Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2005-11-10

Real-Time Motion Transition by Example

Cameron Quinn Egbert
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Egbert, Cameron Quinn, "Real-Time Motion Transition by Example" (2005). *All Theses and Dissertations*. 429.
https://scholarsarchive.byu.edu/etd/429

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu.

REAL-TIME MOTION TRANSITION BY EXAMPLE

by

Cameron Egbert

A thesis submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2005

Copyright © 2005 Cameron Egbert

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Cameron Egbert

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

___________________________	___________________________
Date	Bryan Morse, Chair
___________________________	___________________________
Date	Parris Egbert
___________________________	___________________________
Date	Michael Jones

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Cameron Egbert in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

__________________________ ________________________________

Date

Bryan Morse
Chair, Graduate Committee

Accepted for the Department

____

Parris K. Egbert
Graduate Coordinator

Accepted for the College

____

G. Rex Bryce
Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

REAL-TIME MOTION TRANSITION BY EXAMPLE

Cameron Egbert

Department of Computer Science

Master of Science

Motion transitioning is a common task in real-time applications such as games. While most character motions can be created a priori using motion capture or hand animation, transitions between these motions must be created by an animation system at runtime. Because of this requirement, it is often difficult to create a transition that preserves the feel that the actor or animator has put into the motion. An additional difficulty is that transitions must be created in real-time. This paper provides a method of creating motion transitions that is both computationally feasible for interactive speeds, and preserves the feel of the original motions. To do this, we build the transition from both a procedural motion and a motion segment taken from the motions being transitioned between.

Table of Contents

1. Introduction.. 1
   1.1 Background.. 1
   1.2 Statement of the Problem.. 2
   1.3 Thesis Statement.. 4

2. Motion Transitioning Methods.. 5
   2.1 Current Methods.. 5
   2.2 Approach Presented in This Thesis 7

3. Real-Time Motion Transition by Example.. 9
   3.1 Abstract.. 9
   3.2 Introduction.. 9
   3.3 Related Work 12
   3.4 Real-Time Motion Transition by Example.. 16
      3.4.1 Finding Transition Points.. 17
         3.4.1.1 Dynamic Timewarping………………………………………… 18
         3.4.1.2 AligningMotions…………………………………………………...20
         3.4.1.3 Searching for an Example…………………………….....................21
         3.4.1.4 Motion Modification……………………………….....................22
   3.5 Experiments and Results.. 25
   3.6 Discussion and Further Work 28

4. Summary and Conclusions 31

5. Bibliography 33

List of Figures

Figure 1: The distance array for two similar motions. 20

Figure 2: A motion signal and its Laplacian pyramid decomposition. 23

Figure 3: The result of motion modification. 25

Figure 4: Computation time in seconds of determining a timewarp 26

Figure 5: Number of frames in the test motions. 26

Figure 6: Walking to kicking 27

Figure 7: Walking to jumping 27

Figure 8: Jumping to skipping 28

Figure 9: Kicking to jumping 28

Chapter 1

Introduction

1.1 Background

Realistic character motion is a necessity for computer graphics applications such as movies and games. Three main methods exist to create motion for a virtual character- *motion capture*, *hand animation*, and *simulation*. *Motion capture* is the process of recording the motion of a live human actor. *Hand animation* refers to the use of a software package to manipulate a 3D model of a character over time in order to achieve an animation. In *simulation*, the motion of the character is computed using a physical model.

Simulation methods are generally thought to be too unrealistic for games. In addition, since an interactive application, such as a game, must generate animation on the fly, and both *motion capture* and *hand animation* produce pre-made motion, these methods can only be used if further processing is done.

The method most commonly used in games is to create several base motion segments (i.e., walking, running, jumping, etc.) using *motion capture* or *hand animation*, and transition between these motions on the fly [Mizuguchi et al. 2001; Gleicher et al. 2002].

Most current animation systems use skeletal animation. Each model has both a skin and a skeleton. The skin is made up of a polygonal mesh, and this mesh is what is seen in the final rendering. The skeleton is made up of a hierarchy of bones, which

define the position of the skin. Moving the skeleton, therefore, moves the skin of the model; much like a human skeleton moves a human body. Usually, a large number of skin vertices will map to very few skeleton bones. This makes it so that only the relatively few skeleton bones must be manipulated in order to move the model, instead of moving each vertex of the model's skin.

A motion is a series of transformations for each of the bones of the skeleton. Each bone transformation is expressed in terms of Euler rotations of each of the three axes, **X**, **Y**, and **Z**. These three orientations can also be expressed in terms of a single quaternion, which removes the ambiguities of the Euler rotation representation. In addition, each bone has a constant length. A motion for a particular skeleton, then, is defined as a continuous function:

$$\mathbf{M}(f) = (\mathbf{p}_R(f), \mathbf{q}_1(f) \ldots \mathbf{q}_n(f))$$

where $\mathbf{p}_R(f)$ is the position of the root, and $\mathbf{q}_i(f)$ is the orientation of the ith bone. Each parameter vector $\mathbf{M}(f)$ is called a *frame*, and f is called the *frame index*. A transition is a motion that links two other motions together.

1.2 Statement of the Problem

The goal of this research has been to find a method to create plausible transitions for interactive applications. Since the point of transition is not known ahead of time,

these transitions must be created dynamically. This puts more severe constraints on the transitioning method than would be needed for an offline method.

Ideally, the method of creating the transition should have the following properties, in order of decreasing necessity:

1. Computing the transition should be efficient enough to run in real time.
2. Transitioning should be responsive.
3. The transitioning method should not require excessive space resources (disk space, memory, etc.).
4. The motion created should be continuous and believable.

These are the basic criteria for any algorithm that creates a transition. The first three are hard constraints for real-time applications. If the algorithm doesn't meet these requirements, it is of no use. The fourth is a softer constraint and is somewhat subjective. At the very least, the algorithm should produce a motion for which C^1 continuity is preserved for the position and rotation of the joints.

We have produced a method that is feasible for real-time applications and improves on the motion quality available from existing transitioning methods. While current methods either sacrifice interactivity or motion quality, our method preserves both. We do this by using a Laplacian pyramid decomposition to warp animation information from the two motions being transitioned between in order to create the transition. This preserves the same "feel" of the original motions in creating the transition, while being computationally efficient enough for interactive rates.

