Constrained Synthesis of Textural Motion for Animation

A thesis submitted in fulfillment
of the requirements for the degree of
Master of Science

by

Shmuel Moradoff

under the supervision of
Dr. Dani Lischinski

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

December 13, 2002

Abstract

Obtaining high quality, realistic motions of articulated characters is both time-consuming and expensive, necessitating the development of effective and easy-to-use tools for motion editing and reuse. We propose a new simple technique for generating constrained variations of different lengths from an existing captured or otherwise animated motion. Our technique is applicable to **textural** motions, such as walking or dancing, where the motion sequence can be decomposed into shorter motion segments without an obvious temporal ordering among them. Inspired by previous work on texture synthesis and video textures, our method essentially produces a re-ordering of these shorter segments. Discontinuities are eliminated by carefully choosing the transition points and applying local adaptive smoothing in their vicinity, if necessary. The user is able to control the synthesis process by specifying a small number of simple constraints.

Acknowledgements

I would like to thank Dani Lischiski for his help and support. Without him, this work would never have been possible.

I would also like to thank all my friends at the computer graphics lab, especially Raanan Fatal and Amichai Nitsan for their ideas and help, Rony Goldenthal for his support, ideas and artistic advices and Ohad Barzilay for his help with finding mo-cap data on the web.

Special thanks to my parents, my brother and sister for their moral support.

Contents

1 Introduction **5**

2 Previous Work **7**

2.1 Signal processing techniques 7

2.2 Constraint-based motion editing 8

2.3 Style machines and video textures 8

2.4 Texture synthesis techniques for motion synthesis 9

2.5 Motion graphs 10

3 Textural Motion Synthesis **12**

3.1 Overview 12

3.2 Multi-resolution tree 14

3.3 Forward kinematics 15

3.4 Neighborhoods 16

3.5 Metrics 17

3.6 The synthesis process 18

3.7 Gap filling 19

3.7.1 Finding a meeting point 19

3.7.2 Finding a path in a graph 22
3.8 Local smoothing 23
3.9 Choosing the constraints 24
3.10 Root trajectory reconstruction 25

4 Results **26**

5 Conclusions and future work **34**

List of Figures

3.1 Input and output trees. 13

3.2 Construction of Gaussian multi-resolution tree 15

3.3 The three-level neighborhood used in our implementation. 17

3.4 Finding a meeting point. 19

3.5 Acceleration of the synthesis process. 21

4.1 Drunk walk. 27

4.2 Drunk walk plot. 28

4.3 High-wire 29

4.4 Cool walk 30

4.5 Ballet walk data. 31

4.6 Ballet walk synthesized. 32

Chapter 1

Introduction

High quality motion control of articulated figures is one of the most challenging tasks in computer animation. Such motion may be specified manually by skilled animators with the aid of sophisticated software tools, generated using simulation, or captured using optical or magnetic tracking. All of these creation processes can be tedious, time-consuming, and expensive. Therefore, there is a real need in a variety of easy-to-use and effective tools for motion editing and adaptation in order to facilitate motion reuse.

In this work we describe a new tool for generating constrained variations of different lengths from an existing captured or otherwise animated *textural motion*. By this term we refer to motion which can be regarded as a stationary signal at some scale. Informally, textural motion is decomposable into segments whose duration is typically small with respect to that of the entire motion, such that by looking at any given segment it is impossible to say which part of the motion it came from[1].

Many human motions are either entirely textural, or have large textural parts. One example is walking, since it can be decomposed into single step cycles, which can typically be re-ordered without making a noticeable difference to an observer. Dancing is another, more interesting example. A dance sequence can often be decomposed into short choreographic elements which may be re-ordered to yield a different sequence that would still look like a natural dance sequence to an observer. A pole vaulting sequence, however, is non-textural since it does not consist of elements that could be re-ordered.

[1] Our definition of textural motion resembles that of a quasi-periodic signal, but it is slightly more general, since we do not require the motion elements to be similar to each other.

Our tool has many possible applications. Using an existing library or relatively short motion sequences, an animator can use our technique to generate a much larger variety of motions, subject to animator-specified constraints. An existing motion can be made longer, or transformed into a loop by placing an identical constraint at the beginning and at the end of the synthesized motion sequence. A group of characters may be easily animated by assigning each character a variation of the same single captured motion. A motion may be fine-tuned to a new script or adapted to a new soundtrack by appropriate placement of a small number of constraints. In all of these tasks we do not attempt to modify the motion's style or alter any other high-level characteristics; quite the contrary, we attempt to remain as close as possible to the input motion sequence.

Our approach is inspired by previous work on texture synthesis [3, 9, 27] and video textures [25]. Given a motion sequence and a small set of simple constraints, we essentially produce a re-ordering of short segments present in the input sequence that satisfies the specified constraints. Such a reordering inevitably introduces discontinuities into the synthesized motion. To eliminate these discontinuities we find transition points at which the magnitudes of the discontinuities are minimized and apply an adaptive smoothing scheme in their vicinity.

Chapter 2

Previous Work

2.1 Signal processing techniques

In the past few years the problem of editing and reusing existing motion has attracted considerable attention. Several researchers explored ways of modifying motion by means of signal processing techniques. This approach may be used to generate entire families of realistic motions from a single input sequence with a small amount of input from the animator.

Bruderlin and Williams [6] build a Laplacian pyramid of all degrees of freedom (DOF's). In this representation each frequency band can be manipulated separately to change the motion in an interesting way. They have also experimented with blending and displacing a motion sequences.

Witkin and Popović [29] introduced "motion warping", a technique which takes motion-captured (mo-cap) data and a few key-frame constraints to create a new motion that passes through a motion warp of these key-frames and blending of given motion clips from the mo-cap data.

Unuma *et al.* [26] used a Fourier series expansions of the data. They smoothed, interpolated, extrapolated, and found transitions between motions. They also extracted the "mood" or characteristic behavior of the character, such as tiredness, etc.

All of these methods require a great deal of manual work from the animator. The results do not always looks realistic and these methods do not support hard constraints. Our approach is complementary to these techniques: we also provide a

tool for easily generating variations from a given motion, but we operate by essentially "reshuffling" the input sequence, rather then applying signal processing operations on it.

2.2 Constraint-based motion editing

Another relevant and powerful paradigm is constraint-based motion editing. A survey and comparison of such methods, which may be used to alter existing motions so as to satisfy a set of spatial or geometric constraints, is provided by Gleicher [12]. In particular, much work has been done by Gleicher and co-workers on retargeting motion to new characters [11], motion adaptation [14], and motion path editing [13], as well as by Lee and Shin [19] on interactive motion editing. Other notable representatives of this paradigm are spacetime constraints methods [7, 10, 28], which satisfy constraints by optimizing over an entire motion.

All constraint-based motion editing methods employ inverse kinematics solvers and often involve constrained non-linear optimization. Such methods also require the animator to specify a large set of constraints, such as constraints for the foot to touch the ground at each time slot. Our approach employs neither of the above, operating in a fundamentally different manner. It is not intended to be as general or as powerful as these methods; rather, it is a simple-to-use and fast alternative for editing textural motions. Again, we see it as a complementary tool, rather than as a replacement for any of these methods.

2.3 Style machines and video textures

Brand and Hertzmann [5] introduced *style machines*, probabilistic finite-state machines augmented by a multi-dimensional style variable. A style machine is constructed using unsupervised machine learning from a training set of several motion capture sequences. The learning process automatically separates between the structure of a motion (its choreography) and its style. Once a style machine has been learned, it is possible to change a style of a given new motion, assuming it consists from the same set of structural elements. It is also possible to generate an entirely new motion from a style machine by specifying the choreography as a reordering of the machine's state sequences and choosing values for the style variables. The reordering can be performed by an animator, or generated using a random walk. A similar, although somewhat less sophisticated approach was

independently developed by Bowden [4].

