AN ABSTRACT OF THE THESIS OF

Madhusudhanan Srinivasan for the degree of Master of Science in Computer Science presented on March 17, 2005.

Title: Interactive Human Locomotion Using Motion Graphs and Mobility Maps

Redacted for Privacy

Abstract approved: [illegible]

Ronald A. Metoyer

Graph-based approaches for sequencing motion capture data have produced some of the most realistic and controllable character motion to date. Most previous graph-based approaches have employed a run-time global search to find paths through the motion graph that meet user-defined constraints such as a desired locomotion path. Such searches do not scale well to large numbers of characters. In this thesis, we describe a locomotion approach that benefits from the realism of graph-based approaches while maintaining basic user control and scaling well to large numbers of characters. Our approach is based on precomputing multiple least cost sequences from every state in a state-action graph. We store these precomputed sequences in a data structure called a mobility map and perform a local search of this map at run-time to generate motion sequences in real time that achieve user constraints in a natural manner. We demonstrate the quality of the motion through various example locomotion tasks including target tracking and collision avoidance. We demonstrate scalability by animating crowds of up to a hundred and fifty rendered articulated walking characters at real-time rates.

Interactive Human Locomotion Using Motion Graphs and Mobility Maps

by

Madhusudhanan Srinivasan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented March 17, 2005
Commencement June 2005

Master of Science thesis of Madhusudhanan Srinivasan presented on March 17, 2005

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Associate Director of the School of Electrical Engineering and Computer Science

Redacted for Privacy

Dean of the Graduate School

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

Redacted for Privacy

Madhusudhanan Srinivasan, Author

ACKNOWLEDGMENTS

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 acheive the educational goals that I had set for myself. I am deeply indebted to my younger sister, 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 unending support and motivation. She has always stood by me during the times I doubted myself, and offered courage and love. I humbly thank her for being in 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 Dr. Eric Mortensen whose insightful discussions and inputs have always been priceless. I express my gratitude to all my other comittee members, for their time and efforts, to serve on my committee. Thank you all all very much.

I would also like to thank my friends at the Interactive Graphics and Vision 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, Seattle and Virginia for their enduring support. I would also like to thank all my friends in Corvallis who have always exepcted the best from me.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

DEDICATION

I dedicate this thesis to my father and mother, who are responsible for what I am today.

Interactive Human Locomotion Using Motion Graphs and Mobility Maps

1. INTRODUCTION

Interactive environments such as video games, architectural walkthroughs, and training simulators are present in many aspects of our everyday lives. With the gain in popularity of 3D interactive environments, it is evident that real-time, controllable character motion is needed. Of particular importance in these environments are human characters that move about the environment in a believable manner: both in terms of realism, and in terms of controllability.

When we talk about motion synthesis for characters in interactive environments, speed is clearly an issue and in complex scenes with many characters, the motion generation technique must scale well to multiple characters. For example, imagine an architectural visualization of a train station design. To experiment with design scenarios, an architect might want to populate the design and interactively modify the design while observing the motion of the simulated pedestrians. The simulated pedestrians should be realistic, should respond to each other and the environment and should do so in real-time. Furthermore, the architect should be able to populate a scene with a substantial number of pedestrians.

Creating such a scene is currently a challenging task. With the advance of motion capture graph-based approaches, one can generate realistic motion that is controllable in several ways including the placement of keyframes and desired paths. With a robust library of motion capture data, one can generate compelling scenes with fairly strict user constraints on character location, pose, and paths.

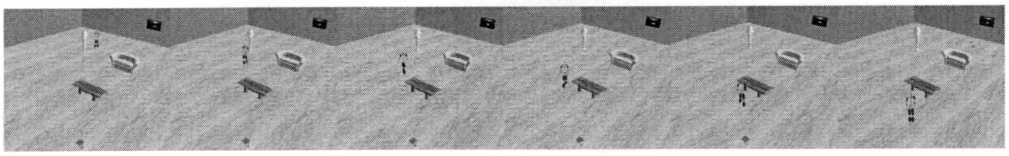


FIGURE 1.1. High-level direction by specification of location in the environment

However, although these approaches generate compelling motion, they typically require expensive search procedures and are therefore not suited to real-time environments or large numbers of characters. On the other hand, techniques that are real-time and scalable to a large number of characters often lack the compelling life-like details provided by motion capture data.

The goal behind this thesis was to create an approach that allowed a user to quickly populate an environment and direct the characters at a high level. In our case, high-level direction is the specification of desired goals on the floor plane of the environment (Figure 1.1). The user requires the character to reach these goals by locomotion. The task of figuring out the best and the most believable way to get to the goal is left to the character. For instance, a running character should come to a complete stop, turn around and walk back when the user specifies a spot behind the character. Or, the character should stop and turn to its side and start walking, to reach a location specified orthogonal to its initial direction of motion. The task of choosing the best motion transitions depends on and is limited by the motion data used.

In order to populate a virtual environment with multiple characters and direct them, the approach should be scalable. For interactive control, the character motion should be reactive to high level directives. A straight walking character should make immediate changes to its motion, and start turning in a believable manner when directed to do so. For this reason, we choose to precompute as much of the motion synthesis problem as possible. Precomputed motion segments can then be tied together at run time and this scales well with the number of characters in the environment. Run time locomotion synthesis can be combined with high-level directives to produce controllable character motion.

Our solution to the problem involves precomputing all the reachable locations from a given pose of a character over a small time window. The result is a tradeoff of memory for computation time. We then develop a novel approach to index into precomputed motion sequences. Like Lee *et al.*, our solution is based on a state–action graph, built directly from the pose–transition graph [25]. However, instead of precomputing action policies for every sampled location in the environment, we precompute possible action sequences from every state. This provides a flexible structure that can be queried at run time. Other graph-based approaches search the pose transition graph, or some form of the graph, to find a graph–walk that globally achieves some set of user constraints. Instead of using this graph directly, we reduce it to a state–action graph [25]. A state represents a pose, and an action represents a sequence of poses transitioning between a pair of states. We compute the sequence of actions with the least cummulative cost between every pair of states in the state–action graph. This all pairs shortest path (APSP) matrix encodes the smoothest sequence of actions (in pose space) from one state to any other state.

As Kovar *et al.* noted in 2002, this APSP matrix alone is virtually useless [19]. It must be indexed via a start and end pose and even then, we are given no guarantees about the spatial path traveled or the time elapsed during the graph walk between the two poses. We introduce the mobility map for mapping possible character movements over a specified time window to end poses. This mobility

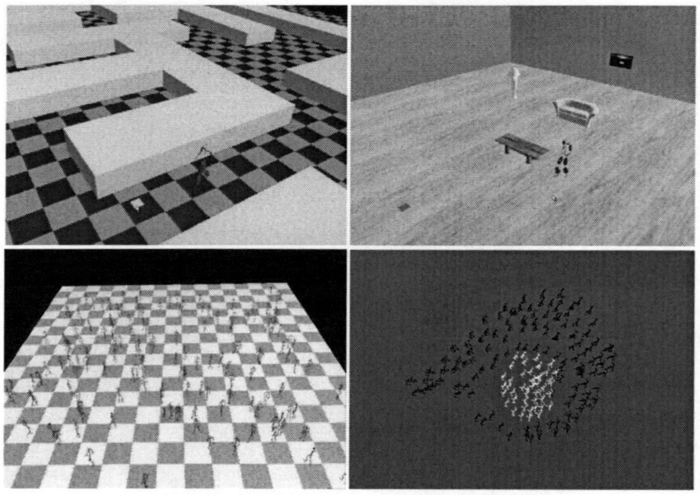


FIGURE 1.2. Locomotion control using our approach. (Top, left) The user can control a character by giving it desired target locations. (Top, right) A single character tracks a target while avoiding stationary obstacles. (Bottom, left) and (Bottom, right) A demonstration of scalability. These scenes contain 150 characters tracking targets in real time.

map encodes, over a small time frame, all locations the character can move to given its current pose. Along with the locations, it stores the resulting pose at the end of the time window. We then perform a local greedy search of these options combined with a cost function to choose the option that best achieves the user's constraints. The current pose and target pose associated with the chosen option then index into the APSP matrix in constant time to produce a smooth motion sequence.

This approach results in real-time character motion that scales well with the introduction of more characters into the scene and that allows for high level character control. We demonstrate the performance of our system through three example scenes. In the first example, the character simply tracks the user's mouse position. The user can interactively and continuously direct the character to locations in the environment by pointing on the ground plane. In the second example scene, we demonstrate reactive collision avoidance. In the last example, we demonstrate scalability with a scene containing 150 walking characters (Figure 1.2). All example movies are recorded from the application in real-time.

We present our thesis as follows. First we address the problem of natural motion generation. we describe the preprocessing steps required to build a graph of all possible natural motions. Then, we present a data-structure for representing control - the *Mobility Map*. Finally, we present a run-time algorithm for combining control and motion generation, that results in controllable character motion. We describe the run-time search process and how it integrates the user's constraints with the precomputed structures. We shall conclude with a discussion of implications of using mobility maps and future research directions.