1.3 Thesis Statement

Believable motion transitions can be created by modifying pre-stored motion using a Laplacian pyramid decomposition. In addition, these transitions can be computed at interactive rates, are responsive to user input, and don't require extraordinary space resources, making the algorithm feasible for real-time applications.

Chapter 2

Motion Transitioning Methods

2.1 Current Methods

A brief overview of a few methods currently in use for creating transitions will now be given. For a more in depth review of these methods, refer to Chapter 3.

Currently, the most widely used method of creating transitions involves linearly interpolating between two motions. A pre-determined number of frames at the end of the first motion are overlapped with the first frames of the second motion, and the values of each are linearly interpolated, creating a smooth transition between the motions. Unfortunately, this transition may not be realistic, especially in the case of extremely dissimilar motions. Even when the motions are similar, the problem of synchronizing motions is not addressed using this method alone. Transitioning between two walking motions that are at different points in their cycle will give an unrealistic transition, even though the motions are similar.

This problem is addressed by using *dynamic timewarping* [Bruderlin and Williams 1995; Kovar and Gleicher 2003]. *Dynamic timewarping* creates a function that synchronizes both motions to be at similar poses at any given time by first determining the similarity of each pair of frames for both motions. A distance function is used to determine this similarity. Then the synchronization function is determined by finding the best path through these similarity values. The computation of this function can be done as an offline step by creating a lookup table. This table is then referenced at runtime to

synchronize the two motions. While timewarping alleviates the problem of unsynchronized motions, it doesn't address the problem of two dissimilar motions.

Other methods for creating transitions have been proposed. Rose et al. [1996], use spacetime constraints to create transitions. In their method, a combination of dynamic and kinematic constraints is placed on the skeleton, and a transition is generated using these constraints. This method gives realistic motion for short transitions (between about 0.3 and 0.6 seconds) but is not computationally efficient enough for real-time applications.

Kovar, et al. [2002] use a method they call a Motion Graph which is a way of arranging motion data into a graph. Traversing this graph gives a new motion. Each node of the graph corresponds to a common pose, such as standing. Traversing an edge corresponds to playing a short motion segment between two poses. In order to determine an entire motion, a traversal through the graph between two particular nodes is computed using a set of constraints, after which the motion is played by traversing the graph node by node. Motion graphs are not suitable for interactive applications because of the computation time needed to find a traversal of the graph.

An extension to Motion Graphs, called Snap-together motion [Gleicher et al. 2003], processes a corpus of motion into a graph similar to a motion graph. Snap-together motion differs from Motion Graphs primarily in the way that it produces motion. Instead of computing an entire traversal through the graph, each edge traversal is determined one by one at run-time from the user's input. In this way, the graph can be used for real-time applications. The downfall of this method is that once an edge is

taken, no further input can be given until the motion reaches the next node. While this is sufficient for some real-time applications, in general, more interactivity is required.

2.2 Approach Presented in this Thesis

This thesis proposes a method that will be feasible for real-time applications, and give more believable motion than a simple linear transition. Specifically, the motion for the transition is adapted from a segment of motion from one of the two motions being transitioned between. This segment can be chosen to resemble any specific motion. This ability is leveraged to choose a segment that resembles the desired transition. The segment chosen is then warped to match this transition even more closely. In this way the method enables the synthesis of a transition that preserves the same "feel" of the original motion while producing a motion that is feasible as a continuous transition.

The remainder of the thesis will be presented as follows: Chapter 3 is a paper describing the research done that will be submitted for publication, and Chapter 4 contains a summary and conclusions.

Chapter 3

Real-Time Motion Transition by Example

3.1 Abstract

Motion transitioning is a common task in real-time applications such as games. While most character motions can be created a priori using motion capture or hand animation, transitions between these motions must be created by an animation system at runtime. Because of this requirement, it is often difficult to create a transition that preserves the feel that the actor or animator has put into the motion. An additional difficulty is that transitions must be created in real-time. This paper provides a method of creating motion transitions that is computationally feasible for interactive speeds and that preserves the feel of the original motions. To do this, we build the transition from both a procedural motion and a motion segment taken from the motions being transitioned between.

3.2 Introduction

Realistic character motion is becoming a necessity for real-time applications such as games. As real-time rendering technology is advancing to the point of photo-realism, more and more attention is being placed on realistic motion techniques, especially in the case of virtual characters.

One basic motion technique required in real-time applications is transitioning. Typically, a set of base motions (i.e., walking, running, jumping, etc.) is created by an animator or by motion capture, and then transitions are created to link these motions together. Because it is not known beforehand when the character will need to transition, and because it is not feasible to create every possible transition ahead of time, transitions are usually created at run-time [Mizuguchi et al. 2001; Menache, 2000].

We propose the following criteria to measure the effectiveness of methods used for creating realistic motion transitions, in order of decreasing necessity:

1. Computing the transition should be efficient enough to run in real time.
2. Transitioning should be responsive.
3. The transitioning method should not require excessive space resources (disk space, memory, etc.).
4. The motion it creates should be continuous and believable.

First, and most importantly, the transitioning algorithm must be efficient enough to be computed in real time. This is a hard constraint. If the algorithm doesn't run in real time, it is of no use to an interactive application. This constraint is somewhat of a moving target, as compute power is constantly increasing. However, an algorithm that runs in exponential time will likely not achieve real-time performance regardless of the available compute power.

Second, transitioning should be responsive. Transitioning should start as an immediate response to some user input. In other words, latency should be minimal. If

the user presses a jump button, the virtual character should jump immediately, not after a delay. For a few applications this is not as necessary, but in general this is a requirement of real time applications.

Third, the transitioning method should not require extraordinary space resources. For example, an algorithm that pre-computes every possible transition and stores it on disk to be looked up when needed, while computationally efficient at run-time, would constitute an algorithm that requires excessive resources. Current real-time environments have space and resource limitations, some of which are quite restrictive. Thus, the technique should be such that it can meet these restrictions without sacrificing the other system goals. This constraint is also a moving target, as resource constraints become less severe with each new generation of hardware.

Fourth, the motion it creates should be believable. For example, in a transition from a walking animation to a running animation, when a foot touches the ground, it shouldn't move. The transition should also look natural, like something the character would do. At the very least, the motion should be continuous from the end of the first motion to the beginning of the second motion. In particular, at least C^1 continuity should be preserved for the joint orientations. Believability is the softest constraint, since believability is a subjective term, and perfectly believable motion is still an unsolved problem. However, reasonable believability is necessary.

