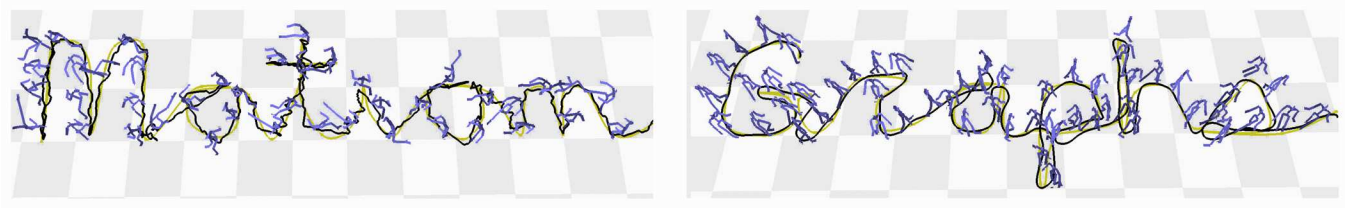
Motion Graphs

Lucas Kovar
University of Wisconsin-Madison

Michael Gleicher*
University of Wisconsin-Madison

Frédéric Pighin†
University of Southern California
Institute for Creative Technologies



Abstract

In this paper we present a novel method for creating realistic, controllable motion. Given a corpus of motion capture data, we automatically construct a directed graph called a *motion graph* that encapsulates connections among the database. The motion graph consists both of pieces of original motion and automatically generated transitions. Motion can be generated simply by building walks on the graph. We present a general framework for extracting particular graph walks that meet a user's specifications. We then show how this framework can be applied to the specific problem of generating different styles of locomotion along arbitrary paths.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: motion synthesis, motion capture, animation with constraints

1 Introduction

Realistic human motion is an important part of media like video games and movies. More lifelike characters make for more immersive environments and more believable special effects. At the same time, realistic animation of human motion is a challenging task, as people have proven to be adept at discerning the subtleties of human movement and identifying inaccuracies.

One common solution to this problem is motion capture. However, while motion capture is a reliable way of acquiring realistic human motion, by itself it is a technique for *reproducing* motion. Motion capture data has proven to be difficult to modify, and editing techniques are reliable only for small changes to a motion. This limits the utility of motion capture — if the data on hand isn't sufficiently similar to what is desired, then often there is little that can be done other than acquire more data, a time-consuming and expensive process. This in particular is a problem for applications that require motion to be synthesized dynamically, such as interactive environments.

Our goal is to retain the realism of motion capture while also giving a user the ability to control and direct a character. For example, we would like to be able to ask a character to walk around a room without worrying about having a piece of motion data that contains the correct number of steps and travels in the right directions. We also need to be able to direct characters who can perform multiple actions, rather than those who are only capable of walking around.

This paper presents a method for synthesizing streams of motions based on a corpus of captured movement while preserving the quality of the original data. Given a set of motion capture data, we compile a structure called a *motion graph* that encodes how the captured clips may be re-assembled in different ways. The motion graph is a directed graph wherein edges contain either pieces of original motion data or automatically generated transitions. The nodes then serve as choice points where these small bits of motion join seamlessly. Because our methods automatically detect and create transitions between motions, users needn't capture motions specifically designed to connect to one another. If desired, the user can tune the high-level structure of the motion graph to produce desired degrees of connectivity among different parts.

Motion graphs transform the motion synthesis problem into one of selecting sequences of nodes, or *graph walks*. By drawing upon algorithms from graph theory and AI planning, we can extract graph walks that satisfy certain properties, thereby giving us control over the synthesized motions.

To demonstrate the potential of our approach, we introduce a simple example. We were donated 78.5 seconds of motion capture, or about 2400 frames of animation, of a performer randomly walking around with both sharp and smooth turns. Since the motion was donated, we did not carefully plan out each movement, as the literature suggests is critical to successful application of motion capture data [Washburn 2001]. From this data we constructed a motion graph and used an algorithm described later in this paper to extract motions that travelled along paths sketched on the ground. Characteristic movements of the original data like sharp turns were automatically used when appropriate, as seen in Figure 1.

It is possible to place additional constraints on the desired motion. For example, we noticed that part of the motion had the character sneaking around. By labelling these frames as special, we were able to specify that at certain points along the path the character must only use sneaking movements, and at other parts of the motion it must use normal walking motions, as is also shown in Figure 1.