2. LITERATURE REVIEW

Character motion synthesis from motion data has been an active area of research in computer graphics in recent years. When animating characters using motion capture data, motion editing refers to the techniques for editing and combining individual sequences of motion data to synthesise new motion sequences. Retargetting refers to the techniques used for mapping motion sequences to characters with different physical characteristics. Motion synthesis is a more generic term used for sythesis of motion either by editing motion sequences, by retargetting or by combining existing motion sequences in a meaningful way.

2.1. Motion Editing and Motion Retargetting

Early work focussed on techniques for editing and combining individual sequences of data. One of the earliest and probably the most significant contribution to motion editing techniques was by Witkin and Popovic [45]. They present a scheme that operates on a set of *motion curves*. A motion curve describes the value of one of the model's parameters, for instance the orientation of the knee joint, as a function of time. Constraints are imposed by the animator in the form of keyframes. Time warp constraints are easily imposed by sliding the keyframe markers on a timeline. The new motion curve is generated by computing the scale and offset required to warp the original motion curve. This scale and offset is computed as a function of time. To concatenate motion clips with blending, they overlap an interval at the end of the first clip with an interval at the beginning of the second and progressively blend from the first clip to the second over the course of the overlap interval using an ease-in/ease-out function.

Bruderlin and Williams approach the same problem in a different way [6]. They use techniques from image and signal processing for multiresolution analysis of human motion. They treat the motion curve as a sampled signal. The entire human body motion can then be described by a set of sampled one-dimensional signals over time. Principles of multiresolution filtering can then be applied to each of these signals. Low frequency components contain general, gross motion patterns, whereas high frequencies contain detail. This resolves the motion curves into multiple frequency bands. They combine multiresolution analysis with multitarget interpolation to blend the frequency bands of two or more movements seperately. A meaningful interpolation between two movement requires that they parametrically correspond. In other words, the movements need to be time-aligned with each other. This is automatically achieved by performing a dynamic time warp between the two motion signals. They introduce *displacement mapping* as a technique to constrain motion playback through keyframes specified by the animator. Displacement mapping provides a means to change the shape of a signal locally through a displacement map while maintaining continuity and preserving the global shape of the signal.

More recently, Gleicher presented a novel approach for motion path editing [11]. His work applied and extended the displacement mapping technique to provide more control over a character's motion path. The goal is to provide the user with an intuitive tool to edit and modify motion paths, without worrying about the details of the character motion. This can be acheived by storing the position and the orientation of the character at every frame as a local detail relative to the direction of motion. In this way, the details are preserved when the motion direction is changed to follow a different path. The character can then be made to playback along any user given path by comparing the two paths. The relative orientation between the two paths is extracted by comparing the path tangents at every point. This additional relative orientation is then applied to the character, with the effect that the character follows the new path. For instance, users can take a straight walking motion path and alter it to make the character walk along a curve. Foot-plant constraints are enforced using inverse kinematics.

The motion editing problem has also been approached from an optimization persepctive by many researchers. Gleicher presents a method to interactively position characters using direct manipulation [9]. A spacetime constraints solver edits and optimizes the existing motion clip to meet user constraints. He presents a simple and effective constraint formulation technique that can be solved at interactive rates. His work combines inverse kinematic constraints with motion smoothness constraints. Lee and Shin present a similar constraint based approach for motion editing [26]. They combine and augment their inverse kinematic solver with a hierarchical curve fitting technique. The kinematic solver is used to adjust the configuration of an articulated figure to meet constraints in each frame. The motion displacement of every joint at each constrained frame is smoothly propagated to multiple frames using curve fitting techniques. Their work differs from that of Gleicher [9] in that they decouple the smoothness constraints across frames from the inverse kinematic constraints at each frame.

Lee and Shin also extend their work to solve the problem of retargetting motion data to characters with different physical characteristics. Gleicher presents a similar spacetime optimization technique for retargetting motion from one character to another [10]. He identifies specifc features of the motion as constraints that must be maintained. A spacetime constraints solver computes an adapted motion that re-establishes these constraints while preserving the frequency characteristics of the original signal. For a good survey of these and other constraint-based motion editing approaches see [13].

The work in this thesis focuses not on editing existing data sequences but on generating continuous streams of motion by piecing together sequences of data.

2.2. Motion Synthesis

A quite different approach for motion synthesis is to solve the opposite problem. Rather than modify motion capture data with constraints such as keyframes, Pullen and Bregler presented a novel approach for using motion capture data to augment keyframed motion with the realistic detail from the motion capture data [35].

Another popular approach for synthesizing motion is to interpolate multiple motion sequence examples to generate new sequences that are similar to the originals. Rose *et al.* use a radial basis function model to generalize motion captured behaviors such as walking and running for speed and angle for the terrain. They also emphasize parametric control of emotional expressiveness for a set of basis behaviors [36]. Park *et al.* present a scattered data interpolation method that produces on-the-fly locomotion controllable with user defined paths [31]. Wiley and Hahn show that linear interpolation of example hand motions can produce compelling motion [44]. Kovar and Gleicher produce a continuous parameterized space by extracting and blending similar motion clips from a database [20], [21].

A quite different approach for motion synthesis is to generate sequences by drawing upon examples. Hsu *et al.* match movements of a lead dancer to stored example clips to extract and synthesize corresponding follow movements [16]. Kim *et al.* present a similar approach for generating motion corresponding to an input rhythm [15]. In this thesis work, we will focus solely on sequencing provided data without significantly introducing any new motion data through interpolation or blending.

2.2.1. Statistical Modeling of Human Motion

Researchers have focussed on uncovering the statistical nature of a library of data in order to generate new motion sequences with similar statistical properties [3, 34, 4, 27, 28].

In most of the statistical approaches, motion generation is cast as an unsupervised learning problem in which the goal is to acquire a generative model that captures the essential structure of the motion data. One characteristic feature of all the statistical approaches is the representation of one or more motion frames as a state. A motion playback sequence can then be considered as a probability distribution over time-series and can be modeled as a markov chain. This generative model is then used for synthesis of new motion sequences.

Various statistical approaches use a state-space representation of motion parameters. Bowden considers human motion as a deformation of shape in a three-dimensional space [3]. Linear shape deformations can be modeled using a point distribution model through principle component analysis of shape deformation. Human motion deformation is non-linear. Non-linear statistical models of deformation are learned through piecewise linear approximation of deformation. The temporal dynamics of the model is represented by a markov chain. In this manner, a spatio-temporal model of human motion can be learned and reproduced. Tanco presents a similar approach for motion generation [28].

Li *et al.* model the local repetitive patterns in complex human motion using a linear dynamic system and the global dynamics of the entire sequence by switching between these linear systems. [27]. The local repetitive patterns may consist of primitives like spinning, hopping, kicking, and tiptoeing for instance, in a dance sequence. These patterns are analogous to textures and are called *motion textons*. Each motion texton in the state-space is represented by a linear dynamic system that is learned from motion examples using an expectation-maxmization algorithm. Again, the relationship between the motion textons is modeled as a markov process, and is represented as a matrix of transition probabilities. Once the motion is learned, users can synthesize and edit the motion at the texton level and at the distribution level.

Brand *et al.* make a distinction between structure and style in motion data. Motion structure is captured by transition probabilities represented by a hidden markov model. States are modeled by a gaussian distribution over a small space of motion frames. A multidimensional hidden markov model is parameterised by the style of the motion and is called a style machine. The style machine defines a space of HMMs and fixing the style of motion yields a unique HMM.

These approaches generate new motion sequences with characteristics or style similar to that of the motion library, but do not take user constraints into account. In this paper, we are interested in generating motion that meets user defined constraints.

2.2.2. Graph based approaches

In 2000, Schödl *et al.* presented the video textures approach for generating infinitely long video sequences given a library of video data. Particularly interesting was the controllable motion sprite example of a fish that could interactively track the user's mouse [38]. The controllable sprites inspired several of the examples that will be presented in this paper. This work was followed by an improved approach for interactively controlled video sprites [37]. The Video Textures work inspired a body of work in the motion capture area presented at Siggraph 2002. Each of these approaches was built around the use of a graph to represent the possible transitions among the motion capture data. This graph representation was much like the graph representation used by Schödl *et al.* to structure the video texture sequences. Kovar and his colleagues build a motion graph from sequences of motion capture data, synthesize smooth transitions to add to the graph, and search the graph to find a sequence of poses that generates character locomotion along an arbitrary path [19]. Lee *et al.* take a very similar approach with a two-layer graph that allows for efficient search. They also presented several different user interaction methods including trajectory specification, keyframe specification, and a full-body vision-based interface. Arikan and Forsyth also presented a graph-based approach with a hierarchy of graphs and a novel randomized search technique that produces controllable animated motion at interactive rates [2]. Arikan *et al.* followed this work with a technique for generating motion from automatically annotated libraries of motion capture data [1]. Each of these approaches created very natural motion that achieved user constraints such as path following, keyframes, or task performance but each these approaches also involves a run-time global search for a sequence. Although these approaches generate highly controllable and strikingly natural motion, they do so at the cost of time and scalability. In 2003, Gleicher *et al.* presented "snap-together" motion which relies on preprocessing the data for an efficient simple graph structure that can be queried at run-time to generate natural transitions

[12]. Our approach builds on a similar structure for natural transitions. We augment our structure with spatial information to enable real-time path planning with natural-looking transitions. Most recently, Lee *et al.* [25] presented a reinforcement learning approach for precomputing action policies based on a motion graph. This approach is similar to ours, however, the solution is tied to the cost function used for computing the optimal actions during the learning process. We present a similar idea for precomputing actions but we store them in an efficient structure that is not tied to a particular cost function, and in fact, could utilize a new cost function at run time.

