character plans a path to a target while avoiding obstacles in the environment. Images flow left to right.*

Figure 7.6 shows an example motion-sequence synthesized by our global-planner. The planner plans a motion sequence that reaches a target location behind an obstacle in the form of a wall. For the result shown in the figure, each of the Left-walk and Right-walk state was discretized into 15 samples. Our planner took about 5.96 seconds to plan a motion clip that is 12 seconds long.

It maybe noted that since the number of samples is a user-parameter, it is easy to trade-off planning time for quality of the generated plan. For instance, with only 6 samples per behavior state, our planner took only 2.4 seconds to plan a motion-sequence. However, the motion clip takes a more indirect route when compared to the plan generated with more samples.

7.3 Navigation and Obstacle Avoidance

We use the same behavior machine shown in Figure 7.5 to demonstrate character navigation with reactive obstacle-avoidance. As described in Section 7.3, we train a separate neural network for each of the two behavior states. We use an off-the-shelf neural network library called RBF++ [7]. In order to train each of the networks to approximate the function $\mathcal{N}(\alpha_f(t))$, we use a training-set consisting of 10000 examples (see Section 7.3), and validate with a data-set of size 1000. It took approximately 30 minutes to train the neural networks. The average error (on the validation data-set) for Left-walk and Right-walk behavior was 4.1% percent and 4.3% percent, respectively.

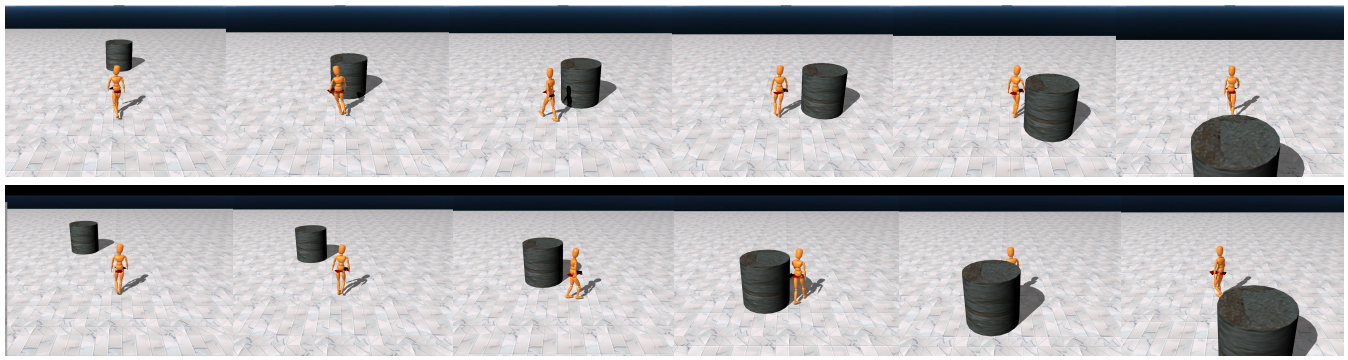


Figure 7.7: *Reactive navigation: The top row demonstrates a character navigating around a stationary obstacle and the bottom row demonstrates the same character navigating around an obstacle in motion.*

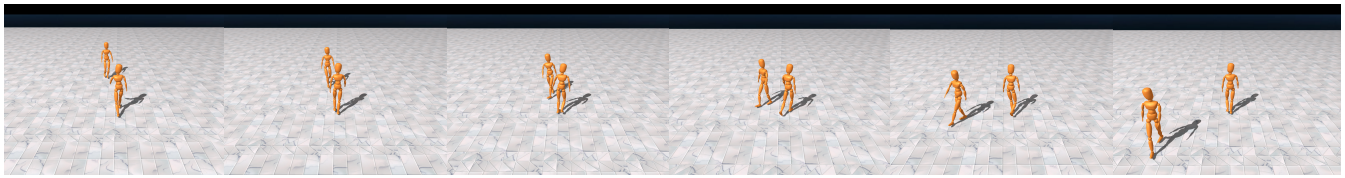


Figure 7.8: *Reactive navigation with two characters. The characters avoid running into each other while walking along the Z axis from opposite directions.*