*e-mail:{kovar,gleicher}@cs.wisc.edu
†e-mail:pighin@ict.usc.edu

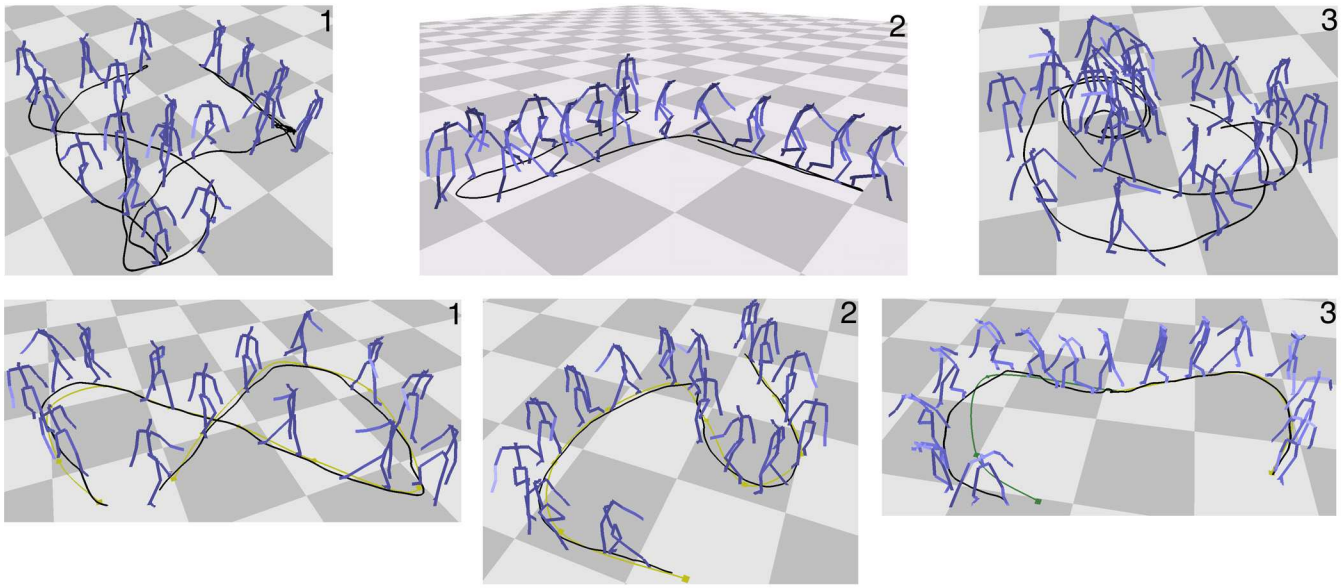


Figure 1: The top images show original motion capture data; two are walking motions and one is a sneaking motion. The black curves show the paths travelled by the character. The bottom images show new motion generated by a motion graph built out of these examples plus their mirror images. Images 1 and 2 show the result of having the motion graph fit walking motion to the smooth yellow paths. The black curve is the actual position of the center of mass on each frame. Image 3 shows motion formed by having the character switch from walking to sneaking halfway down the path.

The remainder of this paper is organized as follows. In Section 2 we describe related work. In Section 3 we describe how a motion graph is constructed from a database of motion capture. In Section 4 we set forth a general framework for extracting motion from the motion graph that meets user specifications. Section 5 discusses the specific problem of generating movements along a path and how it is handled in our framework. We conclude in Section 6 with a discussion of the scalability of our approach to large data sets and potential future work.

2 Related Work

Much previous work with motion capture has revolved around editing individual clips of motion. Motion warping [Witkin and Popović 1995] can be used to smoothly add small changes to a motion. Retargeting [Gleicher 1998; Lee and Shin 1999] maps the motion of a performer to a character of different proportions while retaining important constraints like footplants. Various signal processing operations [Bruderlin and Williams 1995] can be applied to motion data. Our work is different from these efforts in that it involves creating continuous streams of motion, rather than modifying specific clips.

One strategy for motion synthesis is to perform multi-target blends among a set of examples, yielding a continuous space of parameterized motion. Wiley and Hahn [1997] used linear interpolation to create parameterizations of walking at various inclinations and reaching to various locations. Rose et al. [1998] used radial basis functions to blend among clips representing the same motion performed in different styles. These works have a focus complementary to ours: while they are mainly concerned with generating parameterizations of individual clips, we are concerned with constructing controllable sequences of clips.

Another popular approach to motion synthesis is to construct statistical models. Pullen and Bregler [2000] used kernel-based probability distributions to synthesize new motion based on the statistical properties of example motion. Coherency was added to the model by explicitly accounting for correlations between parameters. Bowden [2000], Galata et al. [2001], and Brand and Hertzmann [2000] all processed motion capture data by constructing abstract "states" which each represent entire sets of poses. Transition probabilities between states were used to drive motion synthesis. Since these statistical models synthesize motion based on abstractions of data rather than actual data, they risk losing important detail. In our work we have tighter guarantees on the quality of generated motion. Moreover, these systems did not focus on the satisfaction of high-level constraints.

We generate motion by piecing together example motions from a database. Numerous other researchers have pursued similar strategies. Perlin [1995] and Perlin and Goldberg [1996] used a rule-based system and simple blends to attach procedurally generated motion into coherent streams. Faloutsos et al. [2001] used support vector machines to create motion sequences as compositions of actions generated from a set of physically based controllers. Since our system involves motion capture data, rather than procedural or physically based motion, we require different approaches to identifying and generating transitions. Also, these systems were mainly concerned with appropriately generating individual transitions, whereas we address the problem of generating entire motions (with many transitions) that meet user-specified criteria. Lamouret and van de Panne [1996] developed a system that used a database to extract motion meeting high-level constraints. However, their system was applied to a simple agent with five degrees of freedom, whereas we generate motion for a far more sophisticated character. Molina-Tanco and Hilton [2000] used a state-based statistical model similar to those mentioned in the previous paragraph to rearrange segments of original motion data. These segments were attached using linear interpolation. The user could create motion by selecting keyframe poses, which were connected with a high-probability sequence of states. Our work considers more general and sophisticated sets of constraints.

Work similar to ours has been done in the gaming industry to meet the requirements of online motion generation. Many companies use

move trees [Mizuguchi et al. 2001], which (like motion graphs) are graph structures representing connections in a database of motion. However, move trees are created manually — short motion clips are collected in carefully scripted capture sessions and blends are created by hand using interactive tools. Motion graphs are constructed automatically. Also, move trees are typically geared for rudimentary motion planning ("I want to turn left, so I should follow this transition"), as opposed to more complicated objectives.

The generation of transitions is an important part of our approach. Early work in this area was done by Perlin [1995], who presented a simple method for smoothly interpolating between two clips to create a blend. Lee [2000] defined orientation filters that allowed these blending operations to be performed on rotational data in a more principled fashion. Rose et al. [1996] presented a more complex method for creating transitions that preserved kinematic constraints and basic dynamic properties.