There are also many techniques for sequencing multiple pieces of motion data that do not use the graph-based approach. One of the earliest examples was presented by Perlin who used a simple blending approach combined with scripted rules to generate responsive characters [32, 33]. Lamouret and van de Panne present a technique for searching for and extracting the motion sequences that preserve continuity and match the terrain on which the character is moving [22].

In this paper, we focus on real-time controlled motion and in particular, character locomotion. Locomotion is of interest for obvious reasons in video games, virtual environments and any graphical environment that involves human characters. Locomotion requires not only smooth motion but planning for a natural path.

2.2.3. Physically based modeling of human motion

A considerable amount of research has been done in the area of developing physics based approaches to simulate and control human motion. A majority of these approaches are insipired by previous work in robotics and control. Laszlo *et al.* model walking and running as a *periodic limit cycle* [23]. A human character can be considered as a mechanical toy that drives its joints in a repetitive, periodic fashion. However, this open-loop control is insufficient for unstable dynamic motions such as walking and running. They propose a control technique that provides a general method of turning unstable open-loop motions into stable closed-loop motions. Using this procedural method, users can control the global characteristics of human motion such as direction, speed and stride rate.

Van de Panne and Lamouret address the same problem by using guiding forces to allow progressive learning of control actions for balanced locomotion [41]. The guiding forces are external torques applied on the posture to enforce balance during walking or running. The appropriate control actions to produce the desired motion are then progressively learned. Once a basic control strategy for a gait has been synthesized, the guiding force can subsequently removed or eliminated through further optimization. Hodgins and her colleagues provide control algorithms for men and women performing running, bicycling and vaulting [14].

More recently, Faloutsos *et al.* presented a framework for composing individual controllers for performing more complicated tasks [8]. Their framework allows for researchers to integrate various low-level controllers into a single controller for broader functionalities. They demonstrate their framework by creating a *virtual stuntperson* that can balance itself, react protectively using arms in a fall, roll over and get up from a fall.

These physics based approaches have the advantage of being reactive to the environment but it is currently very difficult to develop robust and flexible controllers for physical simulations and they are often computationally expensive. Several others have developed procedural techniques for walking and running [39, 17, 7]. For a good overview of the various approaches for generating character locomotion, see [30].

3. MOTION PREPROCESSING

We use the Biovision hierarchical data format to represent our character motion. Each motion sequence is a collection of poses that can be played back over time. A pose is defined as a single sample of motion capture data and consists of a root node position and orientation, and joint angles for each joint. A pose of a character is represented as a hierarchy of joints, each joint having three degrees of rotational freedom. Our character has 23 joints (Figure 3.1), and therefore a total of 69 degrees of freedom.

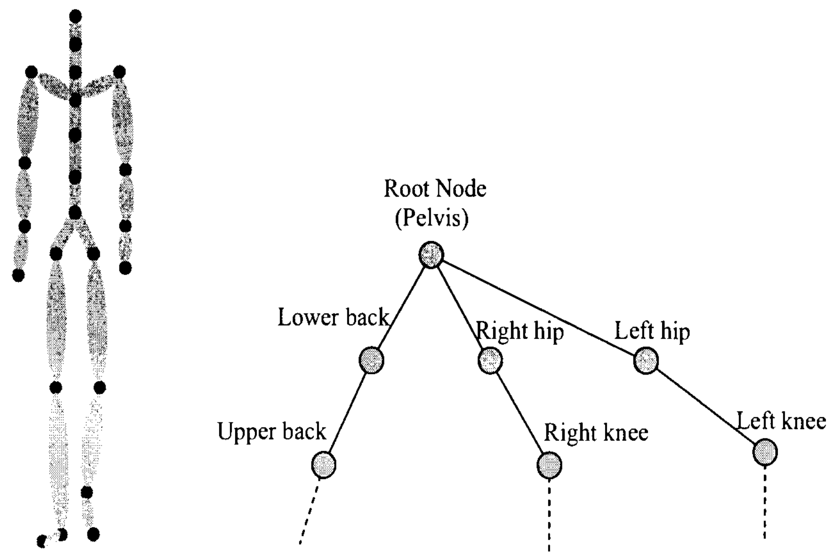


FIGURE 3.1. An articulated figure with 23 joints and its hierarchical representation. The joints have offsets and orientation with respect to their parent and the root node has a translation and orientation, that defines the position and orientation of the character.

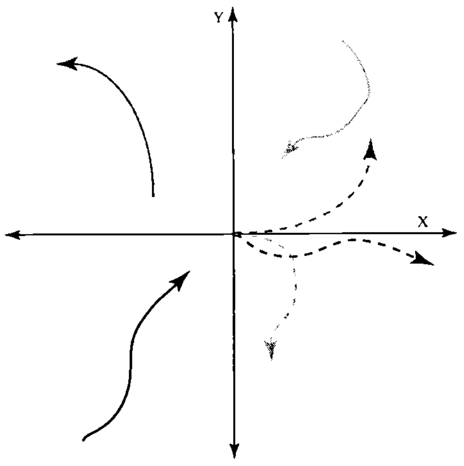


FIGURE 3.2. Original motion sequences (solid lines) are normalized by placing them along a common facing direction, and computing the relative position and orientation at each frame.

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 3.1).

One of the keys to our approach is precomputing much of the motion generation problem. To this end, we build on graph based techniques that sequence individual poses. To facilitate placement of the character at any location with any orientation during motion synthesis, we store poses as relative translation and orientation offsets of the root node from the previous pose.

3.1. Translation and Orientation Offsets

To orient a character along any direction on the floor plane, we must make the orientation of the character independent of its captured orientation

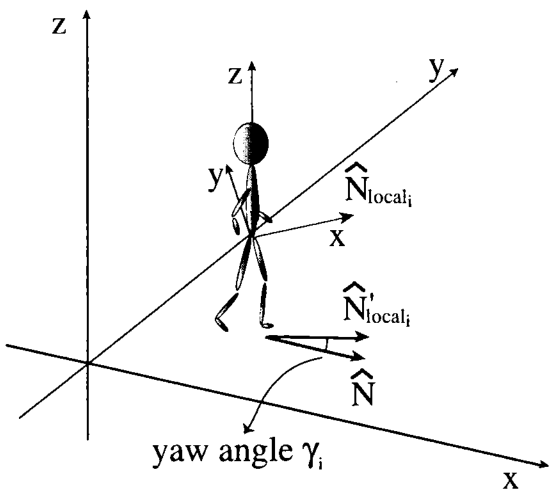


FIGURE 3.3. Computing the facing direction of a character in pose *i*.

(Figure 3.2). We first define the local frame as a right-handed coordinate frame centered on the character's root body, the pelvis, with the positive x-axis pointing out the front of the pelvis and with the z-axis up. Let the facing direction, $\widehat{N}_{local_i}$, be defined as the vector aligned with the local x-axis for pose i. We also choose a reference vector to be the unit vector in the direction of the world x-axis, $\widehat{N}$.

We project the facing direction vector $\widehat{N}_{local_i}$ onto the ground plane and normalize it, resulting in a vector $\widehat{N}'_{local_i}$(Figure 3.3). We then compute the facing angle or the yaw angle, γ_i, to be the angle between $\widehat{N}'_{local_i}$ and $\widehat{N}$.

$$\gamma_i = \arcsin(\widehat{N}'_{local_i} \times \widehat{N}) \tag{3.1}$$

γ_i is used to build the yaw rotation matrix M_{yaw_i}. The orientation of a pose i, without the yaw component, is denoted by $\widehat{R}_i$ and is computed by factoring out the facing direction M_{yaw_i} from its orientation R_i.

$$\widehat{R}_i = M_{yaw_i}^{-1} * R_i \tag{3.2}$$

For motion synthesis, we need to position and orient successive poses, given an arbitrary initial position and orientation. This is possible if we know the relative position and orientation of a pose with respect to the previous pose. Let the recorded world orientation and position of a pose i be denoted by R_i and P_i respectively. We compute the relative orientation R_{rel_i} and relative position P_{rel_i} with respect to pose $i-1$ as follows:

$$R_{rel_i} = R_{i-1}^{-1} * R_i \tag{3.3}$$

$$P_{rel_i} = R_i^{-1} * (P_i - P_{i-1}) \tag{3.4}$$

Given an arbitrary world position P'_{i-1} and orientation R'_{i-1} of pose $i-1$, the new world position and orientation of pose i is

$$P'_i = P'_{i-1} + R'_{i-1} * P_{rel_i} \tag{3.5}$$

$$R'_i = \widehat{R'}_{i-1} * M'_{yaw_{i-1}} * R_{rel_i} \tag{3.6}$$

Our motion preprocessing step involves computing $\widehat{R}_i$, R_{rel_i}, P_{rel_i} from equations (3.2, 3.3, 3.4) for every pose i in our motion database.

At run time, once the character is initialized with a starting pose, position, and orientation, successive poses can be drawn at the correct location with the correct orientation using equations (3.5 , 3.6).