Since our navigation controller is reactive in nature, it can handle both static and moving obstacles. Figure 7.7 shows a character navigating around both a static and a moving obstacle represented by a cylindrical object of diameter 1.5 meters. In both the cases, the character avoids the oncoming obstacle while traveling along the positive Z axis. The character uses a 60 degree field-of-view with a 7m look-ahead. We use $w = 0.25$ for both the cases in Equation 6.8. Figure 7.8 shows two characters navigating to avoid running into each other in the environment. Each character is a "moving obstacle" in the other character's field-of-view.

It maybe noted that there are two separate controllers for Left-walks and Right-walks. We choose the correct controller to use based on the direction of the final heading calculated from Equation 6.8. i.e. If the final heading points to the right of the character, we generate motion from the Right-walk behavior state using its controller. A transition from Left-walk behavior to Right-walk behavior is triggered when the heading direction switches from left to right (or vice-versa) relative to the character.

Chapter 8 – Discussion and Future Work

We have presented a novel approach to designing real-time character control that builds on the strengths of motion blending and motion graphs to generate responsive and natural motion while keeping the designer in a tight design loop. We have demonstrated our approach in multiple domains and we have shown how we exploit both automated methods (time-warp curves, graph mining, editing BFSM transitions) and manual design (identifying meaningful blends, interactive timewarp edits for transitions) for an efficient motion design pipeline. We have demonstrated the quality of the resulting motion sequences in two dynamic motion domains and shown that the resulting BFSMs can be used for real-time control in complex environments. In addition, we have also demonstrated that these BFSMs can be used for synthesizing motion sequences through both offline global planning techniques and real-time reactive obstacle avoidance techniques.

While the technology has evolved for creating parameterized continuous motion spaces and graph sequencing, less attention has been paid to methods for including the human in the loop in designing these motion spaces and graphs. On one hand, automation is a clearly desirable goal for many applications. More complex and believable environments would require larger and more complex behavior spaces constructed from many example motion-clips. Automated methods, such as those

for motion parameterization [25], find useful applications in such cases. On the other hand, user-in-the-loop design methods are necessary for interactive motion design. For instance, Kovar et al. note that only a user can reliably determine whether a set of sequences can be successfully blended by actually creating and viewing the blends [25].

Our first contribution attempts to automate the discovery of the BFSM edges and their transition probabilities. We foresee this approach as being ideal for the future end user of motion capture data. As systems become more affordable, we envision an explosion of available motion capture data. Our motion graph mining approach allows designers to capture or collect large amounts of data from the domain of interest without much attention to carefully scripted sequences and to produce a viable graph structure given that data.

The mining approach is also capable of providing useful information to the user regarding the completeness of the motion data-set. For instance, while producing the BFSM in Figure 4.1, our approach failed to discover any transition edge between a Jog behavior and a Fast-Walk behavior. The lack of a transition edge between these behaviors suggests that additional motion data (eg. a Jog-to-Fast-walk motion) is required to realize a smooth transition.

Our design pipeline can be augmented and improved by implementing an automated technique for segmenting and annotating sequences [1, 25]. Automatic segmentation of motion-capture data is also an open and difficult problem. One

advantage of our approach is that the motions do not have to be perfectly segmented. Since the user is tightly engaged in the loop of designing the behavior states, it is quite easy to visually design transitions between the appropriate parts of motions as long as they contain portions that have a reasonable correspondence.

While our current mining approach results in plausible states and transition costs, the user must be able to review the resulting motions and possibly edit the BFSM to fine-tune its transition costs. Our BFSM editing scheme presented in Chapter 4 provides one intuitive way to edit the BFSM. A user may subjectively evaluate the shortest path between two different behaviors, and decide that the behavior-transition is not appropriate, compared to an alternative path between the same behaviors. The alternative path can then be enforced by the user as the new shortest-path between the behaviors. The transition costs are automatically adjusted to ensure that the new "user-defined" behavior transition path has the least path-cost, without adversely affecting other optimal paths in the BFSM. Few manual edits such as adding new edges, deleting a few edges etc. are still required of the user.