Motion for a particular skeleton is defined as a continuous function:

$$\mathbf{M}(f) = (\mathbf{p}_R(f), \mathbf{q}_1(f) \ldots \mathbf{q}_n(f))$$

where $\mathbf{p}_R(f)$ is the position of the root, and $\mathbf{q}_i(f)$ is the orientation of the ith joint. Each parameter vector $\mathbf{M}(f)$ is called a *frame*, and f is called the *frame index*. A transition is a motion that links two other motions together in a continuous fashion.

3.3 Related Work

One of the most widely used methods of creating real-time transitions involves linearly interpolating between two motions [Menache, 2000; Mizuguchi et al. 2001]. A pre-determined number of frames at the end of the first motion are overlapped with the first frames of the second motion, and the values of each are linearly interpolated, with interpolation values decreasing from 1.0 to 0.0 for the first motion, and increasing from 0.0 to 1.0 for the second motion. This creates a smooth transition such that the first frame of the transition is a frame from the first motion and the last frame is a frame from the second motion. Formally, if $\mathbf{M}_0$ is the first motion, which has m frames, $\mathbf{M}_1$ is the second motion, which has n frames, and $\mathbf{T}$ is the desired transition, which has u frames, then

$$\mathbf{T}(x) = \mathbf{M}_0(m - u + x) * (1 - x/u) + \mathbf{M}_1(x) * (x/u)$$

As an example, for a 10 frame transition between motions $\mathbf{M}_0$, and $\mathbf{M}_1$, the first frame of the transition would be $1*\mathbf{M}_0(n\text{-}10) + 0*\mathbf{M}_1(0)$, the next frame would be $0.9*\mathbf{M}_0(n\text{-}9) + 0.1*\mathbf{M}_1(1)$, and the last frame would be $0*\mathbf{M}_0(n) + 1*\mathbf{M}_1(10)$. In practice, this interpolation is usually done as a spherical linear interpolation on the orientation quaternions, as opposed to simply interpolating each degree of freedom.

This method is efficient, responsive, and requires no space other than that required for the original motions. The downfall of this method is that it doesn't always produce realistic motion. The motion is synthesized from two motions that may or may not be similar (a walk to a jump). Even if they are similar, they may not be synchronized. For example, a walk with left foot forward to a run with left foot forward will cause the transition to look like a half step from left foot forward to both feet at neutral, then back to left foot forward, instead of left foot forward, neutral, then right foot forward.

One proposed method of synchronizing the motions is to use *dynamic timewarping* [Bruderlin and Williams 1995; Kovar and Gleicher 2003]. *Dynamic timewarping* creates a function that synchronizes both motions to be at similar poses at any given time. This is usually done by computing a distance function between each pair of frames for both motions, which gives a 2-dimensional distance array, and then searching for a path through this array that follows the valleys. While it is not trivial to compute the distance metric between each frame of both motions for the distance array, this can usually be done as an offline step. The timewarp can then be computed from this at runtime with minimal overhead using a dynamic programming solution. This method does introduce significant storage consumption, as a distance array has to be stored for each pair of motions. Thus, the array will require $O(n^2)$ storage. *Timewarping* also doesn't address the problem of two dissimilar motions.

Park et al. [2002] use the idea of *dynamic timewarping* to align clips of motion before interpolating between them. In addition, the motion clips are parameterized to provide a method for controlling the synthesized motion. If these interpolations are parameterized correctly, a large amount of control can be had over the animation. Their

approach allows for specification of locomotion over a range of directions and speeds. Unfortunately, this method is geared toward generating motion from a set of similar motions, and not between two arbitrary (possibly different) motions.

Other methods for creating transitions have been proposed. Rose et al. [1996] propose a method of generating transitions using spacetime constraints. In their method, a combination of dynamic and kinematic constraints is placed on the skeleton, and a transition is generated using these constraints. This method gives realistic motion for short transitions (between about 0.3 and 0.6 seconds) but is not computationally efficient enough for real-time applications.

Physically-based motion synthesis is another method of synthesizing motion [Hodgins et al. 1995; Liu et al. 2002]. In these methods, Motion is generated from a dynamic simulation of the character. It is prohibitively difficult to produce realistic motion using physically-based approaches except for a few special cases. Additionally, the level of realism is often proportional to processing complexity, further hindering the achievement of realism. As processing complexity is lowered to the critical point of real-time performance, motion quality degrades. These approaches also fail to capture the nuances of human motion. The motion is physically valid, which is a good first step, since human motion is also physically valid, but the motion produced lacks the feel of a living character. Humans are noisy creatures, whose motion is never exactly repeatable, while simulations give "perfect" motion that can be recreated exactly.

Another approach is to construct a mathematical model from a set of motion capture data. Hidden markov models [Brand et al. 2000] and switched linear dynamic systems [Li et al. 2002] are among the most popular approaches. These methods can

produce arbitrary motion that resembles the pre-existing cache of motion capture data, but at the cost of low control and high processing requirements.

Another approach, Motion Graphs [Kovar et al. 2002], converts a corpus of motion data into a graph representation, which gives new motion when traversed. Nodes in the graph correspond to frames that are similar in two or more motions (local minima of the distance array), and edges correspond to motion segments in-between these frames. A motion is created by calculating a traversal through the graph given a set of constraints. Motion graphs are not suitable for interactive applications because of the computation time needed to find a traversal of the graph.

An extension to Motion Graphs, Snap-together motion [Gleicher et al. 2003] processes a corpus of motion into a graph similar to a motion graph. The main difference between Motion Graphs and Snap-together motion is the process of creating motion. Instead of calculating an entire traversal through the graph, each edge traversal is determined one at a time at run-time from the user's input. For example, a simple Snap-together motion graph might consist of three nodes: one corresponding to a standing pose, another corresponding to a crouching pose, and the last corresponding to a kneeling pose. The graph also has three edges, one between each pair of nodes. Suppose the character starts in the standing pose. If no input is given, the character will keep standing. If the user tells the character to crouch, the edge from the standing pose node to the crouching pose node will be taken, and the corresponding motion played. From this pose, the character can then go back to crouching, or to kneeling. A walking motion would be an edge from the standing node that loops back to the standing node after two steps. Snap-together motion is a step in the direction of interactivity, since the transitions

between nodes are taken as the user gives input, instead of computing the entire path through the motion graph. However, this approach doesn't quite achieve full responsiveness. Once an edge is taken, i.e., a motion segment has started, the motion of the character can't be interrupted until reaching a node. This is sufficient for some applications but in general does not produce enough responsiveness.