3.2. Pose Transition Graph

One of our goals is to precompute much of the motion generation problem. The first step is to compute a pose-transition graph similar to the variations presented at Siggraph 2002 [19, 24, 2].

We measure the difference, $D_{i,j}$, between two poses i and j considering both joint angles and joint angle velocities as well as the linear velocity of the root node. Joints angles are stored as quaternions and the joint angle difference between two poses is

$$D_{i,j} = \sum_{k=0}^{N} w_k \| log(q_{j,k}^{-1} q_{i,k}) \| \tag{3.7}$$

where $q_{i,k}$ is the k^{th} joint angle quaternion at pose i and w_k is the weight for that particular joint. Same angle difference would be less evident at the wrist joint than at the shoulder joint. This is beacuse of the hierarchical structure of the skeleton. Orientation of a joint higher in the hierarchy affects the orientation of more joints than that of a joint lower in the hierarchy. For this reason, each joint is weighted separately using a heuristic: the further the joint is from the root node, the lower the weight. In particular, we use the weights suggested by Jin Wang and Bobby Bodenheimer [43].

To retain the dynamics of the motion when making transitions, we include velocity terms for the joint angles and the root node. An alternative approach to consider motion dynamics is to compare poses over a window [19]. In our case, including joint angle velocities works well in practice. The joint angle velocity for the kth joint of the ith pose is

$$\dot{\alpha}_{i,k} = \frac{\log(q_{i,k}^{-1}(t) q_{i,k}(t-1))}{\Delta t} \tag{3.8}$$

where Δt represents the capture rate for the motion data (0.033s). The linear root velocity difference, $V_{i,j}$, between two poses i and j is $V_{i,j} = \|V_j - V_i\|$ where V_i is the velocity of the root node at pose i.

Now that we can determine the distance between two poses, we will use this information to determine the cost of transitions between two poses. For

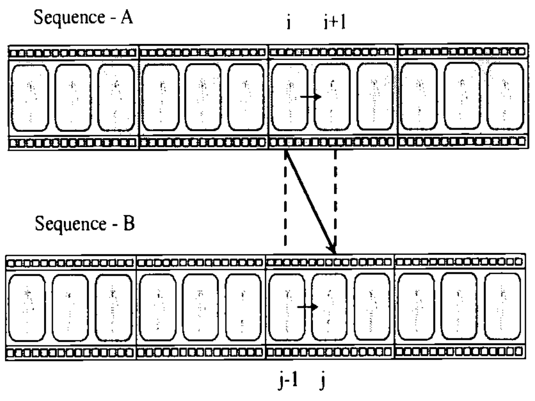


FIGURE 3.4. The transition from a current pose in sequence A to a target pose in sequence B will only be smooth when the next pose in A is similar to the target pose in B and the previous pose for the target in B is similar to the current pose in A.

a transition from one pose i to another pose j to appear smooth, the distance between pose i and $j-1$ should be small and the distance between pose $i+1$ and j should be small (Figure 3.4).

We define the total cost to transition from pose i to pose j in terms of the differences in joint angles, joint velocities, and root node velocity. The joint angle term of the transition cost is

$$\theta_{i,j} = 0.5 * D_{i+1,j} + 0.5 * D_{i,j-1} \tag{3.9}$$

The joint velocity term of the transition cost, $\dot{\theta}_{i,j}$, is computed similarly.

$$\dot{\theta}_{i,j} = 0.5\dot{D}_{i+1,j} + 0.5\dot{D}_{i,j-1} \tag{3.10}$$

$$\dot{D}_{i,j} = \sum_{k=0}^{N} w_k \|\dot{\alpha}_{i,k} - \dot{\alpha}_{j,k}\| \tag{3.11}$$

The root node term of the transition cost is

$$v_{i,j} = 0.5 * V_{i+1,j} + 0.5 * V_{i,j-1} \tag{3.12}$$

The total transition cost between any two poses i and j is then computed as

$$\tau_{i,j} = \omega_\theta * \theta_{i,j} + \omega_{\dot{\theta}} * \dot{\theta}_{i,j} + \omega_v * v_{i,j} \tag{3.13}$$

where ω_θ, $\omega_{\dot{\theta}}$ and ω_v are weight factors for the joint angle, joint velocity and root node velocity terms.

The resulting transition cost matrix is a weighted directed graph where the nodes are the poses and the edges with weights are valid transitions and their costs. As suggested by Lee *et al.* [24], 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 [40]. The remaining components are removed resulting in a single graph where any node can be reached from any other node.

For our implementation, we used a motion capture library with 33,404 poses in the graph. The computation is distributed over a cluster of 48 2.4 Ghz Intel Xeon machines with 1GB RAM each. It takes approximately 40 minutes to compute the pose transition graph.

3.3. Finding Strongly Connected Components

As mentioned above, pruning may introduce dead ends in the graph and there is a possibility that a transition may lead to a pose from which there are no exits. We must remove such transitions from the graph before computing the state-action graph. For a smooth and continuous playback, we do not want to get into such states (Figure 3.5).

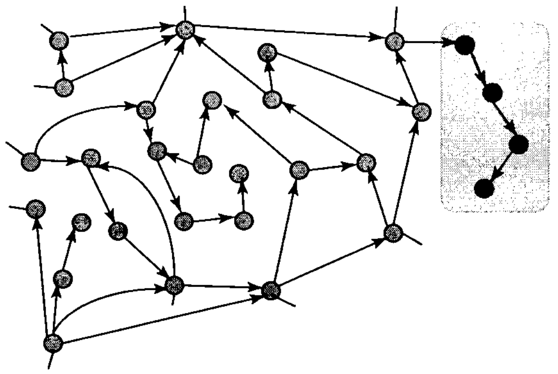


FIGURE 3.5. The directed pose graph can have nodes with no exits. These nodes are dead-ends. Dead-end nodes and nodes that lead to dead-ends are removed from the pose graph.

To be able to reach any node from any other node, we prune the pose-transition graph to remove dead-ends and nodes leading only to dead-ends. We use Tarjan's algorithm to determine the strongly connected graph components [40]. The goal is to generate a single large strongly connected graph. The unconnected components are deemed dead ends and are removed resulting in a single graph where any node can be reached from any other node.

We present the basic idea behind Tarjan's algorithm here. The pseudocode for Tarjan's algorithm is presented in Figure 3.6. It consists of a recursive procedure *VISIT* and a main program that applies procedure *VISIT* to each node that has not already been visited. Procedure *VISIT* enters the nodes of the graph in depth-first order. For each strongly connected component C, the first node of C that procedure *VISIT* enters is called the *root* of component C. The main goal of the algorithm is to find the component roots. For this purpose, we define

```
(1)   procedure VISIT($v$);
(2)   begin
(3)       $root[v]$ := $v$; $InComponent[v]$ := False;
(4)       $PUSH(v, stack)$;
(5)       for each node $w$ such that $(v,w) \in E$ do begin
(6)           if $w$ is not already visited then $VISIT(w)$;
(7)           if not $InComponent[w]$ then $root[v]$ := $MIN(root[v], root[w])$
(8)       end;
(9)       if $root[v] = v$ then
(10)          repeat
(11)              $w$ := $POP(stack)$;
(12)              $InComponent[w]$ := True;
(13)          until $w = v$
(14)  end;
(15)  begin/* Main program */
(16)      $stack$ := $\emptyset$;
(17)      for each node $v \in V$ do
(18)          if $v$ is not already visited then $VISIT(v)$
(19)  end.
```

FIGURE 3.6. Tarjan's algorithm detects the strongly connected components of a graph $G = (V,E)$.

a variable $root[v]$ for each node v. When procedure $VISIT$ is processing node v, $root[v]$ contains a candidate node for the root of the component containing v.

Initially (at line 3) node v itself is the root candidate. When procedure $VISIT$ processes the edges leaving node v (at lines 5-8), new root candidates are obtained from children nodes that belong to the same component as v. The MIN operation (at line 7) compares the nodes with respect to the order in which procedure $VISIT$ has entered them, i.e., $MIN(x,y) = x$ if procedure $VISIT$ entered node x before it entered node y, otherwise $MIN(x,y) = y$. This is implemented by using an array and and a counter to assign a unique depth-first number to each node. When procedure $VISIT$ has processed all edges leaving v, $root[v] = v$ if and only if v is the root of the component containing v (line 9). Note however, that if v is not a component root we do not know if $root[v]$ is the right root of the component containing v.

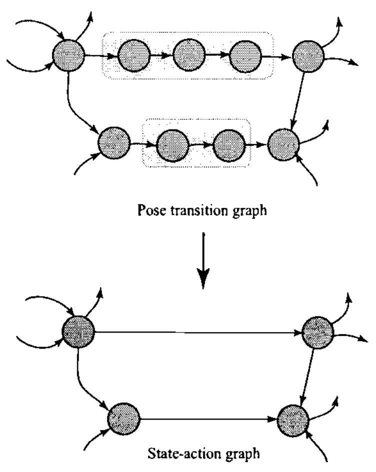


FIGURE 3.7. Computing a state–action graph from the pose transition graph. Consecutive poses with single out–going transitions are collapsed into a single action. States represent poses with multiple out–going transitions.