Our main application of motion graphs is to control a character's locomotion. This problem is important enough to have received a great deal of prior attention. Because a character's path isn't generally known in advance, synthesis is required. Procedural and physically based synthesis methods have been developed for a few activities such as walking [Multon et al. 1999; Sun and Metaxas 2001] and running [Hodgins et al. 1995; Bruderlin and Calvert 1996]. While techniques such as these can generate flexible motion paths, the current range of movement styles is limited. Also, these methods do not produce the quality of motion attainable by hand animation or motion capture. While Gleicher [2001] presented a method for editing the path traversed in a clip of motion capture, it did not address the need for continuous streams of motion, nor could it choose which clip is correct to fit a path (e.g. that a turning motion is better when we have a curved path).

Our basic approach — detecting transitions, constructing a graph, and using graph search techniques to find sequences satisfying user demands — has been applied previously to other problems. Schödl et al. [2000] developed a similar method for synthesizing seamless streams of video from example footage and driving these streams according to high-level user input.

Since writing this paper, we have learned of similar work done concurrently by a number of research groups. Arikan and Forsythe [2002] constructed from a motion database a hierarchical graph similar to ours and used a randomized search algorithm to extract motion that meets user constraints. Lee et al. [2002] also constructed a graph and generated motion via three user interfaces: a list of choices, a sketch-based interface similar to what we use for path fitting (Section 5), and a live video feed. Pullen and Bregler [2002] keyframed a subset of a character's degrees of freedom and matched small segments of this keyframed animation with the lower frequency bands of motion data. This resulted in sequences of short clips forming complete motions. Li et al [2002] generated a two-level statistical model of motion. At the lower level were linear dynamic systems representing characteristic movements called "textons", and the higher level contained transition probabilities among textons. This model was used both to generate new motion based on user keyframes and to edit existing motion.

3 Motion Graph Construction

In this section, we define the motion graph structure and the procedure for constructing it from a database of clips.

A clip of motion is defined as a regular sampling of the character's parameters, which consist of the position of the root joint and quaternions representing the orientations of each joint. We also allow clips (or, more generally, sets of frames) to be annotated with other information, such as descriptive labels ("walking," "karate") and constraint information (left heel must be planted on these frames).

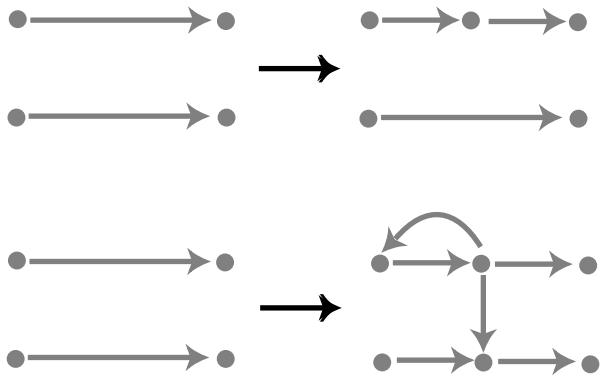


Figure 2: Consider a motion graph built from two initial clips. (top) We can trivially insert a node to divide an initial clip into two smaller clips. (bottom) We can also insert a transition joining either two different initial clips or different parts of the same initial clip.

A motion graph is a directed graph where all edges correspond to clips of motion. Nodes serve as choice points connecting these clips, i.e., each outgoing edge is potentially the successor to any incoming edge. A trivial motion graph can be created by placing all the initial clips from the database as arcs in the graph. This creates a disconnected graph with $2n$ nodes, one at the beginning and end of each clip. Similarly, an initial clip can be broken into two clips by inserting a node, since the later part of the motion is a valid successor to the earlier part (see Figure 2).

A more interesting graph requires greater connectivity. For a node to have multiple outgoing edges, there must be multiple clips that can follow the clip(s) leading into the node. Since it is unlikely that two pieces of original data are sufficiently similar, we need to create clips expressly for this purpose. *Transitions* are clips designed such that they can seamlessly connect two segments of original data. By introducing nodes within the initial clips and inserting transition clips between otherwise disconnected nodes, we can create a well-connected structure with a wide range of possible graph walks (see Figure 2).

Unfortunately, creating transitions is a hard animation problem. Imagine, for example, creating a transition between a run and a backflip. In real life this would require several seconds for an athlete to perform, and the transition motion looks little like the motions it connects. Hence the problem of automatically creating such a transition is arguably as difficult as that of creating realistic motion in the first place. On the other hand, if two motions are "close" to each other then simple blending techniques can reliably generate a transition. In light of this, our strategy is to identify portions of the initial clips that are sufficiently similar that straightforward blending is almost certain to produce valid transitions.

The remainder of this section is divided into three parts. First we describe our algorithm for detecting a set of candidate transition points. In the following two sections we discuss how we select among these candidate transitions and how blends are created at the chosen transition points. Finally, we explain how to prune the graph to eliminate problematic edges.

3.1 Detecting Candidate Transitions

As in our system, motion capture data is typically represented as vectors of parameters specifying the root position and joint rotations of a skeleton on each frame. One might attempt to locate transition points by computing some vector norm to measure the difference between poses at each pair of frames. However, such a simple approach is ill-advised, as it fails to address a number of important issues:

1. Simple vector norms fail to account for the meanings of the parameters. Specifically, in the joint angle representation some parameters have a much greater overall effect on the character than others (e.g., hip orientation vs. wrist orientation). Moreover, there is no meaningful way to assign fixed weights to these parameters, as the effect of a joint rotation on the shape of the body depends on the current configuration of the body.
2. A motion is defined only up to a rigid 2D coordinate transformation. That is, the motion is fundamentally unchanged if we translate it along the floor plane or rotate it about the vertical axis. Hence comparing two motions requires identifying compatible coordinate systems.
3. Smooth blends require more information than can be obtained at individual frames. A seamless transition must account not only for differences in body posture, but also in joint velocities, accelerations, and possibly higher-order derivatives.