In another graph-based approach, Arikan and Forsyth [2002] applied a randomized algorithm to search for motions from a hierarchy of transition graphs. In later work, Arikan et al. [2003] created a motion by using a similar graph structure but satisfied user-specified annotations in the creation of the resulting motion. When the number of example motions becomes too large, it becomes prohibitively time-consuming to search through these graph structures for a suitable motion.

Pullen and Bregler [2002] propose a method of using motion capture to assist an artist in creating an animation. In their method, the artist creates a rough animation using conventional key-framing, and motion capture data is used to enhance the animation in order to make it look more lifelike. Part of our method leverages this research in making the final transition more natural.

3.4 Real-Time Motion Transition by Example

In order to produce a method of creating transitions that meet the four proposed goals, we propose a new method that is both feasible for real-time applications and produces more believable motion than a simple linear transition. The "feel" of the motion is

preserved by adapting pre-existing motion from the two motions being transitioned between.

This approach at producing real-time motion transitions is encapsulated in a 4-step process:

1. Find transition points.
2. Align motions.
3. Search for an example.
4. Motion modification.

In the following discussion, $\mathbf{M}_0$ is the motion that is being transitioned from, $\mathbf{M}_1$ is the motion that is being transitioned to, $\mathbf{T}$ is the transition, and t is the length of the transition (in frames).

3.4.1 Finding Transition Points

First, since the start of the transition is a frame from $\mathbf{M}_0$, and the end of the transition is a frame from $\mathbf{M}_1$, transition points are found for $\mathbf{M}_0$ and $\mathbf{M}_1$. The transition point for $\mathbf{M}_0$ is the frame at which the transition is initiated. For example, if the character is on frame 10 of a walking animation when the user initiates the transition, the transition point for $\mathbf{M}_0$ is frame 10. The transition point of $\mathbf{M}_1$ is either set manually, or found using a method similar to *dynamic timewarping*. Motions that should be played from start to finish (jumping, kicking, punching, etc.) have their "transition to" frame set

manually to the first frame, while the transition point for two similar motions (walking to running) is computed using *dynamic timewarping*. These transition points are kept in a lookup table for use at runtime.

Keeping a lookup table of the matching frames requires storage space to hold the frame of each motion that could be transitioned to, for each frame in each motion. Therefore, if there are n motions, and each motion has m frames, the space required to store these values is $n*m*n$. Typically a character will have up to 50 motions, at about 200 frames per motion. Since each value of a table is a frame index, these values can be stored in 1 byte, which requires 500,000 bytes (479 kB) to store all of the tables, which is not an excessive space requirement. At run-time, the necessary value is simply read from this lookup table. Since we want the transition to look natural, we find the transition point in $\mathbf{M}_1$ which matches what the frame from $\mathbf{M}_0$ would have been had there been no transition. For example, if a 30 frame transition is initiated on a walk cycle when the left foot is forward, and at the end of 30 frames, the character would have had its right foot forward, we want to transition to a frame in $\mathbf{M}_1$ that is similar to the right foot forward pose. Specifically, given frame i in $\mathbf{M}_0$, to find the correct "transition to" frame j in $\mathbf{M}_1$ (after a transition of length t), just read the lookup table value for $\mathbf{M}_0(i+t)$, instead of simply $\mathbf{M}_0(i)$.

3.4.1.1 Dynamic Timewarping

To determine a timewarp, we use the same distance metric as in [Kovar et al. 2002]. Specifically, to compute the distance between two frames $\mathbf{F}_i$ and $\mathbf{F}_j$, two point

clouds representing each frame are compared. The point clouds are created from the joint positions of the skeleton. In order to take into account derivative information, a small neighborhood of frames about $\mathbf{F}_i$ and $\mathbf{F}_j$ are used to create the point clouds. Finally, the optimal sum of squared distances is computed between the two point clouds, allowing for rigid 2D transformations. The distance metric is defined as:

$$D(F_i, F_j) = \min_{\theta, x_0, z_o} \sum_k w_k \left\| p_{i,k} - T_{\theta, x_o, z_0} p_{j,k} \right\|^2$$

where $\mathbf{p}_{i,k}$ is the kth point in the cloud generated from frame i and $T_{\theta,x0,z0}$ is a linear transformation consisting of a rotation of θ degrees about the vertical axis followed by a translation of (x_0, z_0). $\mathbf{w}_k$ are weights that sum to one and give more importance to $\mathbf{F}_i$ and $\mathbf{F}_j$, and less importance to the frames at the edges of the neighborhoods.

This has the following closed form solution:

$$\theta = \arctan \frac{\sum_i w_i (x_i z_i' - x_i' z_i) - (\overline{x}\overline{z'} - \overline{x'}\overline{z})}{\sum_i w_i (x_i x_i' - z_i z_i') - (\overline{x}\overline{x'} - \overline{z}\overline{z'})}$$
$$x_0 = (\overline{x} - \overline{x'} \cos\theta - \overline{z'} \sin\theta)$$
$$z_0 = (\overline{z} - \overline{x'} \sin\theta - \overline{z'} \cos\theta)$$

where $\overline{x} = \sum_i w_i x_i$ and the other barred terms are similar.

This distance metric is calculated for each pair of frames, which produces a distance array. Figure 1 shows an example distance array for the *weak kick* and *strong kick* actions.

Matching frames are calculated from this array. The idea is to create a minimum cost connecting path through the array, and use this path to determine which frames best match. This path is determined by walking through the array one frame at a time, choosing one of the neighbors of the current position as the next step in the path. The neighbor chosen is the neighbor with the least cost value. The path is also restricted to be continuous, causal (i.e., to always move forward), and to have a slope limit (i.e., a limit to the number of consecutive horizontal or vertical steps). The slope limit is somewhat arbitrary, but in practice a slope limit of 3 steps works well.

This path is calculated for every possible starting point, and the path that yields the minimum average cost is saved. From this path, the matching frames are determined.