Currently, our interface supports editing of one path at a time. This can be easily extended to allow the user to edit multiple paths at once, as demonstrated by Burton et al. in [9]. We also identified several other areas of improvement to make the interface more usable. Since the user must be able to review the resulting motions and possibly edit the BFSM in a tight visual interaction loop, we envision more intuitive methods to edit the structure of the BFSM.

While the above contributions aid in construction and manipulation of a BFSM graph, a user still has the responsibility of designing behavior spaces and transition between behavior spaces, in order to realize the BFSM. Our BFSM design process described in Chapter 5 is an attempt to provide an easy approach to design all of the data-structures needed to generate complex character animation using a BFSM. It alleviates the need to work with the motion clips at the low level of poses and sequences. An end user designs the BFSM working at a higher level of abstraction - blends and transitions.

One drawback with designing high-level graphs, such as the BFSM, is that the number of transitions could still be $O(n^2)$ for a graph with n behaviors. While we did not encounter such a scenario in our work, designing transitions for a fairly complex BFSM could turn out to be a labor-intensive task in the worst case. We believe that this tedious design process can be automated using heuristics, such as the ones described in Section 5.4.4. For instance, the transition regions defined in Figure 5.4 for different transition types could be used as a starting point to automatically search for a suitable transition warp. The user can still be closely involved in the design process by retaining the ability to adjust the timewarps interactively.

Another area for improvement is in our use of default blend weights in target behavior states during transitions. While the resulting motion is plausible, we can achieve better (and more responsive) transitions by sampling the blend weights to identify the correct target state weights. This is the approach taken in recent work

by Heck and Gleicher [20]. These additions are left for future work.

The resulting BFSMs have several useful properties. First, the behavior states can be parameterized and continuously controlled, leading to behaviors capable of being responsive to user input. Second, the overall graph encodes not only connectivity between states, but also the likelihood of these transitions occurring. As such, our BFSMs lend themselves to being exploited in several ways. First, the BFSM can be searched off-line, much like a motion graph, to identify optimal sequences given environmental constraints. Second, because the graph is coarse, it can be searched in real-time for paths that meet run-time dynamic constraints.

We demonstrated global planning using a simple BFSM with a continuously parameterized walk behavior. The advantages of global planning with continuous behavior spaces are several. First, the user can easily trade-off planning time for quality of motion generated by controlling the extent of sampling for the continuous behavior states. Second, the user has the flexibility to sample behaviors according to the needs of the targeted environment. For instance, planning a character to navigate around a dense maze may require a larger set of samples for a reasonable solution. On the other hand, planning navigation tasks in more sparse environments such as playgrounds or open spaces may require only a few samples to generate a reasonable motion sequence. Another advantage is that different behaviors can be sampled at different resolutions, in order to achieve planning goals effectively. For instance, consider planning for a character to walk over to a desk and reach an article. Suppose we assume that the character's BFSM consists of

two parameterized multi-sequence behaviors - Walking, and Reaching. The reach behavior can be more densely sampled, compared to the walk behavior, to achieve a more constrained goal of reaching a particular location in space.

The above mentioned scenarios would be difficult to realize using a discrete finite state machine presented by Lau et al [30].

We have also demonstrated real-time reactive character navigation using steer-to-avoid behavior on top of a simple BFSM. When compared to more sophisticated learning approaches such as the one presented by Treuille et al. [65], our approach provides a simple and quick alternative to set up a character that can navigate along user-given directions while avoiding obstacles. The approach works reasonably well with not only static obstacles, but also with moving obstacles in the environment, and with obstacles having different sizes. However, the approach has several drawbacks. For instance, the character does not react to avoid moving obstacles in a natural way. In real world, humans are capable of making a complex assessment about the relative position of the obstacle, and deciding to avoid the obstacle by pausing briefly, for example, before continuing to walk, as opposed to altering the route.