Style machines are an excellent high-level tool for motion editing and reuse. However, they do not provide an animator with low-level, fine-scale control, such as the ability to specify hard constraints: "I want the character to reach the highest point of its jump exactly at time t". In practice, both high-level and fine-scale controls are necessary for effective motion editing. For example, an animator could take a dance sequence and change its style using a style machine, and then fine-tune the timing of the choreography by feeding the restyled motion to our method, along with a few constraints.

Our approach resembles the one described by Schödl *et al.* [25] for generation of *video textures*. In particular, they discuss *video-based animation*, where a user is provided with high-level controls for guiding video texture synthesis. However, to our knowledge, their approach has not been applied to 3D articulated figure motion synthesis, and it does not directly support hard constraints, such as the ones used in our approach. We also borrow many ideas from recent work on texture synthesis [3, 9, 27].

2.4 Texture synthesis techniques for motion synthesis

Several other researchers applied texture synthesis techniques to the problem of motion synthesis. Probably the first step in that direction was taken by Perlin and Goldberg in their *Improv* system [22], which uses procedural noise to add realistic looking variability to motions.

Pullen and Bregler [23] describe a motion synthesis method inspired by De Bonet's texture synthesis algorithm [8]. Their approach decomposes the training data into frequency bands and synthesizes a new sequence, one frequency band at a time. The approach was applied to generate a realistic repetitive motion of a 2D character with three degrees of freedom (a hopping wallaby). Our approach is different since it is closer to more recent texture synthesis techniques [9, 27], which generally outperform De Bonet's algorithm in terms of the quality of the synthesized textures. In this paper we demonstrate our approach using a variety of complex motions performed by a 3D articulated character with 23 joints and end effectors.

Pullen and Bregler [24] extended their work by adding an input of a partial key-framed data, for example: an animation of the legs of some character. Then they match 1-2 DOF's low pass band of the mo-cap to the key-framed band, split the data and find a sequence in the mo-cap band that best matches the key-framed

band. They use this sequence to add high-pass bands and/or other DOF's to the key-framed data. With our technique, we do not need the extra key-framed data to create a new full animation sequence.

Molina Tanco and Hilton [21] construct a two-level statistical model from motion capture training data. The motion capture data is clustered, and the clusters define the states of a Markov chain which encodes the high-level temporal behavior of the character. The second level relates the states of the Markov chain with segments of the original motion. Dynamic programming is used to generate the state sequence of a synthesized motion. The actual motion is then generated by copying segments from the second level and blending in the areas where these segments join. Li Wang and Shum [20] also described a two level statistical model. They divide the motion into segments, such that each segment is associated to one texton. A texton is modeled by a linear dynamic system (LDS), while the texton distribution is represented by a transition matrix indicating how likely each texton is switched to another. Synthesis of motion is done by first generating a texton path (the user may edit the texton path), and then for each texton, synthesize a texton sequence that begins with two key poses (that are associated to each texton) and constraint to end with two frames that are the key poses of the next texton in the path. Our approach is different in that it does not construct an explicit statistical model of the input data. In order to satisfy the animator's constraints, our approach efficiently identifies good transitions directly in the original motion capture data. Thus, our approach should reduce both the number of transitions blends and the magnitudes of the discontinuities that must be blended.

2.5 Motion graphs

Some work, concurrent to ours, was recently done using the *motion graphs* framework. In this framework, a huge matrix of all transitions from one frame to another is first created. A cost is associated with each transition and the matrix is pruned by eliminating expensive transitions. Next, a graph of transitions is created. Finally, new motion sequences are generated by finding paths in the graph, which meet some constraints or minimize some cost function.

Kovar *et al.* [16] use this framework to create motion that follows some 2D path. They use a metric similar to ours to find the distance between every two frames to create a distance matrix which is then pruned by taking only local minima in the matrix and then use some threshold. Pruning is essential since the data that they are using has many similar frames. They then create a graph of all possible

transitions and prune it again to get the largest strongly connected component to avoid dead ends. Finally they find a path that minimizes some user-defined cost function. They suggest a cost function for a motion that follows a given 2D path. This approach handles new 2D path synthesis very nicely, it does not allow the user to specify hard constraints as we do, and their input data must contain many good transitions between frames.

Lee *et al.* [18] use a low-level Markov process to create a probability matrix for transitions between each two frames. Then they prune the matrix much like Kovar *et al.* [16], with the addition of pruning frames based on contact. They also create a higher level statistical model which clusters the frames in a transitions graph, such that a motion can be thought of as a path between clusters. They describe three ways to create a motion: the user may choose the next cluster to follow, sketch a 2D path (as in [16]), or act in front of a camera set. This method requires a large database to work well, and does not handle hard constraints.

The motion graphs framework was also used by Arikan and Forsyth [2]. Their database contains a large number of short motions. They construct a transition matrix between each pair of motions. After pruning they get clusters of elliptical shape in the matrix. Each ellipse is clustered with k-mean clustering, which is actually a summarized graph. Then they split each cluster again and again to get a hierarchical representation of the graph. A motion is actually a path of clusters from different levels in the hierarchy. They allow only paths that meet user defined hard constrains, and choose a path that best fits some soft constrains. It seems that in order to meet the hard constrains and to get the ellipses, the data should have many similar frames and for every two similar frames, the preceding and succeeding frames are also similar. In contrast, our approach can work even on short but complex motions in which there is not enough data to generate a good clustering.

All of these three methods use large motion databases, and therefore require very long pre-processing time (several minutes to several hours). With our method, the computation time for the entire process is about the same as the generated animation length.

Finally, there is also some related work in the area of computer vision concerned with learning 3D human motion for tracking and gesture recognition (e.g., [30]). In these works, motion synthesis is typically a secondary goal, and therefore little attention is paid to animator control (constraints) and the quality (smoothness) of the synthesized motions.

Chapter 3

Textural Motion Synthesis

3.1 Overview

Our method takes as input a textural motion sequence, a set of synthesis constraints, and the desired length of the synthesized output sequence. The input motion sequence is given in a BVH format which consists of the character's skeleton hierarchy description followed by a sequence of *frames*. Each frame specifies the absolute pose of the articulated figure by means of six degrees of freedom for the root of the hierarchy (three rotation angles and a 3D translation) and three rotation angles for each joint. All rotations are internally represented using quaternions. The constraints are simply a set of pairs; pair (i, j) means that the i-th frame in the input sequence is constrained to become the j-th frame in synthesized output sequence. Optionally, a new path is also specified for the synthesized motion.

The output of our method is a new motion sequence of the desired length. For the most part, it consists of sub-sequences of frames from the original sequence reordered in such a manner that all of the constraints are satisfied. The exceptions are the frames in the neighborhood of the transitions between the sub-sequences; such frames might be slightly modified by our local smoothing scheme for eliminating discontinuities, as described in section 3.8.

By using sub-sequences of frames from the original motion to the synthesized sequence, we restrict ourselves to reusing frames. This mean that we can not create frames that did not exist in the original data (such as extrapolating frames). We do this in order to ensure that each frame in the synthesized motion is valid. We would otherwise have to ensure validity, which would be difficult to do with-

out a higher level understanding of the articulated figure, such as that used by constraint-based motion editing methods (described in section 2.2).

In our current implementation, the output length must be an integer multiple of the input length, but this is not a real limitation, since in order to generate an arbitrary length sequence we can generate one of length rounded up to the nearest integer multiple, and then trim a few frames from the ends.