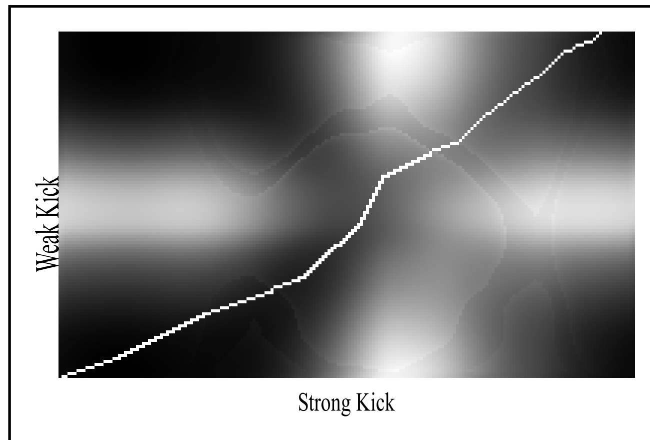

Figure 1: *The distance array for two similar motions. The white line represents the minimum cost path connecting frame 0 and frame n of the weak kicking motion.*

3.4.1.2 Aligning Motions

Second, after the transition points for the motions are determined, $\mathbf{M}_1$ is aligned to $\mathbf{M}_0$. The starting position of $\mathbf{M}_1$ is found from the Newtonian motion formula:

$$\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{v}*t + \tfrac{1}{2}\,\mathbf{a}*t^2$$

where $\mathbf{p}_1$ is the starting position of $\mathbf{M}_1$, $\mathbf{p}_0$ is the position of the final frame of $\mathbf{M}_0$, $\mathbf{v}$ is the velocity of the final frame of $\mathbf{M}_0$, t is the time length of the transition, and $\mathbf{a}$ is the constant acceleration needed to achieve the velocity of the starting frame of $\mathbf{M}_1$ in the time of the transition. The rotations of the root joint of $\mathbf{M}_1$ are found in a similar way.

3.4.1.3 Searching for an Example

At this point the endpoints for the desired transition are known, and hence we are ready to create the transition. In order to preserve the "feel" of the motion, a segment of either $\mathbf{M}_0$ or $\mathbf{M}_1$ is used to build the transition. The third step of creating the transition is to find this segment. Both $\mathbf{M}_0$ and $\mathbf{M}_1$ are searched to find the motion segment that most closely matches the desired transition according to a "closeness" metric. The metric we use is a measure of change in value from the start of the transition to the end of the transition, and the velocity at both endpoints. Specifically,

$$\mathbf{C} = (\mathbf{m}_0 - \mathbf{m}_{0\mathrm{TARGET}})^2 + (\mathbf{m}_1 - \mathbf{m}_{1\mathrm{TARGET}})^2 + (\mathbf{ds} - \mathbf{ds}_{\mathrm{TARGET}})^2$$

where $\mathbf{m}_0$ is the slope of the start of the motion segment, $\mathbf{m}_1$ is the slope at the end of the motion segment, $\mathbf{ds}$ is the change in value of the motion segment, and $\mathbf{m}_{0\mathrm{TARGET}}$, $\mathbf{m}_{1\mathrm{TARGET}}$, and $\mathbf{ds}_{\mathrm{TARGET}}$ are the values of the desired transition. The number of frames between $\mathbf{m}_0$

and $\mathbf{m}_1$ equals the number of frames between $\mathbf{m}_{0TARGET}$ and $\mathbf{m}_{1TARGET}$. In other words, time scaling is disallowed.

For each degree of freedom, the motion segment that produces the minimum value for **C** is used as the example segment in creating the final transition. This process is repeated for each degree of freedom.

3.4.1.4 Motion Modification

The final step is to modify the motion to resemble the desired transition. The previous step yielded a motion segment that roughly matches what the transition should be at the endpoints. This is necessary for the transition to be continuous with the original two motions, but so far no constraint has been made for the motion between the endpoints. What is really desired is a motion that behaves relatively well but looks like what the character would have done if it had chosen the transition. In other words, we want to control the general motion yet have it resemble the pre-existing motions. In order to accomplish this, we construct the motion from both a smooth transition and the example motion. High frequency information, which gives the motion its character, is taken from the example motion, while low frequency information is taken from the smooth transition. The signal is reconstructed from this frequency information into the final signal.

In order to accomplish this, we use a Laplacian pyramid decomposition [Burt and Adelson 1983], first introduced to motion signal processing in [Bruderlin and Williams 1995]. Laplacian pyramids provide a method of breaking up a signal into different

frequency bands, any of which can be replaced before reconstructing the signal. This allows for modification of the low frequency information, while preserving the high frequency information, thus preserving the "feel" of the motion. Figure 2 shows an example of a 3-level Laplacian pyramid decomposition of a signal.

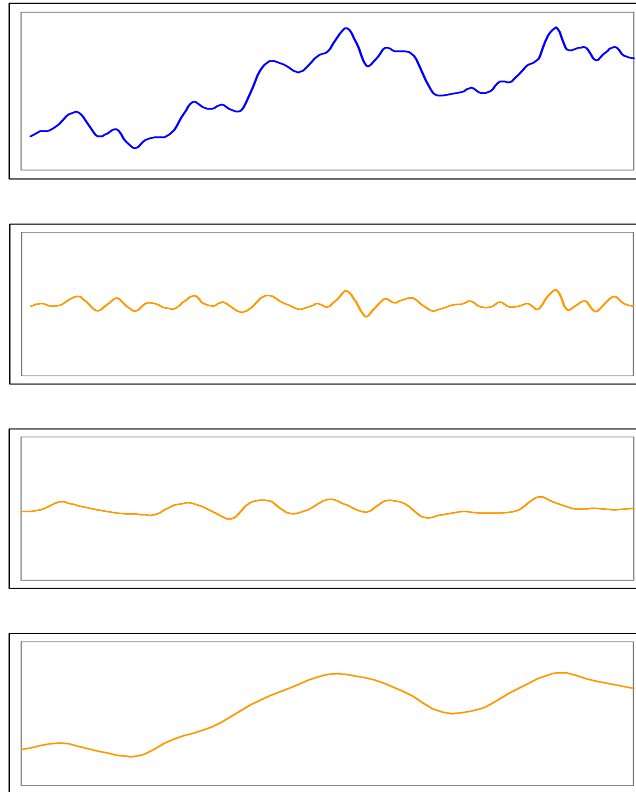

Figure 2: *A motion signal (top), and its 3 level Laplacian pyramid decomposition*

A Laplacian pyramid of a signal is constructed as follows:

1. Call the original signal $\mathbf{V}_0$.
2. Downsample (scale down) the signal to create a signal with half the number of values, and label it $\mathbf{V}_1$ (call this the reduce operation).