Our similarity metric incorporates each of these considerations. To motivate it, we note that the skeleton is only a means to an end. In a typical animation, a polygonal mesh is deformed according to the skeleton's pose. This mesh is all that is seen, and hence it is a natural focus when considering how close two frames of animation are to each other. For this reason we measure the distance between two frames of animation in terms of a point cloud driven by the skeleton. Ideally this point cloud is a downsampling of the mesh defining the character.

To calculate the distance $D(\mathscr{A}_i, \mathscr{B}_j)$ between two frames $\mathscr{A}_i$ and $\mathscr{B}_j$, we consider the point clouds formed over two windows of frames of user-defined length k, one bordered at the beginning by $\mathscr{A}_i$ and the other bordered at the end by $\mathscr{B}_j$. That is, each point cloud is the composition of smaller point clouds representing the pose at each frame in the window. The use of windows of frames effectively incorporates derivative information into the metric, and is similar to the approach in [Schödl et al. 2000]. The size of the windows are the same as the length of the transitions, so $D(\mathscr{A}_i, \mathscr{B}_j)$ is affected by every pair of frames that form the transition. We use a value of k corresponding to a window of about a third of a second in length, as in [Mizuguchi et al. 2001]

The distance between $\mathscr{A}_i$ and $\mathscr{B}_j$ may be calculated by computing a weighted sum of squared distances between corresponding points $\mathbf{p_i}$ and $\mathbf{p_i'}$ in the two point clouds. To address the problem of finding coordinate systems for these point clouds (item 2 in the above list), we calculate the *minimal* weighted sum of squared distances given that an arbitrary rigid 2D transformation may be applied to the second point cloud:

$$\min_{\theta, x_0, z_0} \sum_i w_i \|\mathbf{p_i} - \mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}} \mathbf{p_i'}\|^2 \tag{1}$$

where the linear transformation $\mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}}$ rotates a point $\mathbf{p}$ about the y (vertical) axis by θ degrees and then translates it by (x_0, z_0). The index is over the number of points in each point cloud. The weights w_i may be chosen both to assign more importance to certain joints (e.g., those with constraints) and to taper off towards the end of the window.

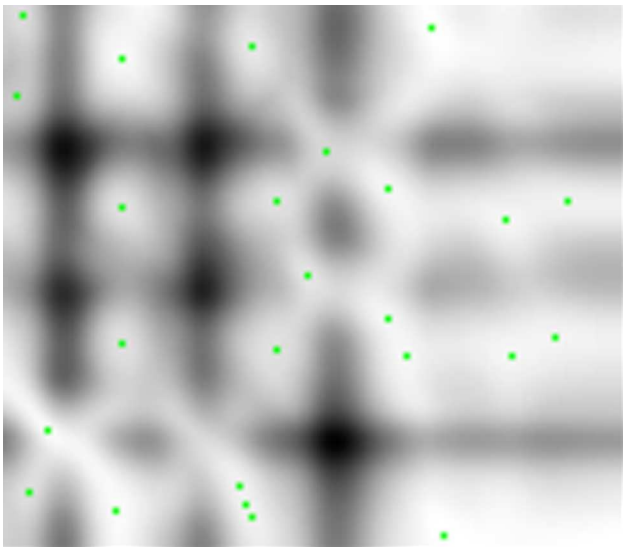


Figure 3: An example error function for two motions. The entry at (i, j) contains the error for making a transition from the i^{th} frame of the first motion to the j^{th} frame of the second. White values correspond to lower errors and black values to higher errors. The colored dots represent local minima.

This optimization has a closed-form solution:

$$\theta = \arctan \frac{\sum_i w_i (x_i z_i' - x_i' z_i) - \frac{1}{\sum_i w_i} (\overline{x}\,\overline{z'} - \overline{x'}\,\overline{z})}{\sum_i w_i (x_i x_i' + z_i z_i') - \frac{1}{\sum_i w_i} (\overline{x}\,\overline{x'} + \overline{z}\,\overline{z'})} \tag{2}$$

$$x_0 = \frac{1}{\sum_i w_i} (\overline{x} - \overline{x'} \cos(\theta) - \overline{z'} \sin\theta) \tag{3}$$

$$z_0 = \frac{1}{\sum_i w_i} (\overline{z} + \overline{x'} \sin(\theta) - \overline{z'} \cos\theta) \tag{4}$$

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

We compute the distance as defined above for every pair of frames in the database, forming a sampled 2D error function. Figure 3 shows a typical result. To make our transition model more compact, we find all the local minima of this error function, thereby extracting the "sweet spots" at which transitions are locally the most opportune. This tactic was also used in [Schödl et al. 2000]. These local minima are our candidate transition points.

3.2 Selecting Transition Points

A local minimum in the distance function does not necessarily imply a high-quality transition; it only implies a transition better than its neighbors. We are specifically interested in local minima with small error values. The simplest approach is to only accept local minima below an empirically determined threshold. This can be done without user intervention. However, often users will want to

set the threshold themselves to pick an acceptable tradeoff between having good transitions (low threshold) and having high connectivity (high threshold).