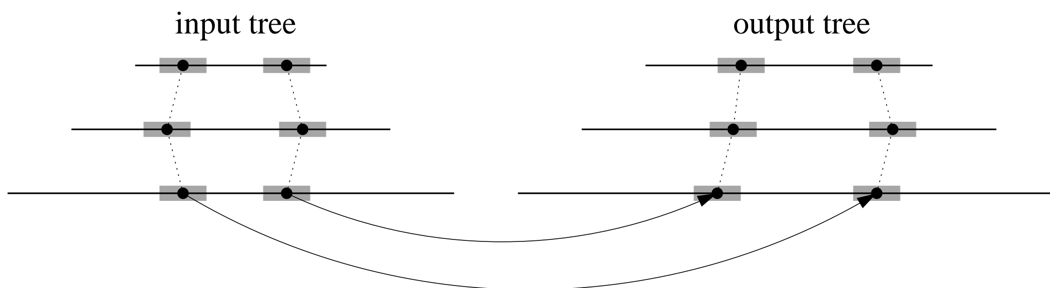

Figure 3.1: Input and output trees.
The horizontal lines represent levels in the trees. The bottom level in each tree contains the frames at the original temporal resolution. The next level up contains frames after low-pass filtering and subsampling, and so forth. Arrows indicate constraints: in the initialization stage each constrained frame and each of its ancestors in the input tree (shown as black circles) is copied to its designated location in the output tree, along with a small neighborhood of siblings (shown in grey). It is now up to the synthesis procedure to fill in the remaining gaps, proceeding from the top level down.

The synthesis procedure employed by our method is similar to the multiresolution constrained texture synthesis procedure described by Wei and Levoy [27]. The main steps of our method are:

1. Construct a Gaussian multi-resolution tree (*input tree*) above the sequence of the root translation and the joints rotations by iteratively low-pass filtering and subsampling. See section 3.2 for details.

2. Apply forward kinematics to compute a vector of 3D joint and end-effector locations, as described in section 3.3. This vector gives the absolute 3D pose of the articulated figure at that frame.

3. Create an empty *synthesis tree* with the same number of levels. The finest resolution level of this tree will eventually contain the output sequence.

4. Copy the constrained frames (and their ancestors) to their target locations in the synthesis tree. Each constrained frame (ancestor) is copied along with a small surrounding neighborhood (Figure 3.1).

5. Starting from the coarsest level, fill in all the gaps left between the constraints by searching for best-matching neighborhoods at the corresponding level of the input tree, as described in sections 3.4, 3.5 and 3.7.

6. Apply adaptive local smoothing in the vicinity of transitions between original motion sub-sequences to eliminate discontinuities, if necessary (section 3.8).

7. Repeat for the next level until the finest level has been filled and smoothed.

8. Construct a new root rotation and translation trajectory, as described in section 3.10.

In the remainder of this chapter we describe in more detail the steps of our method. We begin by describing the multi-resolution tree and the forward kinematics method that we use (steps 1 and 2). Then we define the neighborhood and the distance metric used when searching for matches in the input tree and follow with a detailed description of the synthesis process and gap filling algorithm (steps 5 and 7). We then describe the smoothing algorithm that we use (step 6). We describe how to choose the set of constraints, and we finish this chapter with a description of our method for the reconstruction of the root trajectory (step 8).

3.2 Multi-resolution tree

Constructing a multi-resolution tree is required for the acceleration of the gap filling algorithm (described in sections 3.7.1 and 3.7.2)

We have tried several different filters for the construction of the tree: a Gaussian filter, a Laplacian filter and wavelets. Finally we decided to use the Gaussian filter since it gave us the best results. It is possible that the Gaussian tree is the best choice here because each level of the tree actually corresponds to a valid low-passed version of the original motion: each coarser level frame is obtained by smoothing several frames at the next finer level by applying a Gaussian filter (see figure 3.2). In contrast, the Laplacian and wavelet trees do not contain a valid motion at each tree level.

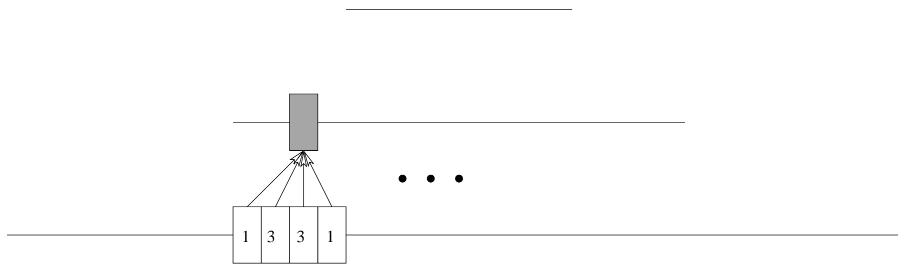

Figure 3.2: Construction of Gaussian multi-resolution tree
This image shows the creation of one frame (the gray frame in the image) at one of the middle levels by applying a Gussian filter on the frames at the finer level. The numbers indicates the filter's weights for a filter of size 4, which we typically used.

The size of the filter should depend on the nature of the motion. A motion with sharp or fast movements will need a shorter filter, because a long filter might use some very different frames to create a certain frame at a courser level, which means that the "essence" of the frame might get lost. For our data we typically used a filter of size four.

The number of levels is also important: more levels lead to faster synthesis. But if we use too many levels, the motion at the coarsest level might be just a series of frames with no connection between them, since we subsample each level. In a walking motion, for example, we would need that each walking step will correspond to at least a few frames at the coarsest level; intuitively, the coarsest level motion should still be recognizable as walking. We experimented with several numbers of levels, starting with one level and adding one more level each time. For the results shown in this work, we ended up using between three and five levels (four levels were used in most cases).

3.3 Forward kinematics

As mentioned earlier, the input motion is given as a description of the skeleton's hierarchy and a sequence of frames. Each frame is represented by its *joint angle representation*, which means that for each frame we hold the derivative of the translation for the root of the skeleton, and a 3D rotation for each joint. This representation has the advantage that whenever we want to change some values (translation of the root or rotation of a joint) it does not change the structure of the

skeleton. So, when we apply a filter on a frame (as we do while constructing the multi-resolution tree in section 3.2 and while smoothing frames — section 3.8), we do this on the joint angle representation.

However, the joint angle representation has some major drawbacks when it comes to comparing between frames: changing the rotation of a joint has an effect on the character pose that depends on the joint length. Furthermore, a rotation of one joint affects all joints that are below it in the skeleton hierarchy; as a result, some joints (such as the hip joints) might have a huge effect on the skeleton, while other joints (such as a hand joint) might have insignificant effect on the overall character pose. Also the same rotation of a joint might have a different effect depending on the position of the joints that are below it in the hierarchy. Therefore defining a distance metric for the joint angle configuration is a very difficult task (the weight of each joint will have to change depending on the orientations of other joints).

Therefore, we construct another representation for each frame: the *3D pose representation*, which is the set of 3D positions of all joints and end-effectors. With this representation, as expected, changing the values will alter the shape of the skeleton. As such, we can not use it while applying a filter on a frame, but when comparing between frames, this representation is exactly what we need: the distances between the positions of joints and end-effectors in two frames can describe the distance between those frames (see more on the metric we used in section 3.5).

Calculating the 3D pose representation of a frame out of its joint angle representation is done with *forward kinematics* methods. We do it by first applying the root translation on all the joints and end-effectors. Then each joint and end-effector is being rotated by the rotations of all its ancestors in the hierarchy.

3.4 Neighborhoods

Gaps in the synthesis tree are regarded as sequences of empty "frame slots". In order to fill in an empty slot we examine its neighborhood in the synthesis tree and look for similar neighborhoods around slots at the same level of the input tree. As explained in section 3.3, the frames we refer to are in 3D pose representation. The neighborhood we use is shown in Figure 3.3. It is similar, in principle, to the neighborhoods used by Wei and Levoy [27], though there are some important differences.