When compared to reactive approaches, policy-learning approaches [65] have the advantage that they can learn more optimal policies in situations where the character is faced with complex and potentially conflicting objectives. However, there are two main drawbacks that precludes them from being used with BFSMs. First,

it is not clear how to set up the learning process and build a policy for continuous behavior spaces encountered in the BFSM. In addition, it is not clear how to generalize and extend these policies. For instance, how do we extend the obstacle avoidance policy to handle obstacles with different sizes ? How can we adapt an existing policy to moving obstacles ?

Building robust controllers for complex BFSMs such as Figure 7.1 is a challenging problem, and presents interesting avenues for future work. One option is to sample each multi-sequence behavior state to create a discrete set of motion examples, and learn control policies as presented by Treuille et al. [65]. The drawback is that it does not take full advantage of the continuous behavior spaces in the BFSM. Another option is to extend the approach taken by Lau et al. [30]. Obstacles and locations in the environment could be annotated with information that would enable the character to transition to a particular behavior in the BFSM. For instance, a low arch could be annotated with "crawl" behavior, alongwith the height to clear the arch. The character could then transition into a parameterized "crawl" behavior to clear the obstacle. Unfortunately, the annotation based approach cannot be extended to dynamically changing environments.

A more generic approach would control a BFSM-driven character at two levels. At a parameterized behavior state, a reactive controller may select the right set of blend weights for animating the character in the environment. The controller could be a "memoryless" controller, such as steer-to-avoid. At a higher level, another controller may be responsible for making high-level decisions. These decisions

would enable the character to plan behavior transitions based on the surrounding environment.

While we have presented tools that may potentially aid an end-user in easily creating content using motion-capture data, we recognize that these approaches can be further refined, and its effectiveness can be validated by performing usability studies. The steps for creating interactive content from motion-capture data can be summarized, in general, as follows:

1. **Motion Graph Mining:** Using motion-capture sequences to identify the nature and structure of a BFSM.

2. **Building States and Transitions:** Aligning the motion-clip(s) in each behavior, and designing transitions between behaviors.

3. **Fine-tuning:** Adjusting and editing the BFSM structure according to the end-user's needs and targeted application.

4. **Content Creation:** Leveraging on the BFSM to create interactive content, or offline motion-synthesis etc.

The usability study could be performed in context of the above steps in the pipeline. Such a study would help quantify the merits of our approach in designing and editing a BFSM. In addition, such a study would shed light over the most unintuitive parts of the design pipeline. This in turn would provide a launching ramp to investigate further into tools and techniques that would allow an end-user

to efficiently leverage on his/her skills in creating 3D content from motion-capture data.

Bibliography

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

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

[3] B. Bodenheimer, C. Rose, S. Rosenthal, and J. Pella. The process of motion capture: Dealing with the data. In D. Thalmann and M. van de Panne, editors, *Computer Animation and Simulation '97*, pages 3–18. Springer NY, September 1997. Eurographics Animation Workshop.

[4] R. Bowden. Learning statistical models of human motion. In *IEEE Workshop on Human Modeling, Analysis and Synthesis, CVPR 2000*, pages 199–206, 2000.

[5] M. Brand and A. Hertzmann. Style machines. In Kurt Akeley, editor, *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 183–192. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[6] A. Bruderlin and L. Williams. Motion signal processing. In Robert Cook, editor, *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 97–104. Addison Wesley, 1995.

[7] S. Buck. Rbf++: A library for function approximation by means of radial basis functions.
http://www9.in.tum.de/people/buck/RBF/, 2003.

[8] D. Burton. On the inverse shortest path problem, 1993.

[9] D. Burton and Ph. L. Toint. On an instance of the inverse shortest paths problem. *Math. Program.*, 53(1):45–61, 1992.

[10] B. Chiu, V. Zordan, and C-C. Wu. State-annotated motion graphs. In *ACM Virtual Reality Software and Technology (VRST) 2007 - to appear.* ACM Press, 2007.