3. Upsample (scale up) and linearly interpolate the values of $\mathbf{V}_1$ to create a signal with the same number of values as $\mathbf{V}_0$ (call this the project operation). This signal has lost some of its higher frequency information, and looks smoother than $\mathbf{V}_0$.
4. Subtract this signal from $\mathbf{V}_0$. The resulting image, $\mathbf{L}_1$, represents the difference (or error) between the original signal and the downsampled image.
5. Repeat this process to produce $\mathbf{V}_2$ and $\mathbf{L}_2$ from $\mathbf{V}_1$, and so on for as many levels as desired. The original signal and the downsampled signals (i.e., $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$, and $\mathbf{V}_3$) form the levels of a multi-resolution pyramid. The signals representing the differences between adjacent levels of the multi-resolution pyramid (i.e., $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$) form the levels of a Laplacian pyramid.

Each level of the Laplacian pyramid can be thought of as containing frequency information for the signal, where $\mathbf{L}_1$ contains the highest frequencies.

Now, for each degree of freedom of each joint in the transition, the new motion segments are decomposed using a Laplacian pyramid, and the lowest level is replaced by a 3rd degree Bezier curve that is C^1 continuous with both $\mathbf{M}_0$ and $\mathbf{M}_1$. The signal is then reconstructed from the Laplacian pyramid to give a function which transitions with C^1 continuity from the end of $\mathbf{M}_0$ to the beginning of $\mathbf{M}_1$ while having the same "feel" as $\mathbf{M}_0$ and $\mathbf{M}_1$. Figure 3 shows an example of a signal that has been modified using this method.

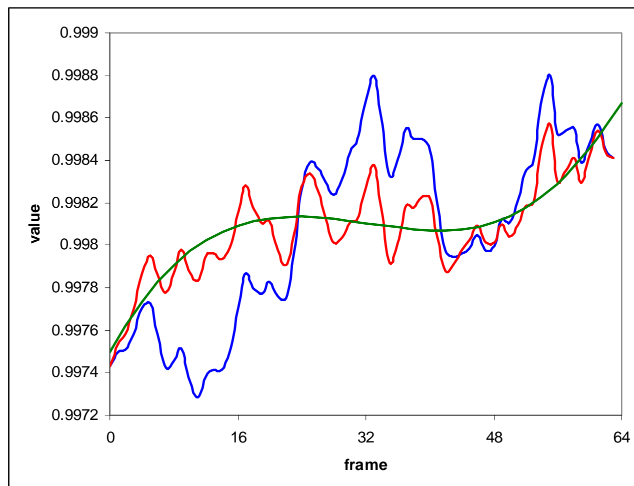

Figure 3: *The result of motion modification. The original signal (blue) is modified by replacing the third level of its Laplacian pyramid decomposition with the Bezier curve in green. The final signal (red) is then reconstructed.*

The level to which the signal is decomposed before substitution and reconstruction can vary. Substitution at the first level is equivalent to using none of the sample signal, while substitution at higher levels introduces more and more of the sampled signal. Practice has shown that substitution at about the third level usually produces the best results.

3.5 Experiments and Results

The method has been tested on a set of seven motions, some similar, others dissimilar. Transition points for similar motions were computed using *dynamic timewarping*, while transition points for dissimilar motions were manually set to the start frame of the motion. Table 1 shows the time in seconds of computing the timewarp between a pair of motions. Table 2 shows the number of frames in each motion.

Motion Number	1	2	3	4	5	6	7
1	x	14.36	23.89	18.47	20.04	24.02	49.03
2	14.62	x	14.11	11.187	12.21	14.39	30.03
3	26.28	14.78	x	19.51	19.41	23.30	46.52
4	19.19	11.40	19.07	x	14.96	18.13	35.88
5	20.18	11.37	19.12	14.90	x	18.37	42.63
6	24.99	21.08	35.32	25.42	27.97	x	64.12
7	68.88	39.69	67.84	53.16	54.30	65.51	x

Figure 4: *Computation time in seconds of determining the timewarp between two motions. The "from" motions are in the rows, and the "to" motions are in the columns.*

Motion	1	2	3	4	5	6	7
Number of Frames	201	116	193	149	158	188	378

Figure 5: *Number of frames in the test motions*

Transitions between motions were created at runtime in response to user input. Calculating the transitions is virtually instantaneous and caused no noticeable delay in frame rate. Figures 6–9 show the results of the method in creating a few different transitions.