The neighborhood is constructed over three levels with the frame that is being synthesized in the middle level (see Figure 3.3). The upper (coarser) level gets

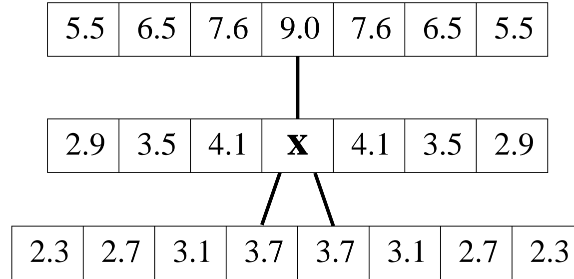

Figure 3.3: The three-level neighborhood used in our implementation. The empty cell whose value is to be determined is marked by an $\boldsymbol{x}$. The neighborhood consists of $\boldsymbol{x}$, its ancestor and its children in the tree, and three more slots to the left and to the right on each level. The numbers in the cells are the relative weights used by our distance metric.

a higher weight because we want to keep the general path that was found in the coarser level (see section 3.7) the other two finer levels get the same weight.

We give different weights to different slots in the neighborhood (decreasing with distance from the center slot). The relative weights of the slots are shown in Figure 3.3.

Slots from the next finer level are included — although this level has not been processed yet, some of its slots may have been filled by constrained frames during the initialization of the output tree. The synthesis algorithm must take these frames into account when searching for the best frame for the current slot.

The difference between two neighborhoods is defined simply as a weighted sum of the distances between corresponding pairs of slots. Pairs in which the output neighborhood slot is still empty are assigned a weight of zero, and all of the remaining non-zero weights are renormalized to sum to one.

3.5 Metrics

As mentioned in the previous (3.4) section, when we compare between two neighborhoods, we actually have to compare between frames. We use the 3D pose representation to compare between frames, as explained earlier (section 3.3).

The definition of distance between corresponding slots is not as trivial as one

might think. Consider two slots containing frames numbered i and j. Simply computing the distance (in some vector metric) between the corresponding position vectors v_i and v_j will not do: such a metric would fail to recognize the similarity between identical poses differing only by translation. Also, it does not take into account the velocities of the joints. This could lead to unnatural direction reversals. Therefore, we compute the distance between the velocity (temporal derivative) vectors $\dot{v}_i$ and $\dot{v}_j$ instead, thus ensuring translational invariance in the synthesis process. Although in principle it is possible for two totally different character poses to have similar velocity vectors, in practice such a situation is extremely unlikely, and we have never encountered it in our experiments.

After experimenting with various standard vector metrics we concluded that a sum of $\|\cdot\|_\infty$ and $\|\cdot\|_{0.5}$ yields the best results. In other words, we take into account both the maximum difference among the velocity components and the sum of the square roots of the differences of all components. This metric was found to give slightly better results than the more familiar $\|\cdot\|_2$ norm.

There is no substantial mathematical reason for using this metric. We base our decision for using this metric on empirical experiments. Having said that, there is some intuition that we used while searching for the best metric: The $\|\cdot\|_{0.5}$ part is actually depressing all pairs of frames that are very similar but has a very small difference in many joints simultaneously. While the $\|\cdot\|_\infty$ part is depressing all pairs that have at least one joint with a big difference.

3.6 The synthesis process

A level in the synthesis tree is a sequence of slots (frames), some of which are empty and some are full. We synthesize a frame only if some of the frames in its neighborhood are full. The synthesis process is simply finding a frame in the input tree that will be copied to the empty slot. The candidates for the frame to be copied are all or some of the frames in the same level as the empty slot: In the coarsest level the candidates are all the frames in that level, but in the finer levels the candidates might be a subset of them (see the notes on acceleration in section 3.7.1).

We compare the neighborhood of the empty slot with each of the candidates using the metric shown in section 3.5 and choose the frame with the neighborhood that is the best match (or is nearly the best match — see the notes on randomization in section 3.7.1) to the neighborhood of the empty slot.

When copying a frame we copy both the 3D pose representation and the joint angle representation. As mentioned in section 3.3, the joint angle representation contains the joints rotation and the derivative of translation for the root of the skeleton. At the end of the synthesis we need to compute the 3D translation of the root out of its derivative for all frames: For the first frame in the synthesized sequence, we copy the 3D translation of the frame that was chosen as the first frame. For all other frames we integrate to get the 3D translation.

3.7 Gap filling

We now describe in more detail how to fill in the gaps at a particular level of the tree. Each gap is a sequence of empty slots bounded by the constrained frames, either from one side or from both sides. We will focus on the latter case, which is the more interesting (and complicated) among the two.

3.7.1 Finding a meeting point

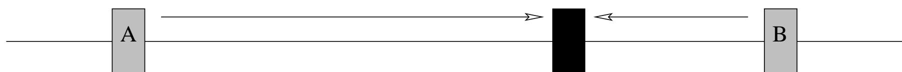

Figure 3.4: Finding a meeting point.
Frames 'A' and 'B' are the constrained frames. The black frame is a candidate for a meeting point. We synthesize the frames alternating sides towards the candidate.

One possible approach towards filling such a gap is to start filling the empty slots working our way from both ends towards the middle. However, choosing the middle of a gap as the meeting point is an arbitrary decision, and it will not necessarily result in the most continuous transition. Therefore, our strategy is to first find the best meeting point inside the gap, and then work our way towards it from both ends (see figure 3.4).

We start at the coarsest level of the tree. For each gap, we search for a meeting point by employing an exhaustive search. We consider all the interior slots of a gap as potential meeting points and perform the synthesis towards each such point from both ends. We synthesize one frame from the left side and then one frame from the right side and so on. We do this due to the fact that when the synthesis

of both sides is getting too close to one another, the neighborhoods might overlap and influence each other. For each of the potential meeting points we record a *sequence cost*. The sequence cost is defined as the sum of distances between the neighborhood of each synthesized slot and its best matching neighborhood in the input tree. The meeting point with the lowest sequence cost is then chosen.

When we finish filling all the gaps in a level, we perform a local smoothing (see section 3.8) at each point of the sequence where two frames are not consecutive. We then continue to the next (finer) level. Again, we search for a meeting point and fill in the gaps. This time our neighborhoods contain much more information since the coarser level is full. When we finish the synthesis of the finest level, we have a smooth synthesized motion that satisfies all of the input constraints.

Acceleration

The exhaustive search is expensive, since for a gap of k slots we consider k different ways of filling it, each time searching for the best matching neighborhood for each slot. Thus, if we have n different neighborhoods at that level we end up evaluating the neighborhood distance metric k^2n times. However, at the coarsest level of the tree, both k and n are typically small. In the following (higher resolution) levels we do not search all possible meeting points. Instead we use the meeting point that we found in the previous level, as an initial guess, and examine only a small neighborhood of slots around this initial guess to find the new meeting point (instead of looking at all the slots in the gap).

We tried an additional optimization to speed up the synthesis of a frame. We used clustering to divide the frames at each level into several groups. When synthesizing a frame we compare the neighborhood of the empty slot to all the frames in the group that is closest to the neighborhood of the synthesized frame. We used K-means as a clustering algorithm (see [15] for a review on clustering algorithms) with a modification: each frame is being assigned to *two* clusters. This is because a cluster is typically containing short, very similar sequences, so all frames at the beginning / end of these sequences don't have consecutive frames to their left / right respectively. As we tried this acceleration method on our data, we found that creating more than five to ten clusters degrades the quality of the animation. With consideration of the overhead caused by creating the clusters, this optimization accelerated the algorithm by a factor of 2 to 3.