To distinguish between nodes belonging to the same component as node v and nodes belonging to other components, a boolean variable $InComponent[w]$ is defined for each node w. Its initial value is **False**. When a component C is fully detected, procedure $VISIT$ sets $InComponent[w]$ = **True** for each node w that belongs to C (lines 10-13). A stack is used for this purpose. Each node stored on the stack in the beginning of procedure VISIT. When the component is fully detected the nodes belonging to it are on the top of the stack. Procedure $VISIT$ removes them from the stack and sets the $InComponent$ values to **True**.

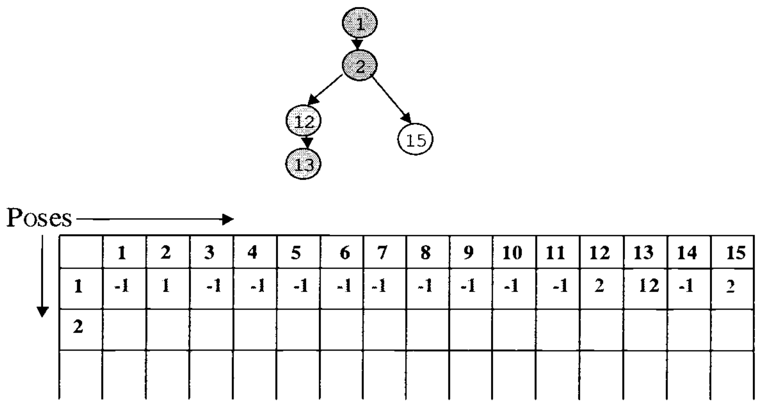
Since Tarjan's algorithm requires that the entire graph be in memory, this algorithm cannot be run in parallel over multiple machines. For our sparse graphs, it took approximately 7 minutes to compute the strongly connected components of the 33,404 node graph. All components of less than 60 nodes were discarded from the original graph. In our experiments, the algorithm found one large connected component with 30,665 poses.

As Lee *et al.* [25] noted in their recent work, the pose transition graph has relatively few poses with multiple out-going transitions. Most of the poses have single out–going transitions. We collapse consecutive poses with single out–going transitions into an action (Figure 3.7). This results in a "state-action" graph in which a state represents a pose with multiple out-going transitions. Taking an action results in a transition from one state to another. The cost of taking an action is the sum total cost of consecutive pose transitions that make up the action. The resulting state–action graph has 15,659 states.

3.4. Representing All-Possible Smooth Sequences

Given the state-action graph, a shortest path between any pair of states defines a sequence of state-actions steps that guarantee smooth playback. Ideally we would like to take this graph and run queries for shortest-path between arbitrary pairs of states (poses) to ensure smooth playback. Previous methods have used such a graph to globally search for motion sequences (graph-walks) that meet some user constraint, such as a ground trajectory. Rather than use the state-action graph directly, we precompute the shortest path between all pairs of states in the state-action graph and store the paths in an all pairs shortest paths matrix (APSP).

We use Dijkstra's algorithm to compute the single-source shortest paths for every state. The running time of Dijkstra's algorithm is $O(N + E)$, where N is the number of states in the graph and E is the number of edges (actions) in the graph. Since our graph is large (about 15,659 states), we use the bucket-sort



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	12	-1	2
2															

FIGURE 3.8. Consider the simple shortest path tree rooted at node 1. The corresponding entry for node 1 is shown in the partial matrix. The path from node 1 to 15, for example, is found by starting at node 15 and following the chain backward to 2 then 1.

implementation of Dijkstra's algorithm [29]. Bellman-ford's principle allows us to optimize the storage of the APSP trees. If the shortest path between node i, and node j is through node k, then the shortest path between i and k, and the shortest path between k and j form the shortest path between i and j. We can take advantage of the inherent redundancy in the paths represented by the APSP matrix to store it in $O(n^2)$ space. We store the APSP trees in a single n x n matrix. At entry i,j, if there is a path from i to j, we store the number of the state that is the last step in the path from i to j. Using this chaining approach, we can easily extract the sequence of states that lead from i to j by following the chain in reverse order. Figure 3.8 shows a simple example.

The result of this step is a matrix that encodes the most natural action sequences the character can take to move from one state to another given the library of recorded data at hand. This graph guarantees smooth motion between any two pairs of states assuming bad transitions have been pruned out in the creation of the original pose transition matrix. The APSP graph computation requires approximately 60 minutes for our library of data.

Given the APSP matrix our approach will focus on choosing the appropriate pairs of states to generate action sequences that meet the user constraints. These poses will serve as our indices into the APSP matrix. In the following chapters, we will discuss how we choose target poses that meet particular user constraints. In Chapter 4, we discuss a method for representing control by computing reachable spatial locations from evey state. Chapter 2.2 combines motion generation and control using a run-time algorithm for motion synthesis.

4. REPRESENTING CONTROL USING MOBILITY MAPS

The All-Pair-Shortest-Path (APSP) matrix transforms the motion-synthesis problem into one of selecting sequences of nodes or *graph walks* having the least total cost. It encodes the various ways that we can re-assemble clips to create natural and smooth motion. The matrix can be queried for a smooth sequence $S_{f_i \to f_j}$ between any two states (poses) f_i and f_j. $S_{f_i \to f_j}$ represents smoothest sequence of actions between states f_i and f_j. This sequence of actions may result in other intermediate states between f_i and f_j.

We are now in a position to consider the problem of finding motion that satisfies user-specified requirements in real-time. In our case, the user is interested in extracting sequences from the APSP matrix that enables a character to reach a specified location on the floor. From now on, we shall use the term "state" and "pose" interchangeably.

The APSP matrix alone cannot be queried for sequences that satisfy this constraint. This is because the query can only be in terms of pairs of states (f_i, f_j). However, the user constraint is specified in terms of a 2-D spatial location, P_{floor}, on the floor. In other words, if the character is in state f_i, we need to find a target state f_j that results in a sequence $S_{f_i \to f_j}$ that takes the character towards P_{floor}. To be able to do this, we need to know the resulting position and orientation of the character on the floor plane if it takes the action sequence $S_{f_i \to f_j}$. In other words, we need to map every action sequence from f_i to a resulting position and orientation on the floor plane. This mapping would allow us to determine, at every state, what the character is capable of achieving spatially.

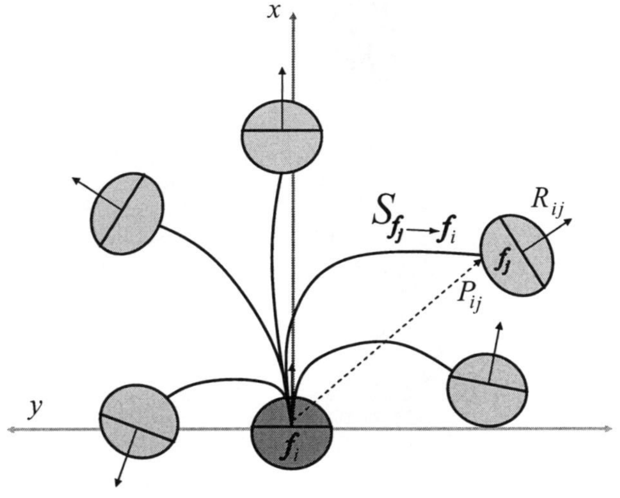


FIGURE 4.1. The mobility map for a particular state stores all of the states that can be reached within a fixed number of state-action steps. It facilitates control because it relates the user's spatial input to the character's spatial capability from f_i.

4.1. Constructing the Mobility Map

The mobility map is a novel data-structure that encodes what the character is capable of achieving spatially from a given state, and the corresponding target states it can get to. This enables us to select an appropriate target state based on where it takes the character on the floor.

We construct the mobility map from the APSP matrix in the following way: For each state f_i, we search the APSP matrix for a set of all states $\mathbf{F}_i$=$\{f_1, f_2, f_3, ..., f_{k_i}\}$ such that

$$n(S_{f_i \to f_j}) = \Gamma, \forall f_j \in \mathbf{F}_i \tag{4.1}$$

where $n(S_{f_i \to f_j})$ represents the number of states in the state-action sequence $S_{f_i \to f_j}$, and $k_i = n(F_i)$. The idea is to compile a list of poses that can be reached from each pose in our motion-graph, within some fixed number of state-action steps (Γ). We choose Γ to be 25 because it gives a reasonable lookahead distance for determining where the character is heading and in practice has proven to be a good window. For each sequence $S_{f_i \to f_j}$, $\forall f_j \in \mathbf{F}_i$, we also compute and store the orientation (R_{ij}) and the position (P_{ij}) of f_j relative to f_i by placing f_i at the origin, facing along the x-axis and evaluating the relative translation and orientation changes for each pose in the sequence (Figure 4.1). This can be accomplished by using equations (3.5, 3.6).

4.2. Pruning the Mobility Map

Each entry in the mobility map is a state f_i followed by a list of target states F_i. Each state in F_i is a terminal state of a motion sequence that ends at a particular location, and at a particular orientation with respect to f_i. The more the spatial locations the character can reach from f_i, the more controllable the character will be. This forms the motivation for removing certain states f_i for which the number of states in F_i is below a certain threshold. We choose the threshold to be 30, since it has practically proven to provide optimum control of the character in our experiments.