Different kinds of motions have different fidelity requirements. For example, walking motions have very exacting requirements on the transitions — people have seen others walk nearly every day since birth and consequently have a keen sense of what a walk should look like. On the other hand, most people are less familiar with ballet motions and would be less likely to detect inaccuracies in such motion. As a result, we allow a user to apply different thresholds to different pairs of motions; transitions among ballet motions may have a higher acceptance threshold than transitions among walking motions.

3.3 Creating Transitions

If $D(\mathscr{A}_i, \mathscr{B}_j)$ meets the threshold requirements, we create a transition by blending frames $\mathscr{A}_i$ to $\mathscr{A}_{i+k-1}$ with frames $\mathscr{B}_{j-k+1}$ to $\mathscr{B}_j$, inclusive. The first step is to apply the appropriate aligning 2D transformation to motion $\mathscr{B}$. Then on frame p of the transition $(0 \le p < k)$ we linearly interpolate the root positions and perform spherical linear interpolation on joint rotations:

$$R_p = \alpha(p) R_{\mathscr{A}_{i+p}} + [1-\alpha(p)] R_{\mathscr{B}_{j-k+1+p}} \tag{5}$$

$$q^i_p = slerp(q^i_{\mathscr{A}_{i+p}}, q^i_{\mathscr{B}_{j-k+1+p}}, \alpha(p)) \tag{6}$$

where R_p is the root position on the p^{th} transition frame and q^i_p is the rotation of the i^{th} joint on the p^{th} transition frame.

To maintain continuity we choose the blend weights $\alpha(p)$ according to the conditions that $\alpha(p) = 1$ for $p \le -1$, $\alpha(p) = 0$ for $p \ge k$, and that $\alpha(p)$ has C^1 continuity everywhere. This requires

$$\alpha(p) = 2\left(\frac{p+1}{k}\right)^3 - 3\left(\frac{p+1}{k}\right)^2 + 1, \; -1 < p < k \tag{7}$$

Other transition schemes, such as [Rose et al. 1996], may be used in place of this one.

The use of linear blends means that constraints in the original motion may be violated. For example, one of the character's feet may slide when it ought to be planted. This can be corrected by using constraint annotations in the original motions. We treat constraints as binary signals: on a given frame a particular constraint either exists or it does not. Blending these signals in analogy to equations 5 and 6 amounts to using the constraints from $\mathscr{A}$ in the first half of the transition and the constraints from $\mathscr{B}$ in the second half. In this manner each transition is automatically annotated with constraint information, and these constraints may later be enforced as a post-processing step when motion is extracted form the graph. We will discuss constraint enforcement in more detail in the next section.

Descriptive labels attached to the motions are carried along into transitions. Specifically, if a transition frame is a blend between a frame with a set of labels L_1 and another frame with a set of labels L_2, then it has the union of these labels $L_1 \cup L_2$.

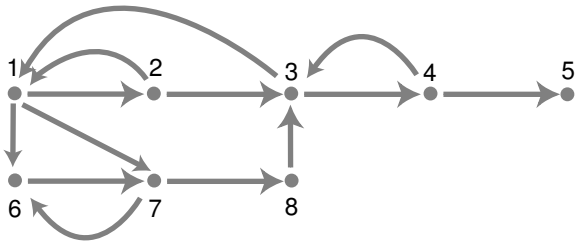


Figure 4: A simple motion graph. The largest strongly connected component is $[1,2,3,6,7,8]$. Node 4 is a sink and 5 is a dead end.

3.4 Pruning The Graph

In its current state there are no guarantees that the graph can synthesize motion indefinitely, since there may be nodes (called *dead ends*) that are not part of any cycle (see Figure 4). Once such a node is entered there is a bound on how much additional motion can be generated. Other nodes (called *sinks*) may be part of one or more cycles but nonetheless only be able to reach a small fraction of the total number of nodes in the graph. While arbitrarily long motion may still be generated once a sink is entered, this motion is confined to a small part of the database. Finally, some nodes may have incoming edges such that no outgoing edges contain the same set of descriptive labels. This is dangerous since logical discontinuities may be forced into a motion. For example, a character currently in a "boxing" motion may have no choice but to transition to a "ballet" motion.

To address these problems, we prune the graph such that, starting from any edge, it is possible to generate arbitrarily long streams of motion of the same type such that as much of the database as possible is used. This is done as follows. Every frame of original data is associated with a (possibly empty) set of labels. Say there are n unique sets. For each set, form the subgraph consisting of all edges whose frames have exactly this set of labels. Compute the strongly connected components (SCCs) of this subgraph, where an SCC is a maximal set of nodes such that there is a connecting graph walk for any ordered pair of nodes (u,v). The SCCs can be computed in $O(V+E)$ time using an algorithm due to Tarjan. We eliminate from this subgraph (and hence the original motion graph) any edge that does not attach two nodes in the largest SCC. Once this process is completed for all n label sets, any nodes with no edges are discarded.

A warning is given to the user if the largest SCC for a given set of labels contains below a threshold number of frames. Also, a warning is given if for any ordered pair of SCCs there is no way to transition from the first to the second. In either case, the user may wish to adjust the transition thresholds (Section 3.2) to give the graph greater connectivity.

4 Extracting Motion

By this stage we have finished constructing the motion graph. After describing exactly how a graph walk can be converted into displayable motion, we will consider the general problem of extracting motion that satisfies user constraints. Our algorithm involves solving an optimization problem, and so we conclude this section with some general recommendations on how to pose the optimization.