This optimization was satisfactory but not good enough. Our algorithm was now running for approximately seven to ten minutes to create a 40 seconds synthesized

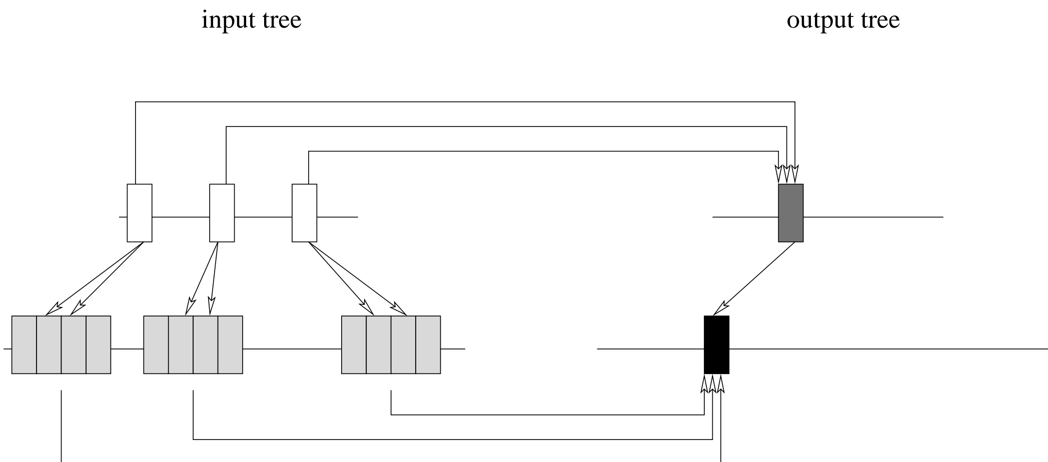

Figure 3.5: Acceleration of the synthesis process.
The black frame is the current slot to be synthesized. The dark-gray frame is the black frame's father. The white frames are the best three candidates for the dark-gray frame, and the light-gray frames are the white frames sons and their neighbors. The light-gray frames are the only candidates for the black frame.

animation. So we tried a different optimization instead of the clustering method. When filling an empty slot, instead of searching for a match among all the neighborhoods at the corresponding level of the input tree, we only examine a constant number of neighborhoods: first we look at the parent of the slot that we are going to fill. This is the chosen frame at the previous level. Actually, we are looking at a few (usually three) frames that had the best match for that slot, so we have the three best candidates for the parent. For each of those parent frames we take their children on the next level and a small number (usually four) of frames around them. So we have twelve frames, which are the candidates for the current slot (see Figure 3.5) . This optimization accelerates the synthesis by at least one order of magnitude, which means that for the synthesis of 40 seconds of animation, the total running time is about 40 seconds (see Table 4.1). The same idea was used to speed up texture synthesis by Bar-Joseph *et al.* [3].

Adding randomization

So far, the approach we have described is entirely deterministic. In order to produce some variations between different runs of the algorithm (even with the same set of constraints), we are using a slightly randomized version of the approach:

After determining the meeting point for a gap, we synthesize the sequence again, but this time instead of filling slots based on the best-matching neighborhood, we consider a small set of neighborhoods (typically 2-3) that had the best matches, and choose one of them at random. This randomization takes place at all levels of the synthesis tree except the last, finest level.

3.7.2 Finding a path in a graph

In the spirit of [2, 16, 18] we experimented graph path finding between constraints as an alternative to the gap filling method, described in section 3.7.1. The graph path finding method enables us to create sequences with more than one meeting point inside a gap.

For the coarsest level, we create a matrix M where $M_{i,j}$ is the distance between frame i and frame j. $M_{i,j}$ is calculated using the same metric as before, except that in this case both frames are taken from the original data. Now we build a graph where each frame is a node, and for each $M_{i,j}$ smaller then some threshold there is an arc in the graph from node i to node j, with the weight $M_{i,j}$.

For each gap in the coarsest level, we search for a path that will start with the frame that is just before the gap and end with the frame just after the gap. We use DFS search to find the path with the lowest cost (sum of weights). We accelerate the DFS search by stopping the search at branches when some maximum cost has reached, or if we already found a path with a lower cost then the current path cost. The path we found is the sequence of frames for the gap. After copying the frames we have a synthesized animation in the coarsest level.

Now we fill the finer levels. We want the next levels to follow the general path that we have found in the coarser level, so all we need to do is use the sons of the chosen frames in the previous level as the frames in the current levels. For each frame we have to decide whether to use left son or the right son. It seems quite simple: the animation should be continuous, so once we determine one frame we actually have the whole animation. But, at each meeting point, there can be several choices for the order of the frames. So we search the neighborhood around each meeting point of the previous level, for the best path among all possibilities of their sons, in a way that will minimize the cost.

Note that this time the cost is calculated as the distances in the synthesized data, which means that we get a more accurate result, since we consider the actual neighborhood that was synthesized and not the input data. This influences mainly

the areas around meeting point — exactly where we need it.

3.8 Local smoothing

Some input sequences do not contain a sufficient number of repeating character poses, making it difficult for our algorithm to find visually continuous transitions. Thus, discontinuities may occur at transition points between sub-sequences of the original motion. The same problem was encountered in video textures by Schödl *et al.* [25].

Since we know in which precisely at which frames of the synthesized sequence such transitions take place, we adaptively apply local smoothing in a small neighborhood around such transitions. More specifically, let i and $i+1$ be two successive frames in the synthesized sequence that were not successive frames in the original sequence. The smoothing is performed as follows:

1. Numerically approximate the acceleration (second derivative) of each joint and end-effector 3D location at frames i and $i+1$.
2. Compute the average acceleration of each joint and end-effector 3D location over frames $i-3$ through $i-1$ and $i+2$ through $i+4$.
3. If the accelerations at frames i and $i+1$ are less than or equal to the average acceleration computed in the previous step there is no need to perform any smoothing.
4. Otherwise, the trajectories of the joints are smoothed out by applying a Gaussian filter of width 5 at the transition frames i and $i+1$. The filter is applied to the quaternions representing the joint rotations, after which the corresponding 3D location vectors are recomputed by forward kinematics.
5. The above four steps are repeated as many times as necessary, until a discontinuity is no longer detected in step 3. At each iteration we increase the neighborhood over which smoothing is performed in step 4. Specifically, in the k-th iteration we convolve frames $i-k$ through $i+k+1$ with the Gaussian filter.

Usually, the smoothing algorithm did not exceed three iterations, that is: $k \leq 3$. This means that the smoothing filter was applied over six frames or less.

Whenever smoothing a data there is always a chance that the smoothing will result in a non-realistic outcome. But with our algorithm, we always smooth frames that are already very similar to one another. So there is no real danger of getting an unnatural motion. Also, all frames in all levels are valid motion poses. Therefore, smoothing between similar frames will give us valid frames. Had we used a Laplacian or a wavelet tree, we might have ended up with poses that are not possible for the character. But we are using a Gaussian tree, so there is no such problem.

As can be seen in our results, this adaptive local smoothing scheme successfully eliminates visual discontinuities in the motion without introducing conspicuous smoothing.

3.9 Choosing the constraints

In this section we will describe the limitations for choosing a set of constraints. The animator must choose a set of constraints as input for our method. A poor or random choice of constraints will usually result in a poor output.

First, at least one constraint is required. Any synthesis must have at least one frame to start the synthesis from. This frame might be anywhere in the synthesis sequence. As we mentioned, we synthesize a frame only if at least part of its neighbors already have values, so there will be something to compare when synthesizing other frames in its vicinity. Of course, one can choose a random frame if he wishes to have only one constraint.

In general, it is not a good idea to choose constraints from the very beginning/end of the input sequence. This is because when synthesizing frames before/after those constraints respectively, the algorithm will be forced to find a meeting point at the frames between the first/last frame and that constraint. Since there are very few frames to choose from, the chosen meeting point might not be a good one.

When setting constraints, the user must make sure that they will not be too close to each other in the synthesis sequence. If for example, the trees have four levels, then setting two constraints that have less then seven frames between them, will cause the constraints to be at the same frame in the coarsest level. Actually the user might want a much bigger distance between constraints at the synthesis sequence, since we copy a few of the neighbors along with the constraint frame.