A mobility map can be considered as a directed graph with every node f_i having neighbors in F_i. For this reason, pruning the mobility map may introduce dead-ends. So we find the strongly connected components as in section 3.3. This results in a new mobility map with a set of states f_i such that every state has at least 30 terminal states in F_i.

4.3. Using the Moility Map

The mobility map for every pose f_i stores $\{P_{ij}, R_{ij}, S_{f_i \rightarrow f_j}, \mathbf{F}_j\}$. Now given the position and orientation of a pose f_i of the character in the environment, we can use this data-structure to quickly evaluate its resultant position and orientation within the next few states. Hence the name *Mobility Map.* The mobility map can be viewed as a coarse representation of the APSP matrix. Rather than store all sequences that lead from a pose to all other poses, it stores all spatial locations that can be reached from a single pose within some small time frame. This coarse map can be efficiently searched to find the appropriate target pose because it stores spatial locations that can be compared to the spatial constraints of the user. This mobility map is similar to that introduced by Brogan and Hodgins as a method for controlling physical simulations at a coarse level of detail [5].

The mobility map takes approximately 20 minutes to compute and has 6550 states after pruning. For the examples in this paper, the mobility map requires approximately 25MB of memory. Through experimentation with our locomotion dataset, we have found that on average, a state can reach approximately 38 target states in 25 state-action steps.

At this point, we have precomputed the mobility map. For each state, this map stores every state that can be reached within 25 state-action steps and the corresponding state-action sequence. We can now discard the APSP matrix and the original transition matrix. At run time, a search through the action alternatives from the current state will result in a target state and a subsequent state-action sequence to that state.

5. COMBINING MOTION AND CONTROL

To summarize our approach, we now have a way of generating smooth motion from any pose and know what the character is capable of achieving spatially from the same pose, by looking at the mobility map (Chapter 4). We put these structures together to form our locomotion generation pipeline. The run time algorithm uses this pipeline to generate interactive and controllable motion. In the next following sections, we describe this run time algorithm.

5.1. Locomotion pipeline

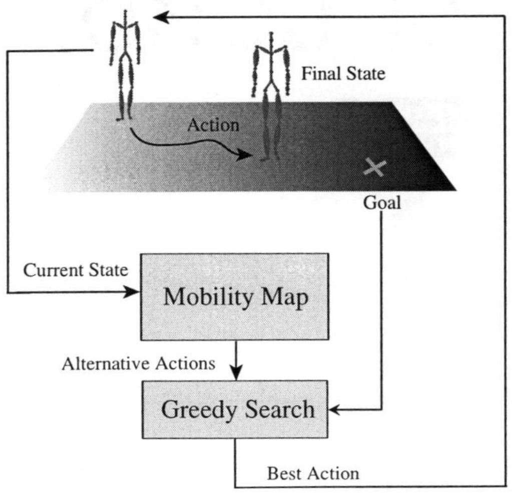


FIGURE 5.1. Our run time locomotion generation. The current state of the character is used to index into the mobility map to obtain a list of action alternatives over the next few states. A greedy search based on a user defined cost function is used to pick the best action that takes the character closest to the goal.

At run time, a constraint is provided in the form of a desired target goal. The current state (pose) of the character is used to index into the mobility map

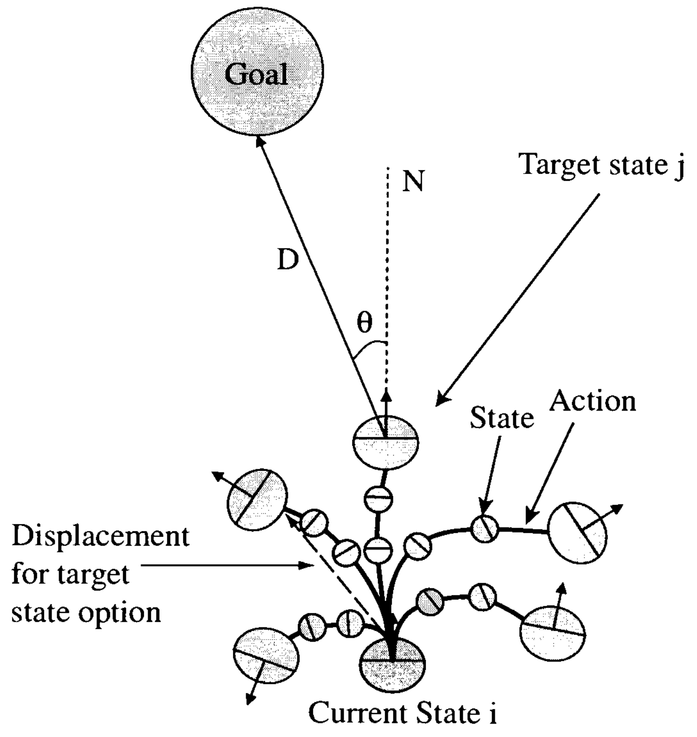


FIGURE 5.2. Choosing the best sequence from a particular pose.

to retrieve a list of possible action alternatives from that state. A greedy search is then performed over these alternatives, to pick the one that gets the character closest to the goal. This process is repeated at each state of the avatar to make a constant progress towards the goal.

5.2. Target tracking: Searching for target poses.

The goal of the run-time search component is to choose state-action sequences that meet user constraints given the character's current configuration. The character's current configuration includes its state, position and orientation. The user constraint is provided as a desired location on the ground plane. When a desired location request is received, the system indexes the mobility map with the character's current state to determine all the action alternatives that are available from that state. It then searches these alternatives for the best option where best is determined by a cost function. The cost function for our examples is of the following form.

$$CostToGoal = \omega_D * D + \omega_\theta * \theta \tag{5.1}$$

Let i represent the current state and j be one of many possible target state options that can be reached from i in 25 state-action steps. D is the Euclidean distance between the goal location and the character's projected location if it chooses target state j. θ is the deviation angle (measured in degrees), for the state j. The deviation angle is the angle between the character's resulting facing direction if it were to follow the sequence for state j and the vector from the character's position to the user's desired location if it were to follow the sequence for state j (Figure 5.2).

Each of the elements of the cost function has an associated weight as well. For all examples in this paper, the cost function weights ω_D and ω_θ were set to 310.0 and 17.0 respectively in equation 5.1. However, these weights could be tuned interactively at run-time. The state-action sequence corresponding to the chosen target state is extracted from the mobility map at run-time and stored in

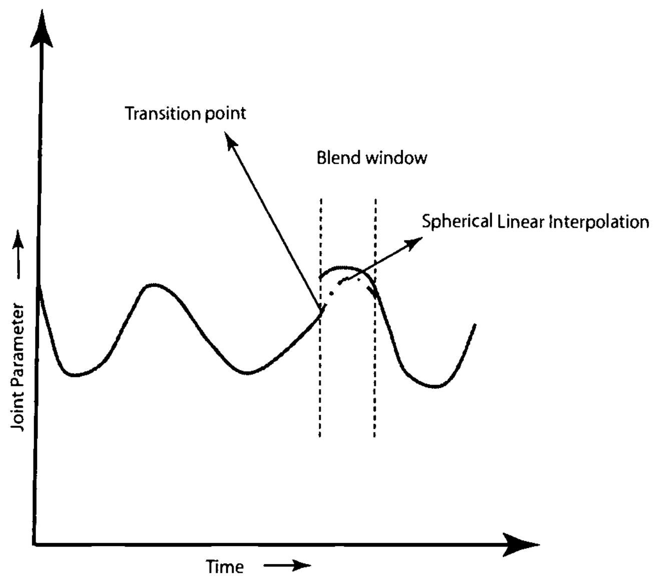


FIGURE 5.3. A one-sided spherical linear interpolation of an angle parameter.

a buffer. The display function then accesses the state-action sequence and plays the motion.

It is important to note that responses to the user direction requests are not limited to the end of the 25 step state-action window. A decision is computed whenever the character is in a state from which it has action alternatives. If a new user direction is provided while a state action sequence is being carried out, the algorithm flushes the buffer and computes a new state-action sequence starting from the current state. Otherwise, a new sequence replaces the buffer only if it takes the character closer to the goal than the sequence currently in the buffer. As a result, at every decision point (or state) if the user input changes or a better sequence is found, a new sequence is chosen that makes progress toward the goal.

The running time of the mobility map search is bounded by the largest number of displacement options from any single pose in the mobility map. In theory, since the average number of options for each pose in the mobility map is approximately 38, the local search is constant time on average. In practice, the number of options from a single state may increase with the total number of poses in the dataset.

5.3. Ensuring smooth transitions.

Recall that we prune the transition matrix in an early step to remove poor transitions. Although our mobility map consists of sequences of poses that represent the smoothest possible motion given our data, there are still cases where a small C_0 discontinuity is visible. To remedy this, we perform a one sided blend (Figure 5.3). We have to do a one sided blend because our motion synthesis is online, and we do not know when a transition is expected in the sequence. In other words, cannot do a blend over a window across the transition point.

When we are at a pose where a transition into a different sequence is about to happen, we look ahead several frames (about 5-10) into the next sequence. We then perform an interpolation from the current pose to that pose. The position of the root node is interpolated linearly, while the orientation of the root node is interpolated using spherical linear interpolation. We also use spherical linear interpolation for the joint angles. If the motion is interrupted during an interpolation, we choose the pose nearest the interpolated pose as the current pose and continue with a new lookup. This interpolation eliminates any discontinuities.