4.1 Converting Graph Walks To Motion

Since every edge on the motion graph is a piece of motion, a graph walk corresponds to a motion generated by placing these pieces one after another. The only issue is to place each piece in the correct location and orientation. In other words, each frame must be transformed by an appropriate 2D rigid transformation. At the start of a graph walk this transformation is the identity. Whenever we exit a transition edge, the current transformation is multiplied by the transformation that aligned the pieces of motion connected by the transition (Section 3.1).

As noted in Section 3.3, the use of linear blends to create transitions can cause artifacts, the most common of which is feet that slide when they ought to be planted. However, every graph walk is automatically annotated with constraint information (such as that the foot must be planted). These constraints are either specified directly in the original motions or generated as in Section 3.3, depending on whether the frame is original data or a transition. These constraints may be satisfied using a variety of methods, such as [Gleicher 1998] or [Lee and Shin 1999]. In our work we used the method described in [Kovar et al. 2002].

4.2 Searching For Motion

We are now in a position to consider the problem of finding motion that satisfies user-specified requirements. It is worth first noting that only very special graph walks are likely to be useful. For example, while a random graph walk will generate a continuous stream of motion, such an algorithm has little use other than an elaborate screen saver. As a more detailed example, consider computing an all-pairs shortest graph walk table for the graph. That is, given a suitable metric — say, time elapsed or distance travelled — we can use standard graph algorithms like Floyd-Warshall to find for each pair of nodes u and v the connecting graph walk that minimizes the metric. With this in hand we could, for example, generate the motion that connects one clip to another as quickly as possible. This is less useful than it might appear at first. First, there are no guarantees that the shortest graph walk is short in an absolute sense. In our larger test graphs (between a few and several thousand nodes) the average shortest path between any two nodes was on the order of two seconds. This is not because the graphs were poorly connected. Since the transitions were about one-third of a second apiece, this means there were on average only five or six transitions separating any two of the thousands of nodes. Second, there is no control over what happens during the graph walk — we can't specify what direction the character travels in or where she ends up.

More generally, the sorts of motions that a user is likely to be interested in probably don't involve minimizing metrics as simple as total elapsed time. However, for complicated metrics there is typically no simple way of finding the globally optimal graph walk. Hence we focus instead on local search methods that try to find a *satisfactory* graph walk within a reasonable amount of time.

We now present our framework for extracting graph walks that conform to a user's specifications. We cast motion extraction as a search problem and use branch and bound to increase the efficiency of this search. The user supplies a scalar function $g(w,e)$ that evaluates the additional error accrued by appending an edge e to the existing path w, which may be the empty path $\emptyset$. The total error $f(w)$ of the path is defined as follows:

$$f(w) = f([e_1,\ldots,e_n]) = \sum_{i=1}^{n} g([e_1,\ldots,e_{i-1}],e_i) \qquad (8)$$

where w is comprised of the edges $e_1,\ldots,e_n$. We require $g(w,e)$ to be nonnegative, which means that we can never decrease the total error by adding more edges to a graph walk.

In addition to f and g, the user must also supply a halting condition indicating when no additional edges should be added to a graph walk. A graph walk satisfying the halting condition is called *complete*. The start of the graph walk may either be specified by the user or chosen at random.

Our goal is find a complete graph walk w that minimizes f. To give the user control over what sorts of motions should be considered in the search, we allow restrictions on what edges may be appended to a given walk w. For example, the user may decide that within a particular window of time a graph walk may only contain "sneaking" edges.

A naïve solution is to use depth-first search to evaluate f for all complete graph walks and then select the best one. However, the number of possible graph walks grows exponentially with the average size of a complete graph walk. To address this we use a branch and bound strategy to cull branches of the search that are incapable of yielding a minimum. Since $g(w,e)$ by assumption never decreases, $f(w)$ is a lower bound on $f(w+v)$ for any v, where $w+v$ is the graph walk composed of v appended to w. Thus we can keep track of the current best complete graph walk w_{opt} and immediately halt any branch of the search for which the graph walk's error exceeds $f(w_{opt})$. Also, the user may define a threshold error ε such that if $f(w) < \varepsilon$, then w is considered to be "good enough" and the search is halted.

Branch and bound is most successful when we can attain a tight lower bound early in the search process. For this reason it is worthwhile to have a heuristic for ordering the edges we explore out of a particular node. One simple heuristic is to order the children greedily — that is, given a set of unexplored children $c_1,\ldots,c_n$, we search the one that minimizes $g(w,c_i)$.

While branch and bound reduces the number of graph walks we have to test against f, it does not change the fact that the search process is inherently exponential — it merely lowers the effective branching factor. For this reason we generate a graph walk incrementally. At each step we use branch and bound to find an optimal graph walk of n frames. We retain the first m frames of this graph walk and use the final retained node as a starting point for another search. This process continues until a complete graph walk is generated. In our implementation we used values of n from 80 to 120 frames ($2\frac{2}{3}$ to 4 seconds) and m from 25 to 30 frames (about one second).

Sometimes it is useful to have a degree of randomness in the search process, such as when one is animating a crowd. There are a couple of easy ways to add randomness to the search process without sacrificing a good result. The first is to select a start for the search at random. The second is retain the r best graph walks at the end of each iteration of the search and randomly pick among the ones whose error is within some tolerance of the best solution.