[11] M.G. Choi, J. Lee, and S.Y. Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics*, 22(2):182–203, 2003.

[12] CMU. Cmu motion capture database. http://mocap.cs.cmu.edu, 2007.

[13] Advanced Computing Center for Arts and Design. Accad motion capture lab. http://accad.osu.edu/research/mocap/, 2003.

[14] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 29–38, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[15] P. Glardon, R. Boulic, and D. Thalmann. Dynamic obstacle avoidance for real-time character animation. *Vis. Comput.*, 22(6):399–414, 2006.

[16] M. Gleicher. Motion editing with spacetime constraints. *1997 Symposium on Interactive 3D Graphics*, pages 139–148, 1997.

[17] M. Gleicher. Retargeting motion to new characters. In Michael Cohen, editor, *Proceedings of ACM SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 33–42. Addison Wesley, 1998.

[18] M. Gleicher. Motion path editing. In *2001 ACM Symposium on Interactive 3D Graphics*. ACM, march 2001.

[19] M. Gleicher, H. Shin, L. Kovar, and A. Jepsen. Snap-together motion: assembling run-time animations. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 181–188. ACM Press, 2003.

[20] R. Heck and M. Gleicher. Parametric motion graphs. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 129–136. ACM Press, 2007.

[21] E. Hsu, S. Gentry, and J. Popovic. Example-based control of human motion. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM Press, 2004.

[22] L. Ikemoto, O. Arikan, and D. A. Forsyth. Learning to move autonomously in a hostile world. Technical Report Technical Report UCB/CSD-51395, 2005.

[23] T-H. Kim, S. I. Park, and S. Y. Shin. Rhythmic-motion synthesis based on motion-beat analysis. *ACM Trans. Graph.*, 22(3):392–401, 2003.

[24] S. Korada. Creating and editing motion machines for 3d characters. Master's Thesis, Oregon State University, 2008.

[25] L. Kovar and M. Gleicher. Automated extraction and parameterization of motions in large data sets. volume 23, pages 559–568. ACM Press, 2004.

[26] L. Kovar, M. Gleicher, and F. H. Pighin. Motion graphs. *ACM Trans. Graph.*, 21(3):473–482, 2002.

[27] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 214–224, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[28] T. Kwon and S. Y. Shin. Motion modeling for on-line locomotion synthesis. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 29–38. ACM Press, 2005.

[29] Y. Lai, S. Chenney, and S. Fan. Group motion graphs. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 281–290, New York, NY, USA, 2005. ACM Press.

[30] M. Lau and J. J. Kuffner. Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280. ACM Press, 2005.

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

[32] J. Lee and K. H. Lee. Precomputing avatar behavior from human motion data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM Press, 2004.

[33] J. Lee and K. H. Lee. Precomputing avatar behavior from human motion data. *Graph. Models*, 68(2):158–174, 2006.

[34] J. Lee and S. Y. Shin. A hierarchical approach to interactive motion editing for human-likefigures. In Alyn Rockwood, editor, *Proceedings of ACM SIGGRAPH 1999*, pages 39–48, Los Angeles, 1999. Addison Wesley Longman.

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

[36] LindenLab. Second life.
http://secondlife.com/, 2008.

[37] R. Metoyer, V. Zordan, B. Hermens, C. Wu, and M. Soriano. Anticipating impacts. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 33, New York, NY, USA, 2006. ACM.

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

[39] 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, IEEE Computer Society*, pages 137 – 142, 2000.

[40] NaturalPoint. Optitrack.
http://www.naturalpoint.com/optitrack/, 2008.

[41] P. Poulin P. Beaudoin, M. van de Panne. Automatic construction of compact motion graphs. Technical Report 1296, May 2007.

[42] S. I. Park, H. J. Shin, T. H. Kim, and S. Y. Shin. On-line motion blending for real-time locomotion generation: Research articles. *Comput. Animat. Virtual Worlds*, 15(3-4):125–138, 2004.