6. RESULTS

We summarize and discuss our results in this chapter. For the examples presented in the following sections, we used a library of motion capture data consisting of 114 sequences for a total of 33,404 frames of data. The data was captured with the Vicon 612 3D optical motion capture system with six high-resolution 1000 Hz digital M-cameras, 64 channels of analog-to-digital input, and real-time data processing capabilities [42]. Our capture area was approximately 15ft.$\times$10ft. We captured an actor performing various locomotion behaviors such as walking straight at various speeds, turning at various radii, coming to a stop, starting from a stop, standing in place, and turning in place. The data was then cleaned and converted to joint angle data using the Vicon Workstation and the BodyBuilder software. Real-time demonstrations were generated on an Intel Xeon 1.7 MHz processor with 1.0G RAM. We demonstrate the performance of our algorithm with three locomotion based examples that are described in the following sections. Each demonstration contains real-time characters controlled at a high-level by the user.

6.1. Target Tracking

To demonstrate a character's target tracking ability, we control the navigation of a character through a maze (Figure 6.1). This example was inspired bythe controllable video sprite fish presented by Schödl *et al.* [38]. The user is given control over a yellow square cursor on the ground plane. The character then moves to this location using the run-time procedure described above. The user can continuously specify new target positions.

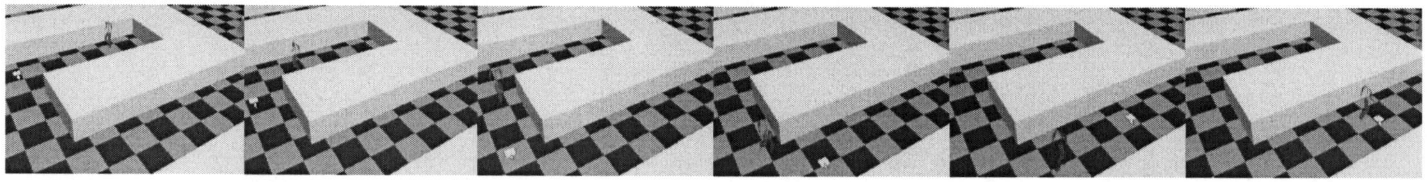


FIGURE 6.1. An example of target tracking. The character is shown tracking the yellow square target. The user can continuously change the target location while the character attempts to walk closer to it.

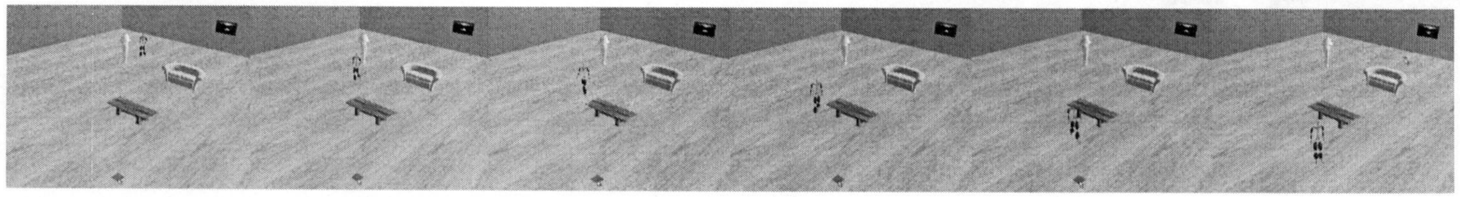


FIGURE 6.2. An example of obstacle avoidance. The character is attempting to reach the user specified target while avoiding obstacles in the environment. Each obstacle is approximated with a bounding cylinder. The character successfully chooses a sequence of poses that move it around the obstacle.

The character's path is not predetermined in any way. Rather, given a start location and desired end location, the character finds a path to the goal that is determined completely by the available motion options and the cost function. In essence, the path taken by the character is completely determined by it's current pose and the sequences available to it. Because there are many poses for which there are very few options to choose from, the resulting path is not always direct. The character sometimes wanders slightly, but eventually reaches the target destination. Wandering occurs under two conditions. First, there are some states that may have poor spatial options to choose from, even if there are a large number of action alternatives. Second, the character may be stuck in a relatively long state-action sequence before reaching a state with more action alternatives. We have found out that on average, every fourth pose is a state with multiple choices.

6.2. Obstacle Avoidance

For this demonstration, we implemented a simple steer to avoid behavior that chooses target locations that avoid collisions with cylindrical obstacles (Figure 6.2). As the character walks in the living room environment, it maintains a five-second lookahead to determine if there are any obstacles in its immediate path. If a potential collision occurs, the system computes a new target location to avoid the collision and take the character around the obstacle (Figure 6.3).

The new target location is computed using simple planar geometry. If the line along the lookahead vector intersects the obstacle at points A and B, then a temporary steer to avoid target is a point P such that $OP \perp AB$ and $\|OP\| = 2r$, where r is the radius of the cylinder representing the obstacle. The character

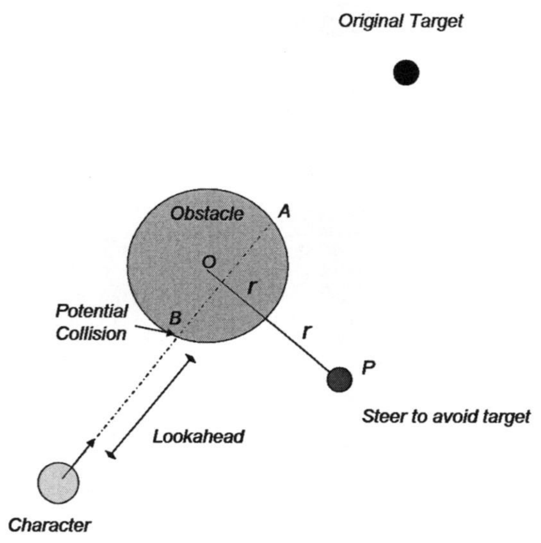


FIGURE 6.3. When a potential collision occurs, the steer to avoid behavior computes a new target location that avoids the obstacle. When the character nears the temporary target area, it is discarded and tracking of the original target resumes.

now tracks the new steer-to-avoid target, rather than the original target. Once the character gets within some threshold distance to the steer-to-avoid target, the target is switched back , so that the character now tracks the original target.

6.3. Crowds

One of the most compelling aspects of our approach is that the generation of motion involves only local searches at run time, not a global optimal search. As such, our approach scales well with the number of characters. Each character can use the same mobility map in memory and thus each new character added to

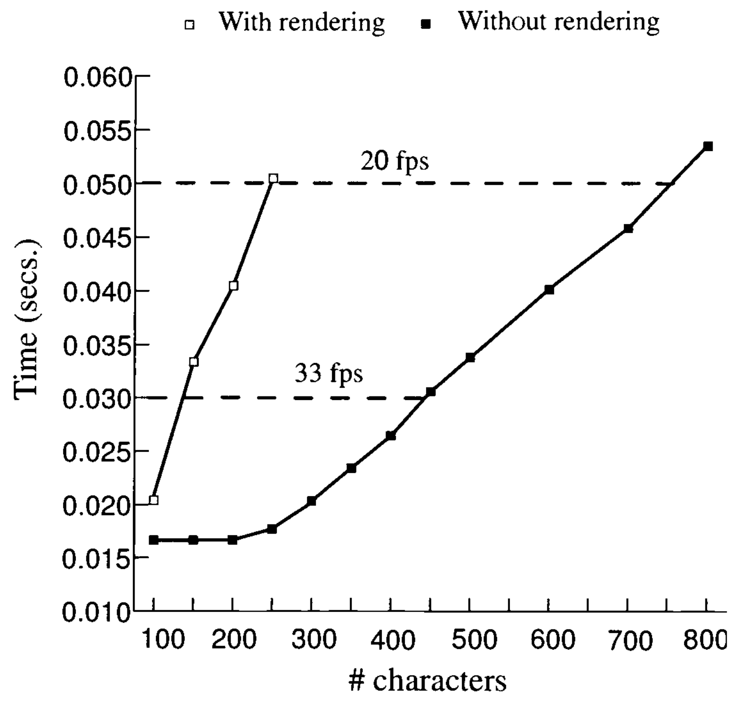


FIGURE 6.4. Scalability of our algorithm. The figure shows a plot of *per frame* processing time (in seconds) versus number of characters, with and without rendering. Our run time algorithm can render approximately 150 characters at 30 frames per second.

the scene results in only an added local search. This means that the "update" time for a scene with multiple characters in it should linearly increase with the number of characters in the scene. The "update" time for a scene only involves the time for performing a local search in the mobility map, and does not include the rendering time.

In our experiments, without rendering or obstacle avoidance, our approach can update the positions of up to 500 target tracking characters at approximately

30 frames per second (Figure 6.4). With rendering turned on, our run-time algorithm can update the positions of approximately 150 characters at 30 fps.

7. DISCUSSION AND FUTURE WORK

We have presented a framework for generating natural, scalable locomotion in real time that is controllable at a high level. We have demonstrated the results for single characters tracking targets as well as for crowd scenes of up to a hundred and fifty characters. We have also presented the algorithmic complexity of the steps in our approach. In the following sections, we discuss some of the problems and issues concerning our framework, and provide an insight into possible future directions opened up by this work.