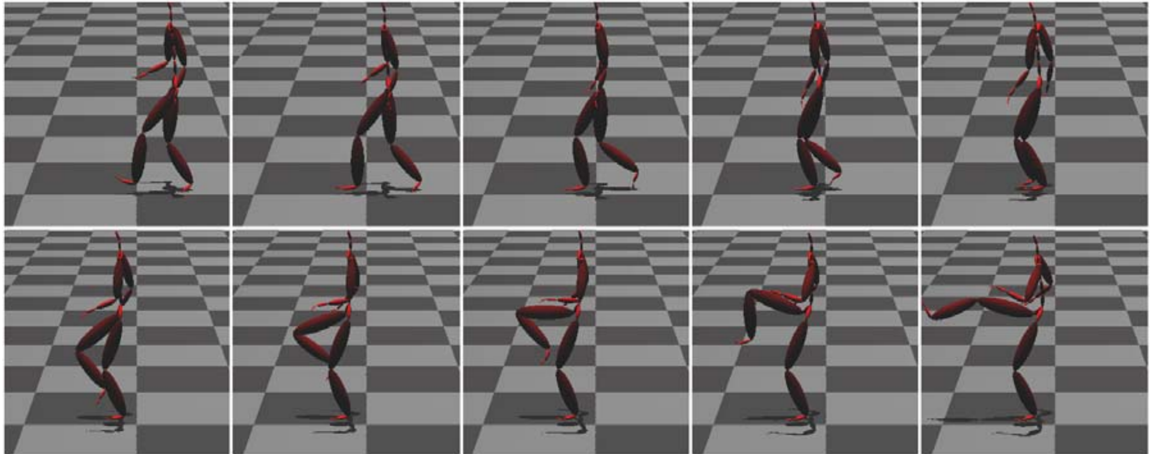

Figure 6: *Walking to kicking*

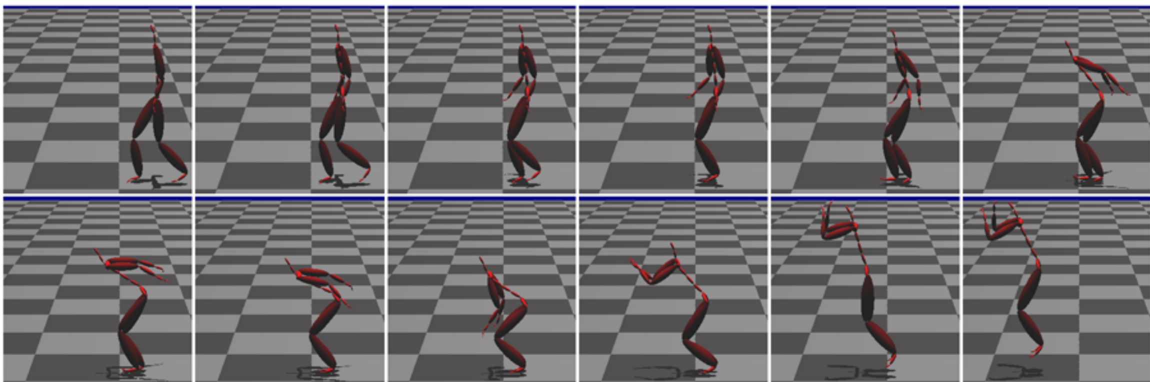

Figure 7: *Walking to jumping*

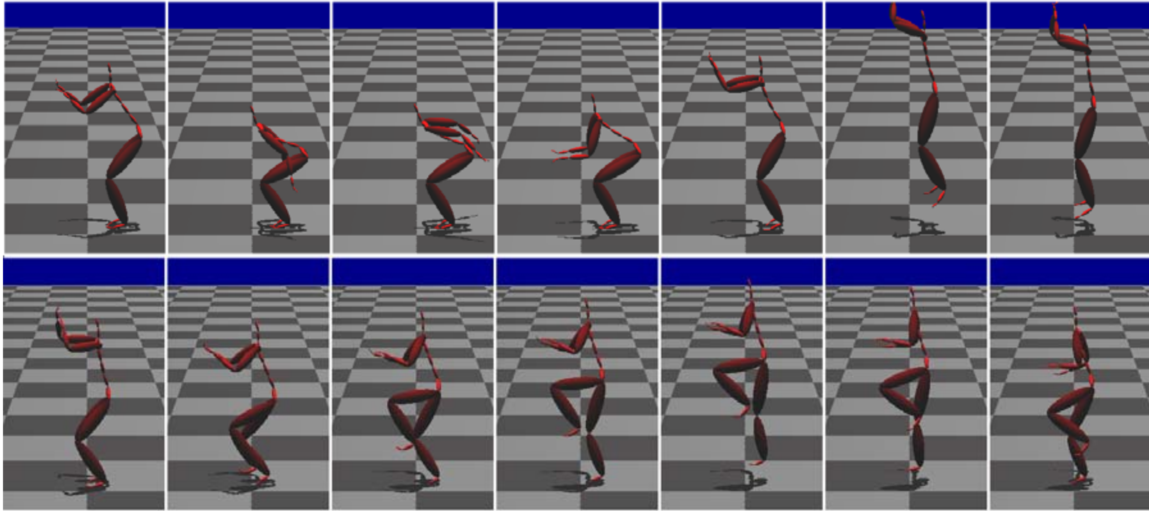

Figure 8: *Jumping to skipping*

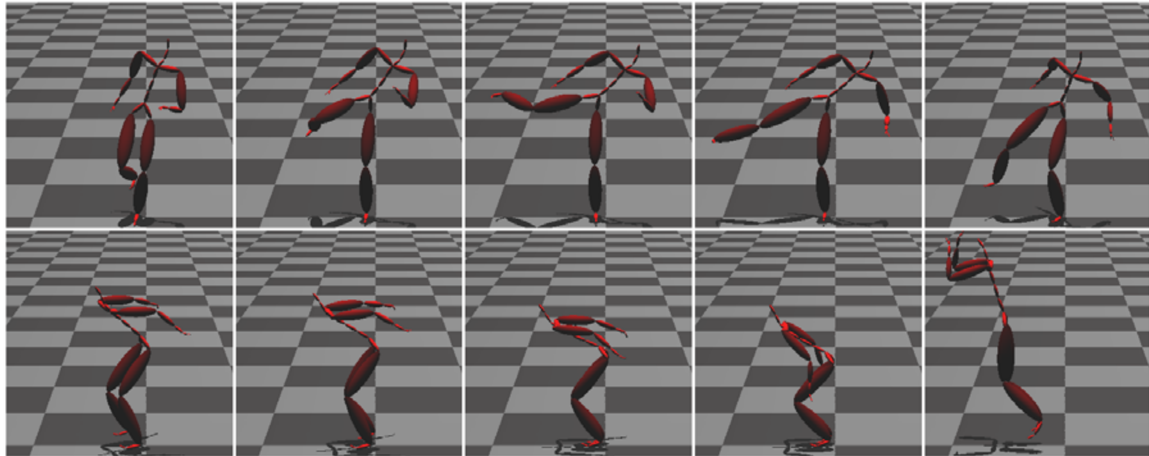

Figure 9: *Kicking to jumping*

3.6 Discussion and Further Work

The goal of this research was to provide a method for creating a motion transition in real time that is both believable and consistent with the motions being transitioned between. Previous methods for creating transitions are either too compute intensive for

real time, or lack the nuances that make the motion appealing. Our method attempts to meet these goals by modifying motion from a pre-existing source, using computationally simple transformations.

We will now attempt to evaluate the strengths and weaknesses of this technique based on the criteria established at the beginning of the paper. Namely:

1. Computing the transition should be efficient enough to run in real time.
2. Transitioning should be responsive.
3. The transitioning method should not require excessive space resources (disk space, memory, etc.).
4. The motion it creates should be continuous and believable.

First, the transition is efficient enough to run in real time. The timewarps are computed as a pre-processing step, and the transitions are created in real time.

Second, transitioning is responsive. The transitions happen instantaneously when the user presses a button.

Third, this algorithm doesn't require excessive space resources. The lookup table for the transition points of the test set of seven motions took 40 kB in ASCII text format. The space required to store this table is $O(mn^2)$, where m is the number of frames in each motion, and n is the number of motions. Since there were 7 motions, each motion had approximately 200 frames, and each entry in the table took approximately 4 bytes, the expected table size is 200*7*7*4 = 38.2 kB. For a motion set containing 50 motions, this table would take 200*50*50*4 = 1.9MB. This space could easily be further reduced

by using a binary representation, since each entry would take only one byte. For example, the set containing 50 motions would reduce in size to 200*50*50*1 = 479 kB.