[43] S. I. Park, H. J. Shin, and S. Y. Shin. On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 105–111. ACM Press, 2002.

[44] J. Pettré, J. Laumond, and T. Siméon. A 2-stages locomotion planner for digital actors. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 258–264, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[45] Z. Popovic and A. Witkin. Physically based motion transformation. *Proceedings of ACM SIGGRAPH 99*, pages 11–20, August 1999.

[46] M. J. D. Powell. Radial basis functions for multivariable interpolation: a review. In *Algorithms for approximation*, pages 143–167. Clarendon Press, 1987.

[47] K. Pullen and C. Bregler. Animating by multi-level sampling. In *Proceedings of IEEE Computer Animation 2000*, 2000.

[48] L. Rabiner and B-H. Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[49] Robert J. Renka. Multivariate interpolation of large sets of scattered data. *ACM Trans. Math. Softw.*, 14(2):139–148, 1988.

[50] C. W. Reynolds. Not bumping into things. http://www.red3d.com/cwr/nobump/nobump.html, 1988.

[51] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Proceedings of ACM SIGGRAPH 87*, 21(4):25–34, July 1987.

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

[53] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New York, 2005.

[54] A. Safonova and J. K. Hodgins. Construction and optimal search of interpolated motion graphs. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 106. ACM, 2007.

[55] A. Schödl and I. Essa. Controlled animation of video sprites. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Computer Graphics Proceedings, Annual Conference Series, pages 121–127. ACM Press, 2002.

[56] A. Schödl, R. Szeliski, D. Salesin, and I. Essa. Video textures. *ACM Transactions on Graphics*, pages 489–498, 2000.

[57] H. J. Shin and H. S. Oh. Fat graphs: constructing an interactive character with continuous controls. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 291–298. Eurographics Association, 2006.

[58] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254. ACM Press, 1985.

[59] M. Srinivasan, R. A. Metoyer, and E. N. Mortensen. Controllable real-time locomotion using mobility maps. In *Graphics Interface*. Canadian Human-Computer Commnication Society, A K Peters, May 2005.

[60] M. Sung, L. Kovar, and M. Gleicher. Fast and accurate goal-directed motion synthesis for crowds. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 291–300. ACM Press, 2005.

[61] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[62] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[63] Demetri Terzopoulos. Artificial life for computer graphics. *Commun. ACM*, 42(8):32–42, 1999.

[64] Demetri Terzopoulos, Xiaoyuan Tu, and Radek Grzeszczuk. Artificial fishes: autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artif. Life*, 1(4):327–351, 1994.

[65] A. Treuille, Y. Lee, and Z. Popović. Near-optimal character animation with continuous control. *ACM Trans. Graph.*, 26(3):7, 2007.

[66] J. Wang and B. Bodenheimer. An evaluation of a cost metric for selecting transitions between motion segments. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 232–238. Eurographics Association, 2003.

[67] D. J. Wiley and J. K. Hahn. Interpolation synthesis of articulated figure motion. *IEEE Comput. Graph. Appl.*, 17(6):39–45, 1997.

[68] A. Witkin and Z. Popovic. Motion warping. In Robert Cook, editor, *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 105–108. Addison Wesley, 1995.

[69] V. B. Zordan and J. K. Hodgins. Tracking and modifying upper-body human motion data with dynamic simulation. In *Computer Animation and Simulation '99*, 1999.

[70] V. B. Zordan and J. K. Hodgins. Motion capture-driven simulations that hit and react. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–96. ACM Press, 2002.

[71] V. B. Zordan and N. C. Van Der Horst. Mapping optical motion capture data to skeletal motion using a physical model. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 245–250. Eurographics Association, 2003.

[72] V. B. Zordan, A. Majkowska, B. Chiu, and M. Fast. Dynamic response for motion capture animation. *ACM Trans. Graph.*, 24(3):697–701, 2005.

APPENDICES

A.1 Representing Articulated Figures

Motion data consists of a bundle of motion signals. Each signal represents a sequence of regularly sampled values for each degree of freedom. These sequence of sampled values form a motion clip that consists of a sequence of frames. In each frame, the sampled values from the signal determine the configuration of an articulated figure at that frame.