As we mentioned, setting random constraints might not result with a good anima-

tion sequence. Suppose the input motion is of a walking character, where each walk cycle is of forty frames. That is, every forty frames the feet are at the same position. Now suppose that we constrain frame number one of the input motion to appear as frame number one of the synthesis sequence, and also to appear at frame number sixty of the synthesis sequence. So in the input motion, sixty frames are one and a half walk cycles but now we forced it to be one or two cycles. With the graph path finding method (section 3.7.2), that might not be so bad, because the algorithm will probably find a few meeting points in that gap and spread the error between those meeting points. Smoothing will make the motion quite good. But with the meeting point search algorithm (section 3.7.1), that will not result with a good motion since there will be only one meeting point, so it will have to smooth between frames that could be very different from each other. Of course, if the data is more diverse, that is if it contains both shorter walk cycles and longer walk cycles, this problem will be much easier even for the meeting point algorithm.

To conclude, hard constraints are very powerful tool, however they must be selected carefully.

3.10 Root trajectory reconstruction

In our current implementation the user is able to specify a new animation path (root trajectory) for the cases where the input motion advances roughly along a straight line. For each synthesized frame, we take the root trajectory derivative (velocity) from the original motion and use this velocity to compute the new root translation in the synthesized sequence, taking into account changes in the local reference frame along the new path. This procedure ensures that the character advances along the new path in a similar fashion to the original motion.

If a new animation path has *not* been specified by the user we reconstruct one by taking the root trajectory derivatives from the original motion, as before, and simply integrating them to obtain a new root trajectory.

Chapter 4

Results

Name	Orig. length	New length	Num. constraints	Synth. time
Drunk walk (A)	512	1024	7 (3)	37 (23)
Drunk walk (B)	512	1024	8 (3)	43 (21)
High-wire	512	1024	6	32
Ballet walk	512	2048	9	81
Cool walk	512	1024	8 (3)	30 (13)

Table 4.1: Statistics for the synthesis examples.
Times were measured in seconds on a 866 MHz Pentium III PC. The numbers in brackets refer to the gap filling with path finding as described in section 3.7.2.

We implemented our algorithm in C++ as a plug-in for the Maya animation system [1], and have been able to generate a variety of motions from different motion capture sequences.

Finding a good animation sequence, whether it is a mo-cap or an artist animation, is very difficult. The sequence should be long enough, in a good quality (no jerks etc.) and should fulfil our definition for *textural motion* as mentioned in the introduction section. This is why the number of data sequences that we worked with is relatively small. Acquiring good data requires a mo-cap studio and/or a good animator, neither of which were available to us.

In all the examples we used a character with 23 joints (i.e. 72 degrees of freedom: 3 translations and 69 rotations). All examples were synthesized using the meeting point search algorithm described in section 3.7.1. Some of them were also synthesized using the graph path search algorithm described in section 3.7.2. In some

examples we bound the skeleton to a mesh. In other cases we used a stick figure to create the movies, so all joints could be seen on the character.

Various statistics regarding the examples in this section are summarized in Table 4.1. All of the corresponding movies can be found at http://www.cs.huji.ac.il/labs/cglab/research/ltca.

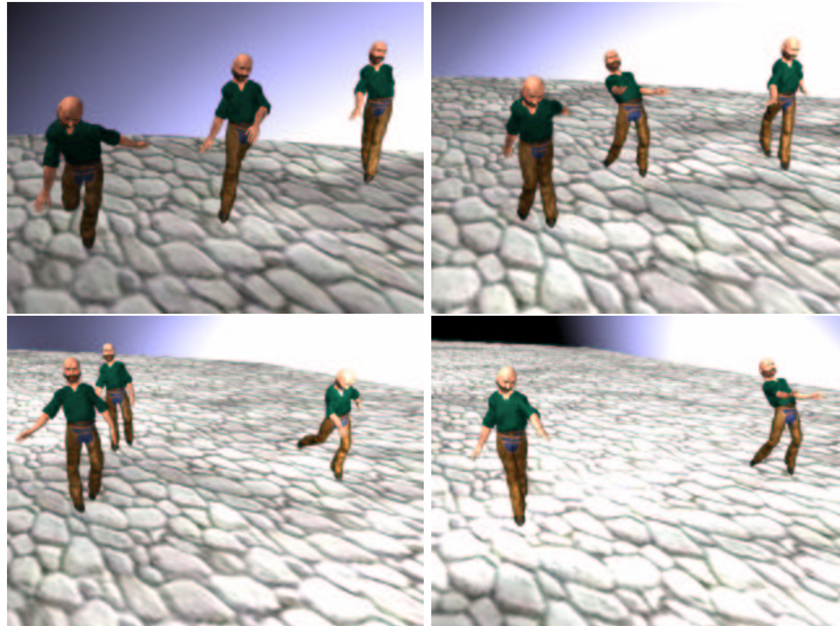

Figure 4.1: Drunk walk.
Four frames from an animation we have generated (drunk.mpg and drunk-path.mpg). The original sequence is performed by the middle character; the other two were synthesized by our method.

Figure 4.1 and the accompanying movie (drunk.mpg) demonstrate an application of our method to modify and double the length of a drunk person walking sequence. Figure 4.1 shows a few frames from the resulting animation, where we have placed the original motion in the middle and the synthesized motions on the right and left. The plots in figure 4.2 show the trajectories of two joint angles in the shorter original sequence (top plot) and in the longer synthesized ones (bottom plots). The constraints specified by the user for this case are indicated by the letters ‘A’ through ‘H’. The letters in each plot indicate the locations of the constrained frames in the corresponding sequence.

This is a rather challenging test case, since the input sequence contains only 14 step cycles, almost none of which is very similar to another because of the waving

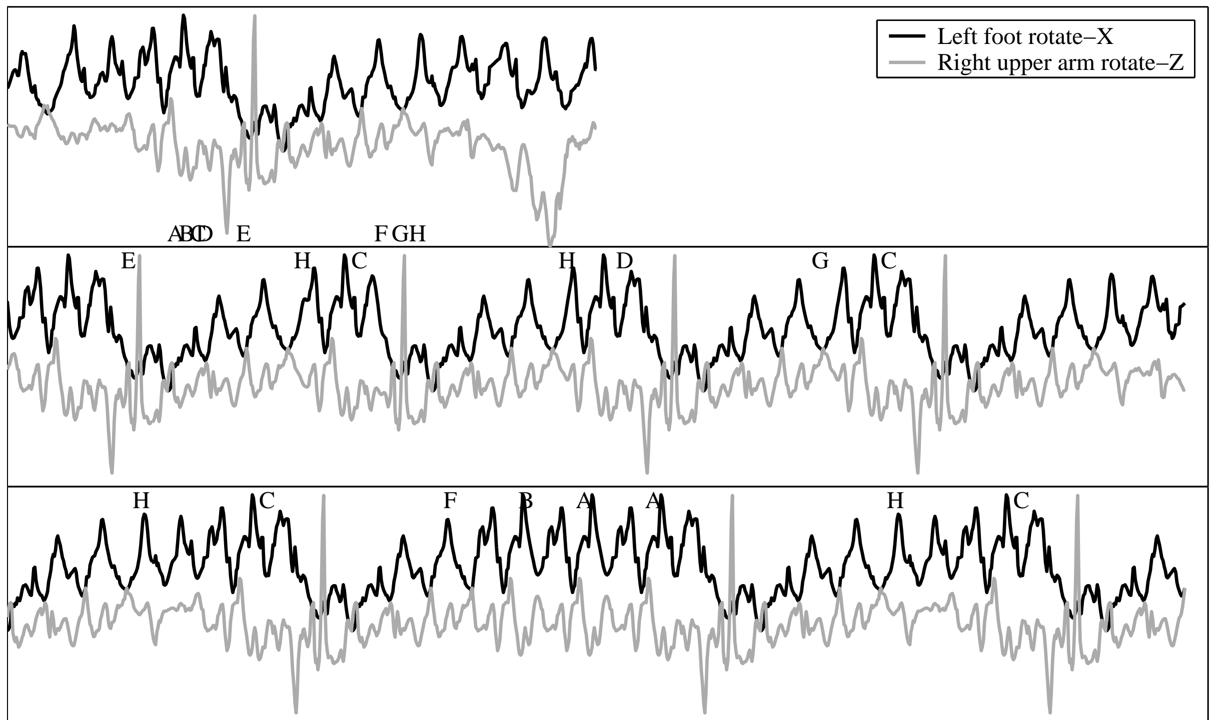

Figure 4.2: Drunk walk plot.
The three plots illustrate the results of the synthesis by means of the angular trajectories of two joint angles. The top plot corresponds to the original sequence. The letters 'A' through 'H' indicate the positions of the constrained frames in the original sequence and in each of the two synthesized ones.

arms and the stumbling nature of the walk and the pose of the character hardly ever repeats itself. Therefore, we believe it would be difficult to construct a good high-level motion model [4, 5, 21] from such a training set. Our method, however, is still able to generate visually continuous and naturally looking motions by locally smoothing over discontinuous transitions. The local and adaptive nature of the smoothing succeeds in preserving the characteristics of the original motion.