Fourth, the motion created is arguably continuous and believable. For transitions between similar motions, the effect is at least as good, and for transitions between dissimilar motions, the method produces motion superior to linear transitioning, though it is not always perfect.

There are some limitations to the algorithm. This method requires transition lengths to be a power of two, because of the use of Laplacian Pyramid decompositions. Laplacian pyramids work well for dimensions that are a power of two, but not as well for other values. Though not straightforward, it is conceivable that this method could be extended to create transitions whose length is not a power of two, but in practice, this restriction isn't a problem.

Since this method deals only with forward kinematics, it is inherently susceptible to foot-skate and other artifacts. An inverse kinematic solution should fit well within this framework, and the addition of IK would alleviate foot-skate and other problems.

Acknowledgements: This work was made possible through a donation from Electronic Arts. The data used in this project was obtained from mocap.cs.cmu.edu. The database was created with funding from NSF EIA-0196217.

Chapter 4

Summary and Conclusions

This thesis has presented a method of creating motion transitions that are both realistic and computable in real time. Previous methods either were too compute-intensive to run in real time, or sacrificed motion quality to be feasible for real time.

Graph-based approaches show potential for creating motion transitions, but most of these methods are not fast enough for real-time applications. One notable exception is snap-together motion. Unfortunately, this method lacks the responsiveness required for all but a few applications.

Linear motion transitioning is the method most commonly used currently in real-time applications. While this method works, it doesn't always capture the feel of the motion. Instead, it sacrifices motion quality for ease of computation.

The method presented in this thesis accomplishes both goals of motion quality and ease of computation. By using a pre-existing motion segment to construct the motion transition, the quality of the motion is preserved. At the same time, no extraordinary computation is required, making this method feasible for real-time. The transitioning mechanism has low latency and is therefore quite responsive. Additionally, the method requires only a modest amount of space resources.

There is still much research that can be done in this area. It is a relatively new area of research, as the focus for graphics-related research for real-time applications has previously been on rendering technology. As rendering methods have matured, there has

been more interest in realistic motion techniques, and transitioning is one of the greatest needs for real-time applications.

With respect to this method, some work needs to be done on the decomposition and subsequent reconstruction of the signal. Only one method has been addressed in this research, namely Laplacian Pyramids, and another method may be more suitable. There are a number of different transformations that take a signal from the time domain to the frequency domain, and it would be beneficial to see the effect of each on this method.

The Laplacian Pyramid decomposition could also be further researched. Specifically, the current method requires the transition length to be a power of 2, but it would be nice to be able to create a transition of arbitrary length.

This method is inherently susceptible to artifacts such as foot-skate because of the forward kinematic framework used. An inverse kinematic solution should fit well in this framework, and would clean up many foot-skate problems.

Bibliography

ARIKAN, O., AND FORSYTHE, D. A. 2002. Interactive motion generation from examples. *ACM Transaction on Graphics 21*, 3, 483–490.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. 2003. Motion synthesis from annotations. *ACM Transactions on Graphics 22*, 3, 402–408.

BRAND, M., AND HERTZMANN, A. 2000. Style machines. In *Proceedings of ACM SIGGRAPH 2000*, 183–192.

BRUDERLIN, A., AND WILLIAMS, L. 1995. Motion signal processing. In *Proceedings of ACM SIGGRAPH 1995*, 97–104.

BURT, P., AND ADELSON, E. 1983. The Laplacian pyramid as a compact image code. *IEEE Transactions on* Communications, 31(4):532-540.

FANG, A. C., AND POLLARD, N. S. 2003. Efficient synthesis of physically valid human motion. In *Proceedings of ACM SIGGRAPH 2003*, 417-426.

GLEICHER, M. 1997. Motion editing with spacetime constraints. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, 139–148.

GLEICHER, M., SHIN, H., KOVAR, L., AND JEPSEN, A. 2002. Snap-together motion: Assembling Run-Time Animation. In S*ymposium on Interactive 3D Graphics 2003.*

HODGINS, JESSICA K., WOOTEN, WAYNE L., BROGAN, DAVID C., AND O'BRIEN, JAMES F. 1995. Animating human athletics. In *Proceedings of ACM SIGGRAPH 1995*, 71–78.

KOVAR, L., AND GLEICHER, M. 2003. Flexible automatic motion blending with registration curves. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003, 214–224*

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

LI, Y., WANG, T., AND SHUM, H. 2002. Motion texture: A two-level statistical model for character motion synthesis*. ACM Transactions on Graphics 21, 3, 465–472*.

LIU, C., AND POPOVIĆ, Z. 2002. Synthesis of complex dynamic character motion from simple animations. In *Proceedings of ACM SIGGRAPH 2002,* 408–416.

MENACHE, A. 2000. *Understanding Motion Capture for Computer Animation and Video Games*. Academic Press, San Diego, CA.

MIZUGUCHI, M., BUCHANAN, J., AND CALVERT, T. 2001. Data driven motion transitions for interactive games. In *Eurographics 2001 Short Presentations.*

MOLINA-TANCO, L., AND HILTON, A. 2000. Realistic synthesis of novel human movements from a database of motion capture examples. In *Proceedings of the Workshop on Human Motion,* 137–142.

PARK, S., SHIN, H., AND SHIN, S. 2002. On-line locomotion generation based on motion blending. *In ACM Symposium on Computer Animation 2002.*

PARK, S., SHIN, H., KIM, T., AND SHIN, S. 2002. On-line motion blending for real-time locomotion generation. *Computer Animation and Virtual Worlds,* 15(3), 125–138.

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

PULLEN, K., AND BREGLER, C. 2002. Motion capture assisted animation: Texturing and synthesis. In *Proceedings of ACM SIGGRAPH 2002,* 501–508.

ROSE, C., COHEN, M., AND BODENHEIMER, B. 1998. Verbs and adverbs: multidimensional motion interpolation. *IEEE Computer Graphics and Application 18*, 5, 32–40.

ROSE, C., GUENTER, B., BODENHEIMER, B., AND COHEN, M. F. 1996. Efficient generation of motion transitions using spacetime constraints. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 147–154.

WANG, J. AND BODENHEIMER, B. 2003. An evaluation of a cost metric for selecting transitions between motion segments. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003.*

WITKIN, A., AND POPOVIĆ, Z. 1995. Motion Warping. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, 105–108.