4.3 Deciding What To Ask For

Since the motion extracted from the graph is determined by the function g, it is worth considering what sorts of functions are likely to produce desirable results. To understand the issues involved, we consider a simple example. Imagine we want to lay down two clips on the floor and create a motion that starts at the first clip and ends at the second. Both clips must end up in the specified position and orientation. We can formally state this problem as follows: given a starting node N in the graph and a target edge e, find a graph walk

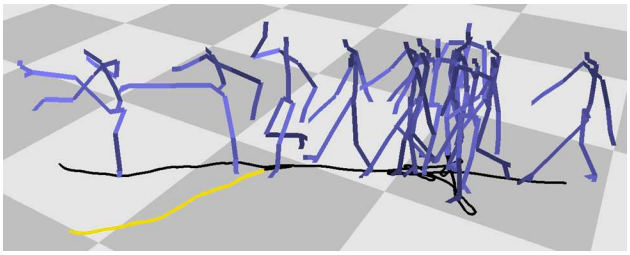


Figure 5: The above motion was generated using the search algorithm discussed in this section. The halting condition was to play a specific clip of two kicking motions. The error of a complete graph walk (which necessarily ended with the kicking clip) was determined by how far away this kicking clip was from being in a particular position and orientation. The character spends approximately seven seconds making minute adjustments to its orientation in an attempt to better align itself with the final clip. The highlighted line shows the the path of the target clip in its desired position and orientation.

that ends with e such that the transformation $\mathbf{T}$ applied to e is as close as possible to a given transformation $\mathbf{T}'$. What one will receive is a motion like in Figure 5, where the initial clip is a walking motion and the final clip is a kick. The character turns around in place several times in an attempt to better line up with the target clip.

While it's conceivable that given a larger database we would have found a better motion, the problem here is with the function we passed into the search algorithm. First, it gives no guidance as to what should be done in the middle of the motion; all that matters is that the final clip be in the right position and orientation. This means the character is allowed to do whatever is possible in order to make the final fit, even if the motion is nothing that a real person would do. Second, the goal is probably more specific than necessary. If it doesn't matter what kick the character does, then it should be allowed to choose a kick that doesn't require such effort to aim.

More generally, there are two lessons we can draw from this example. First, g should give some sort of guidance throughout the entire motion, as arbitrary motion is almost never desirable. Second, g should be no more restrictive than necessary, in order to give the search algorithm more goals to seek. Note the tradeoff here — guiding the search toward a particular result must be balanced against unduly preventing it from considering all available options.

5 Path Synthesis

We have cast motion extraction as an optimization problem, and we have given some reasons why the formulation of this optimization can be difficult. To demonstrate that it is nonetheless possible to come up with optimization criteria that allow us to solve a real problem, we apply the preceding framework to path synthesis. This problem is simple to state: given a path P specified by the user, generate motion such that the character travels along P. In this section we present our algorithm for path synthesis, present results, and discuss applications of the technique.

5.1 Implementing Path Synthesis

Given the framework in the previous section, our only tasks are to define an error function $g(w,e)$ and appropriate halting criteria. The basic idea is to estimate the actual path P' travelled by the character during a graph walk and measure how different it is from P. The graph walk is complete when P' is sufficiently long.

A simple way to determine P' is to project the root onto the floor at each frame, forming a piecewise linear curve[1]. Let $P(s)$ be the point on P whose arc-length distance from the start of P is s. The i^{th} frame of the graph walk, w_i, is at some arc length $s(w_i)$ from the start of P'. We define the corresponding point on P as the point at the same arc length, $P(s(w_i))$. For the j^{th} frame of e, we calculate the squared distance between $P'(s(e_j))$ and $P(s(e_j))$. $g(w,e)$ is the sum of these errors:

$$g(w,e) = \sum_{i=1}^{n} \|P'(s(e_i)) - P(s(e_i))\|^2 \qquad (9)$$

Note that $s(e_i)$ depends on the total arc length of w, which is why this equation is a function of w as well as e. The halting condition for path synthesis is when the current total length of P' meets or exceeds that of P. Any frames on the graph walk at an arc length longer than the total length of P are mapped to the last point on P.

The error function $g(w,e)$ was chosen for a number of reasons. First, it is efficient to compute, which is important in making the search algorithm practical. Second, the character is given incentive to make definite progress along the path. If we were to have required the character to merely be near the path, then it would have no reason not to alternate between travelling forwards and backwards. Finally, this metric allows the character to travel at whatever speed is appropriate for what needs to be done. For example, a sharp turn will not cover distance at the same rate as walking straight forward. Since both actions are equally important for accurate path synthesis, it is important that one not be given undue preference over the other.

One potential problem with this metric is that a character who stands still will never have an incentive to move forward, as it can accrue zero error by remaining in place. While we have not encountered this particular problem in practice, it can be countered by requiring at least a small amount of forward progress γ on each frame. More exactly, we can replace in Equation 9 the function $s(e_i)$ with $t(e_i) = max(t(e_{i-1}) + s(e_i) - s(e_{i-1}), t(e_{i-1}) + \gamma)$.

Typically the user will want all generated motion to be of a single type, such as walking. This corresponds to confining the search to the subgraph containing the appropriate set of descriptive labels. More interestingly, one can require different types of motion on different parts of the path. For example, one might want the character to walk along the first half of the path and sneak down the rest. The necessary modifications to accomplish this are simple. We will consider the case of two different motion types; the generalization to higher numbers is trivial.

We divide the original path into two smaller adjoining paths, P_1 and P_2, based on where the transition from type T_1 to type T_2 is to occur. If the character is currently fitting P_2, then the algorithm is identical to the single-type case. If the character is fitting P_1, then we check to see if we are a threshold distance from the end of P_1. If not, we continue to only consider edges of type T_1. Otherwise we allow the search to try both edges of type T_1 and T_2; in the latter case we switch to fitting P_2. Note that we only allow this switch to occur once on any given graph walk, which prevents the resulting motion from randomly switching between the two actions.