7.1. Discussion

Due to the nature of our approach, the character locomotion is generated at two levels - articulated motion and path planning. The articulated motion is a direct result of graph walks in the APSP matrix. The path planning is a result of the combination of the mobility map and the greedy local search. The character's path to the goal is determined solely by the cost function and exactly what the character is capable of achieving from the current pose.

We have demonstrated that given a target goal location in the environment, the character can successfully find the target. Through experimentation, we have found that goals placed too close to the character can lead to problems because our data set does not include enough maneuvers for tight spaces. We have found that a target distance between 3 and 6 meters works very well for guiding the character with finer control while a larger target distance of greater than 10 meters leads to more organic paths determined primarily by the data in the graph. We have also demonstrated that the character is reactive enough to respond to simple collisions using a steer-to-avoid collision response algorithm. Finally, we have demonstrated scalability by showing a rendered large scene of 150 characters still running in real time and by experimenting with up to 500 characters whose motion can be computed in real-time without rendering.

Like Lee *et al.*, our approach precomputes action sequences from every state [25]. However, instead of storing only the optimal action for a particular behavior and a particular constraint, we store all possible state-action sequences from a state. Given a new behavior constraint, such as desired walking speed or desired task (stop, walk, run etc.), there is no need to learn a new policy table. Rather our approach would only require a new cost function that incorporated these constraints. We would continue to use the same mobility map. Scheduling and combining multiple cost functions is left as future work.

Although we have only presented examples of locomotion, we believe this approach can easily be extended to other domains. Clearly, the pose transition graph and APSP matrix can be built for any library of data. The mobility map we have presented is specific to locomotion behavior. However, one could imagine building a more general, capability map, that maps some other behavior to target poses. For example, imagine that we want to build punching behaviors for a boxing game. The capability map could easily store, for each pose, the target locations for character punches or what behaviors the character was capable of given the current pose. The user input could be defined as desired punch locations. Given a cost function that measured the distance between two punch targets, a search could be performed to choose the appropriate target punch pose given the current pose. These two poses would then be used to index into the APSP matrix. This approach is applicable whenever the user input can be expressed as some physical value that can be derived from the motion sequences, such as positions.

Annotations can also be used in our approach. For example, all poses can be annotated with the type of sequence they belong to. User input might include tasks such as run, jump, fall, etc. Although we did not demonstrate it in any of our examples, we successfully annotated walking and standing data and were able to incorporate them into our cost function. In doing so, we could choose to reward standing poses in certain situations and walking poses in others.

7.2. Future work

Although we have demonstrated successful motion generation in several examples,there are several areas for improvement in our approach. First, although memory is cheap, it is not infinite. We are exploring methods for reducing memory requirements by further compressing the mobility map data structure. Another problem is that in our current implementation we synthesize small portions of transitions, but we do not synthesize any motion sequences via blending or interpolation. This means that we must capture all data necessary to perform the required tasks for the application. Unfortunately, there are some cases where the character just does not have the appropriate data examples to draw from in order to move appropriately. We are investigating methods for filling in the sparse motion areas of the graph automatically as well as techniques for on-the-fly motion synthesis. Motion capture based techniques require the capture of a large amount of motion data. We are also interested in exploring methods for determining when one has enough data and if not, how to identify the types of data needed.

Finally, we would like to explore techniques for improving the responsiveness of the character. In our current experiments, there are some cases where the character just does not have the appropriate data examples to draw from.

For example, we have found that goals placed too close to the character can lead to problems because our data set does not include enough maneuvers for tight spaces. This could be due to a lack of options from a particular state. On the other hand, there could be many options, but those options may not have a good spatial distribution. We are investigating methods for utilizing the mobility map to identify sparse motgion areas of the graph and recognize the types of motion needed. We are also interested in finding better ways to build a mobility map. For example, rather than choosing states with many action alternatives, we could instead choose states with a good spatial distribution of action alternatives. Finally, motion capture based techniques require the capture of a large amount of motion data. We are also interested in exploring methods for determining when one has enough data in the mobility map.

This work provides an insight into the problems involved in developing controllers for character animation driven by motion capture data. Typically, an end-user requires character motion to be re-parameterized by a user specified constraint. In this work, an attempt has been made to re-parameterize character motion by user-specified 2D locations on the floor. As discussed above, this approach is severely limited by the number of options available from a particular pose in the mobility map. Consider for instance a "perfect" mobility map where every node has sufficient options to allow the character to get to a reasonable number of places from the current pose. This would increase the reactivity of the character to user-inputs, and provide us with a near perfect controller that can direct the character anywhere on the floor plane. Unfortunately this would require infeasible amounts of data to be recorded and stored.

One of the future directions aims at achieving this goal. One way of augmenting the mobility map with more options is interpolating the existing options to create more options. This can be achieved by building a registration curve through all the options for a particular pose and varying the blend weights to get the desired distribution of sequences from the current pose [18].

To solve storage problems, we are looking into ways of representing joint angle trajectories and root node trajectories using a Linear Dynamic System (L.D.S). Such a representation will help us build models of human motion, and allow us to compactly represent motion-capture data.

Another area where the concept of mobility map can be applied is in augmenting 2D pedestrian simulation. Pedestrian simulation is built around standard techniques like flocking behaviors, social forces models or cellular automata models, to name a few. Unfortunately these models govern crowd behavior based on standard scripted rules or on point-mass physics. This approach is not realistic because it doegs not take into account what an actual individual is capable of doing, from a particular "state". A "state" may include, but is not limited to, information such as the current pose and orientation of the individual. The mobility map provides us with this information. This structure can be used along with a standard crowd simulation strategy to produce realistic crowd scenarios.

This thesis is but a small step towards realizing the goals of creating believable, controllable and scalable character motion. Much work has to be done in addressing the issues discussed above, and in improving the robustness of the approach.

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

[4] 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.

[5] D. Brogan and J. Hodgins. Simulation level of detail for multiagent control. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 199–206. ACM Press, 2002.

[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] 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.

[8] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In Eugene Fiume, editor, *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 251–260. ACM Press / ACM SIGGRAPH, 2001.

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

[10] 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.

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

[12] 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.

[13] Michael Gleicher. Comparing constraint-based motion editing methods. *Graphical models*, 63(2):107–134, 2001.

[14] J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien. Animating human athletics. In Robert Cook, editor, *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 71–78. Addison Wesley, August 1995.

[15] Tae hoon Kim, Sang Il Park, and Sung Yong Shin. Rhythmic-motion synthesis based on motion-beat analysis. *ACM Trans. Graph.*, 22(3):392–401, 2003.

[16] Eugene Hsu, Sommer Gentry, and Jovan Popovic. Example-based control of human motion. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation.* ACM Press, 2004.

[17] H. Ko and J. Cremer. Vrloco: Real-time human locomotion from positional input streams. In *Proceedings of Presence '96*, volume 5, pages 367–380, 1996.

[18] L. Kovar and M. Gleicher. Flexible automatic motion blending with registration curves. In *In Proceedings of Eurographics/SIGGRAPH Symposium on Computer Animation (2003).*

[19] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. *ACM Transactions on Graphics*, 21:473–482, 2002.

[20] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 214–224. Eurographics Association, 2003.

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

[22] A. Lamouret and M. van de Panne. Motion synthesis by example. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 199–212. Springer-Verlag New York, Inc., 1996.

[23] J. F. Laszlo, M. van de Panne, and E. Fiume. Limit cycle control and its application to the animation of balancing and walking. In Holly Rushmeier, editor, *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 155–162. Addison Wesley, 1996.

[24] 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.

[25] Jehee Lee and Kang Hoon Lee. Precomputing avatar behavior from human motion data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation.* ACM Press, 2004.

[26] Jehee Lee and Sung Yong 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.

[27] 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.

[28] 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.

[29] Eric N. Mortensen. Vision-assisted image editing. *Computer Graphics*, 33(4):55–57, November 1999.

[30] F. Multon, L. France, M. Cani-Gascuel, and G. Debunne. Computer animation of human walking: a survey. *The Journal of Visualization and Computer Animation*, 10(1):39–54, 1999.

[31] Sang Il Park, Hyun Joon Shin, and Sung Yong 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.

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

[33] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In Holly Rushmeier, editor, *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 205–216. Addison Wesley, 1996.

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

[35] K. Pullen and C. Bregler. Motion capture assisted animation: Texturing and synthesis. *ACM Transactions on Graphics*, 22, 2002.

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

[37] 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.

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

[39] H. Sun and D.N. Metaxas. Automating gait generation. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 261–270. ACM Press, 2001.

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

[41] Michiel van de Panne and Alexis Lamouret. Guided optimization for balanced locomotion. *Computer Animation and Simulation '95*, pages 165–177, 1995.

[42] Vicon. Vicon 612 optical motion capture system. http://www.vicon.com, 2004.

[43] Jing Wang and Bobby 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.

[44] D. J. Wiley and J. K. Hahn. Interpolation synthesis for articulated figure motion. In *Proceedings of IEEE Virtual Reality Annual International Symposium*, pages 156–160, March 1997.

[45] 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.