In the joint-angle representation, a pose of a character is represented as a hierarchy of joints connected by rigid links. Each joint has three degrees of rotational freedom. For instance in (Figure A.1) the character has 23 joints , and therefore a total of 69 degrees of freedom.

Figure A.1: *An articulated figure with 23 joints*

The orientation and offset for each child joint in the hierarchy is specified with respect to its parent joint. The position and orientation of the character is determined by the position and orientation of the root joint (Figure A.2). In the point-cloud representation of the character, a pose is represented by a cloud of 3-D points "sprinkled" all over the character (Figure A.3).

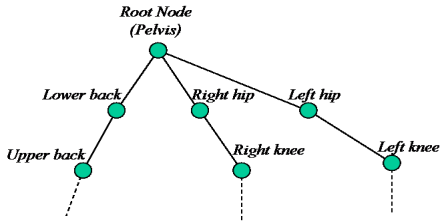


Figure A.2: *Hierarchical representation of an articulated figure. The joints have offsets and orientation with respect to their parent and the root node has a translation and orientation, that defines the position and orientation of the character.*

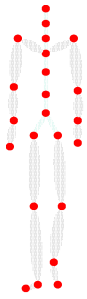


Figure A.3: *An articulated figure represented by a point-cloud.*

A.2 Comparing frames

The most common queries to the motion library aim at extracting poses or sequence of poses that are visually or logically similar. See Appendix A.1 for some common representations of an articulated figure. Most approaches use the joint angle representation where a pose or a frame of motion data is represented as a discrete signal $\mathbf{m(t)} = \{\mathbf{p}(t), \mathbf{q}(t), \mathbf{q}_1(t), ..., \mathbf{q}_N(t)\}$ where N is the number of joints in the character. $\mathbf{p}(t)$ and $\mathbf{q}(t)$ are vectors describing the position and orientation of the root body of the character, and $\mathbf{q}_i(t)$ is the orientation of the i^{th} joint in its parent coordinate system. In the point cloud representation [26], $\mathbf{m(t)} = \{\mathbf{p}_1(t), ..., \mathbf{p}_N(t)\}$ where $\mathbf{p}_i(t)$ is the i^{th} point in a point cloud driven by the character skeleton (Appendix A.1).

Distance between two frames is defined in two different ways, corresponding to the two representations. For the joint angle representation, Lee et.al. [31] define the distance (D_{ij}) between two poses i and j as a weighted combination of joint angle differences (P_{ij}) and joint velocity differences (V_{ij}).

$$D_{ij} = P_{ij} + \omega V_{ij} \tag{A.1}$$

The orientation differences are computed using quaternion arithmetic [58] to avoid singularity issues with other representations.

For the point cloud representation, Kovar et.al. [26] find the distance between

two poses by applying a 2D rigid transformation to align the point clouds, and calculate the Euclidean distance between the corresponding points in the point cloud.

$$D = \sum_i \|p_i - T_{\theta,x,z} p'_i\|^2 \tag{A.2}$$

The 2D rigid transformation $T_{\theta,x,z}$ is computed through an optimization process that has a closed-form solution [26].

A.3 Comparing sequences

Large motion libraries may contain many motion examples that are variants of the same kind of motion. Users would like to have an efficient and automated technique to identify and extract such motion examples. Finding similar sequences is a much more difficult problem than finding similar frames. As Kovar and Gleicher [25] note, a large distance metric between two sequences may reflect either that the motion examples are unrelated or that they are different variations of the same actions. There is no way to distinguish between these cases. For instance, a "front punch" and "sideways punch" are variations of the same "punch" action.

Kovar et.al. [25] suggest a solution by implementing a multi-step search. Distance between two motion examples is determined by identifying corresponding frames and computing the average frame-to-frame cost. Corresponding frames are identified by time-warping the two sequences using the *dynamic time-warping* algorithm [48].