With this example we also used the graph path finding algorithm described in section 3.7.2. The resulting movie is at drunk-path.mpg. We simply removed some of the constraints to get more than one meeting point inside a gap. The synthesized animation that we got had two meeting points in one of the gaps (there were only two gaps in this example since there was only three constraints). The reason for not getting more meeting points is that a meeting point yields a higher cost for that sequence. There is a trade-off between having fewer meeting points and having *good meeting points*. By good meeting points we refer to a meeting point with a low cost, which means a smooth transition between frames. If one would like to get more meeting points, he could simply increase the cost of a sequence with fewer meeting points.

Figure 4.3: High-wire
Four frames from the animation movie (highwire.mpg). The original motion is performed by the character in the back.

Our next example uses a high-wire walking input sequence (Figure 4.3, and the movie highwire.mpg). Again, the basic cycle of this motion is quite complex and does not repeat itself too much, but our method succeeds in generating a longer sequence without noticeable discontinuities[1].

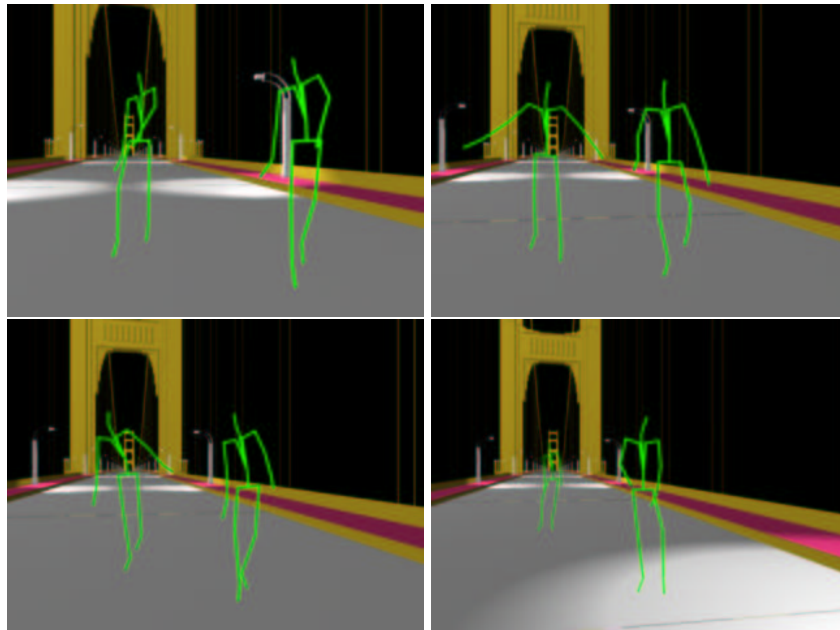

Figure 4.4: Cool walk
Four frames from the animation movie. The original motion is performed by the left character. This example was synthesized using the 'finding a meeting point' algorithm described in section 3.7.1 and the associated movie is cool.mpg. It was also synthesized using the 'Finding a path in a graph' algorithm described in section 3.7.2 and the movie is cool-path.mpg.

Another example is a "cool walk" sequence (Figure 4.3, and the movie cool.mpg). In this case the synthesized sequence was constrained to begin identically to the original sequence, and then to continuously diverge from it. Here we also experimented with the graph path finding algorithm (cool-path.mpg). Note that with this method, the synthesis time is much shorter. This is because there are not so many good transitions between frames, so when we search the graph for a path, many branches will be excluded early in the search.

[1] A jerk in the right leg of the synthesized character can be noticed around seconds 8, 18, and 27. This flaw is present in the original motion, as can be seen by observing the original motion around second 6.

Finally, we apply our method to a "ballet walk" sequence. In the original sequence the character is walking in a roughly straight line (Figure 4.5, and the movie ballet-original.mpg). We constrained the synthesized sequence to begin and end with the same character pose and specified a new 8-shaped path instead of the original straight one. The result is an animation loop of a person walking along the new path (see Figure 4.6, and the movie ballet-eight.mpg).

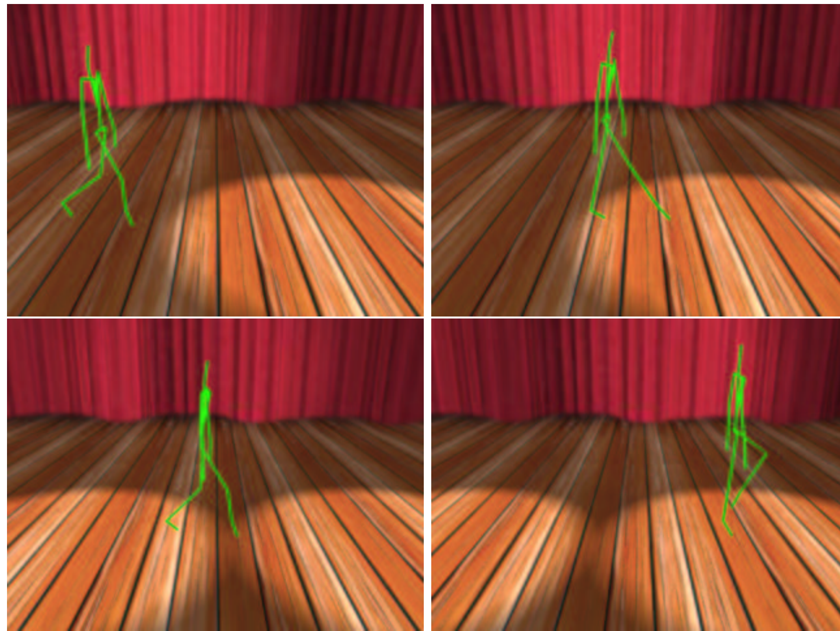

Figure 4.5: Ballet walk data.
Four frames from the original straight line walk as shown at the movie ballet-original.mpg.

The original sequence in this example was too short for our purposes. It was 350 frames long, and we needed a number of frames that were a power of two. So we scaled the sequence to be 512 frames long and shrank it back after the synthesis.

As mentioned in section 3.10, a new path is specified by manipulating only the root trajectories. Of course this is not the ideal way to change a path. A step in a circular walk is different from a step while walking in a straight line, the feet must pass longer distance in a circular motion. With our method we do not change the feet trajectories and as a result we get foot-skate artifacts, which can be noticed at seconds 19 through 22 of the movie ballet-eight.mpg. The best way to avoid these artifacts is to use a path editing algorithm, such as [13]. One can also use the foot-skate cleanup algorithm [17] after using our algorithm.

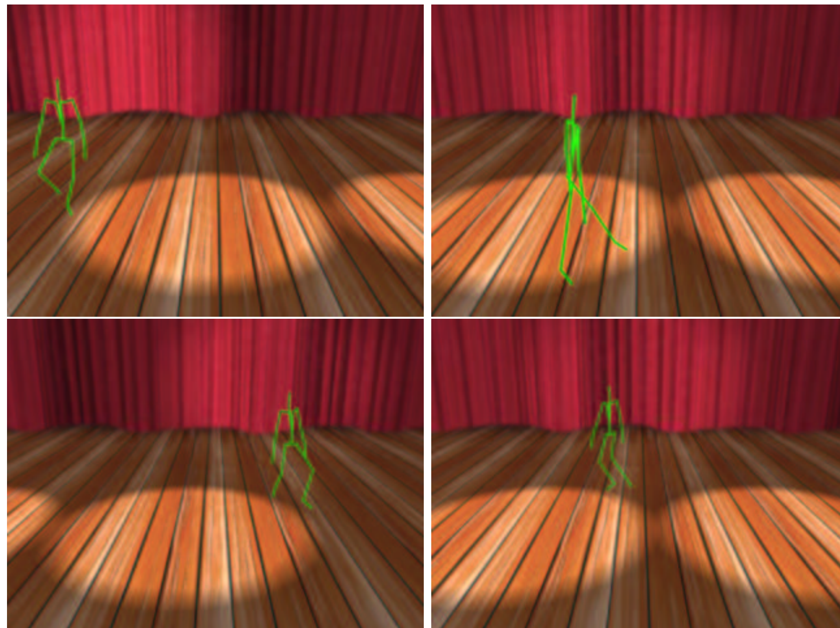

Figure 4.6: Ballet walk synthesized.
Four frames from the synthesized motion. We used a user specified animation path for the root trajectories. The path is a loop along an 8-shape (ballet-eight.mpg).

Limitations

As expected, our method appears to work best for motions containing many small segments similar to each other. For more complex motions with fewer repetitions it is more difficult with our method to find natural looking transitions between pairs of constraints. Thus, the animator must sometimes choose the constraints carefully For example, if two frames far apart from each other in the original sequence are forced to appear too close to each other in the new sequence, our method might not be able to find a natural looking transition between them (see section 3.9 for more details).

Another limitation of our method is that it can be difficult to get interesting results from short input sequences.

Our method does not currently ensure various typically desirable properties, such as that the feet of a character do not penetrate the ground while walking. Such properties are best ensured using some of the previous constraint-based methods described in chapter 2. Thus, we believe that in practice our method should be used in conjunction with these other techniques. For example, an animator might generate a new walking sequence using our technique and then apply spacetime constraints to ensure that the lowest point reached by the character's feet in each step cycle lies exactly on the ground, or he might use Kovar and Gleicher's [17] work for footskate cleanup.

Chapter 5

Conclusions and future work

We have described a new tool for generating constrained variations from existing captured or otherwise animated textural motion sequences. Our technique is not intended as a replacement for previously developed tools for motion editing; rather it is meant to complement them, adding a new useful component into the animator's toolbox.

The graph path finding algorithm for gap filling seems to work faster and it is easier for the animator, since he/she needs to set fewer constrains. But with this algorithm, the animator has less control over the outcome of the animation. Though we might get more than one meeting point inside a gap with this method, it is encouraging paths with lower total cost. So only when it is absolutely impossible to get a reasonable path with one meeting point - we might get more meeting points.

The graph path finding algorithm seems to work better on big databases, as explained in section 2.5. As we have seen in the results 4, the meeting point search algorithm can work very well even if the input sequence is short and has only few good transitions between frames.

In future work, one might try to extend our method to mix together elements from several different input motion sequences, perhaps in a manner similar to the texture mixing algorithm of Bar-Joseph *et al.* [3]. We also plan to consider other, more sophisticated, types of constraints, such as the soft constrains described by Arikan and Forsyth [2].

Acknowledgments

The motion capture sequences used as input in our experiments were obtained from Help3D.COM, Inc. (http://www.help3d.com) in BVH format and imported into Maya using the Dominatrix plug-in by House of Moves. The models were downloaded from 3DCAFE (http://www.3dcafe.com) and 3Dbuzz.com (http://www.3dbuzz.com).

Bibliography

[1] Alias|Wavefront. Maya 4.0. Modeling and animation software, 2001.

[2] O. Arikan and D. A. Forsyth. Interactive motion generation from examples. In *Proc. SIGGRAPH 2002*, Annual Conference Series, July 2002.

[3] Z. Bar-Joseph, R. El-Yaniv, D. Lischinski, and M. Werman. Texture mixing and texture movie synthesis using statistical learning. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):120–135, Apr./June 2001.

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

[5] M. Brand and A. Hertzmann. Style machines. In *Proc. SIGGRAPH 2000*, Annual Conference Series, pages 183–192, 2000.

[6] A. Bruderlin and L. Williams. Motion signal processing. In *Proc. SIGGRAPH 95*, Annual Conference Series, pages 97–104, Aug. 1995.

[7] M. F. Cohen. Interactive spacetime control for animation. *Computer Graphics (Proc. SIGGRAPH 92)*, 26(2):293–302, 1992.

[8] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proc. SIGGRAPH 97*, Annual Conference Series, pages 361–368, 1997.

[9] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proc. SIGGRAPH 2001*, Annual Conference Series, pages 341–346, Aug. 2001.

[10] M. Gleicher. Motion editing with spacetime constraints. In *Proc. 1997 ACM Symposium on Interactive 3D Graphics*, pages 139–148, Apr. 1997.

[11] M. Gleicher. Retargeting motion to new characters. In *Proc. SIGGRAPH 98*, Annual Conference Series, pages 33–42, July 1998.

[12] M. Gleicher. Comparing constraint-based motion editing methods. *Graphical Models*, 63:107–134, 2001.

[13] M. Gleicher. Motion path editing. In *Proc. 2001 ACM Symposium on Interactive 3D Graphics*, Mar. 2001.

[14] M. Gleicher and P. Litwinowicz. Constraint-based motion adaptation. *The Journal of Visualization and Computer Animation*, 9(2):65–94, 1998.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[16] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. In *Proc. SIGGRAPH 2002*, Annual Conference Series, July 2002.

[17] L. Kovar, J. Schreiner, and M. Gleicher. Footskate cleanup for motion capture editing. In *"Proc. 2002 ACM Symposium on Computer Animation"*, 2002.

[18] J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. S. Pollard. Interactive control of avatars animated with human motion data. In *Proc. SIGGRAPH 2002*, Annual Conference Series, July 2002.

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

[20] Y. Li, T. Wang, and H.-Y. Shum. Interactive motion generation from examples. In *Proc. SIGGRAPH 2002*, Annual Conference Series, July 2002.

[21] L. Molina Tanco and A. Hilton. Realistic synthesis of novel human movements from a database of motion capture examples. In *Proc. IEEE Workshop on Human Motion*, 2000.

[22] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proc. SIGGRAPH 96*, pages 205–216, 1996.

[23] K. Pullen and C. Bregler. Animating by multi-level sampling. In *Proc. Computer Animation 2000*, May 2000.

[24] K. Pullen and C. Bregler. Motion capture assisted animation: Texturing and synthesis. In *Proc. SIGGRAPH 2002*, Annual Conference Series, July 2002.

[25] A. Schödl, R. Szeliski, D. H. Salesin, and I. Essa. Video textures. In *Proc. SIGGRAPH 2000*, Annual Conference Series, pages 489–498, July 2000.

[26] M. Unuma, K. Anjyo, and R. Takeuchi. Fourier principles for emotion-based human figure animation. In *Proc. SIGGRAPH 95*, Annual Conference Series, pages 91–96, Aug. 1995.

[27] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proc. SIGGRAPH 2000*, Annual Conference Series, pages 479–488, July 2000.

[28] A. Witkin and M. Kass. Spacetime constraints. *Computer Graphics (Proc. SIGGRAPH 88)*, 22(4):159–168, Aug. 1988.

[29] A. Witkin and Z. Popović. Motion warping. In *Proc. SIGGRAPH 95*, Annual Conference Series, pages 105–108, Aug. 1995.

[30] T. Zhao, T. Wang, and H.-Y. Shum. Learning a highly structured motion model for 3d human tracking. In *Proc. ACCV 2002*, Melbourne, Australia, Jan. 2002.