5.2 Results

While the examples shown in Figure 1 suggest that our technique is viable, it perhaps isn't surprising that we were able to find accurate fits to the given paths. As shown in the upper portion of the

[1] In our implementation we defined the path as a spline approximating this piecewise linear path, although this has little impact on the results.

figure, the input motion had a fair amount of variation, including straight-ahead marches, sharp turns, and smooth changes of curvature. However, our algorithm is still useful when the input database is not as rich. Refer to Figure 6. We started with a single 12.8-second clip of an actor sneaking along the indicated path. To stretch this data further, we created a mirror-image motion and then built a motion graph out of the two. From these we were able to construct the new motions shown at the bottom of the figure, both of which are themselves approximately 13 seconds in length.

Figure 7 shows fits to a more complicated path. The first example uses walking motions and the second uses martial arts motions; the latter demonstrates that our approach works even on motions that are not obviously locomotion. For the walking motion, the total computation time was nearly the same as the length of the generated animation (58.1 seconds of calculation for 54.9 seconds animation). The martial arts motion is 87.7 seconds long and required just 15.0 seconds of computation. In general, in our test cases the duration of a generated motion was either greater than or approximately equal to the amount of time needed to produce it. Both motion graphs had approximately 3000 frames (100 seconds) of animation.

Finally, Figure 8 shows paths containing constraints on the allowable motion type. In the first section of each path the character is required to walk, in the second it must sneak, and in the third it is to perform martial arts moves. Not only does the character follow the path well, but transitions between action types occur quite close to their specified locations. This example used a database of approximately 6000 frames (200 seconds).

All examples were computed on a 1.3GHz Athlon. For our largest graph (about 6000 frames), approximately twenty-five minutes were needed to compute the locations of all candidate transitions points. Approximately five minutes of user time were required to select transition thresholds, and it took less than a minute to calculate blends at these transitions and prune the resulting graph.

5.3 Applications Of Path Synthesis

Directable locomotion is a general enough need that the preceding algorithm has many applications.

Interactive Control. We can use path synthesis techniques to give a user interactive control over a character. For example, when the user hits the left arrow key the character might start travelling east. To accomplish this, we can use the path fitting algorithm to find the sequence of edges starting from our current location on the graph that best allow the character to travel east. The first edge on the resulting graph walk is the next clip that will be played. This process may then be repeated. To make this practical, we can precompute for every node in the graph a sequence of graph walks that fit straight-line paths in a sampling of directions (0 degrees, 30 degrees, ...). The first edges on these paths are then stored for later use; they are the best edges to follow given the direction the character is supposed to travel in.

High-Level Keyframing. If we want a character to perform certain actions in a specific sequence and in specific locations, we can draw a path with subsections requiring the appropriate action types. This allows us to generate complex animations without the tedium of manual keyframing. For this reason we term this process "high-level" keyframing — the user generates an animation based on what should be happening and where.

Motion Dumping. If an AI algorithm is used to determine that a character must travel along a certain path or start performing certain actions, the motion graph may be used to "dump" motion on top of the algorithm's result. Hence motion graphs may be used as a back-end for animating non-player characters in video games and interactive environments — the paths and action types can be specified by a high-level process and the motion graph would fill in the details.

Crowds. While our discussion so far has focused on a single character, there's no reason why it couldn't be applied to several characters in parallel. Motion graphs may be used as a practical tool for crowd generation. For example, a standard collision-avoidance algorithm could be used to generate a path for each individual, and the motion graph could then generate motion that conforms to this path. Moreover, we can use the techniques described at the end of Section 4.2 to add randomness to the generated motion.

6 Discussion

In this paper we have presented a framework for generating realistic, controllable motion through a database of motion capture. Our approach involves automatically constructing a graph that encapsulates connections among different pieces of motion in the database and then searching this graph for motions that satisfy user constraints. We have applied our framework to the problem of path synthesis.

As we had limited access to data, our largest examples used a database of several thousand frames of motion. While we believe this was sufficient to show the potential of our method, a character with a truly diverse set of actions might require hundreds or thousands of times more data. Hence the scalability of our framework bears discussion. The principle computational bottleneck in graph construction is locating candidate transitions (Section 3.1). This requires comparing every pair of the F frames in the database and therefore involves $O(F^2)$ operations. However, this calculation is trivial to parallelize, and distances between old frames needn't be recomputed if additions are made to the database.

It is the exception rather than the rule that two pieces of motion are sufficiently similar that a transition is possible, and hence motion graphs tend to be sparse. In our experience the necessary amount of storage is approximately proportional to the size of the database.

The number of edges leaving a node in general grows with the size of the graph, meaning the branching factor in our search algorithm may grow as well. However, we expect that future motion graphs will be larger mainly because the character will be able to perform more actions. That is, for example, having increasing amounts of walking motion isn't particularly useful once one can direct a character along nearly any path. Hence the branching factor in a particular subgraph will remain stationary once that subgraph is sufficiently large. We anticipate that typical graph searches will be restricted to one or two subgraphs, and so we expect that the search will remain practical even for larger graphs.

We conclude with a brief discussion of future work. One limitation of our approach is that the transition thresholds must be specified by hand, since (as discussed in Section 3.2) different kinds of motions have different fidelity requirements. Setting thresholds in databases involving many different kinds of motions may be overly laborious, and so we are investigating methods for automating this process. A second area of future work is to incorporate parameterizable motions [Wiley and Hahn 1997; Rose et al. 1998] into our system, rather than having every node correspond to a static piece of motion. This would add flexibility to the search process and potentially allow generated motion to better satisfy user constraints. Finally, we are interested in applying motion graphs to problems other than path synthesis.

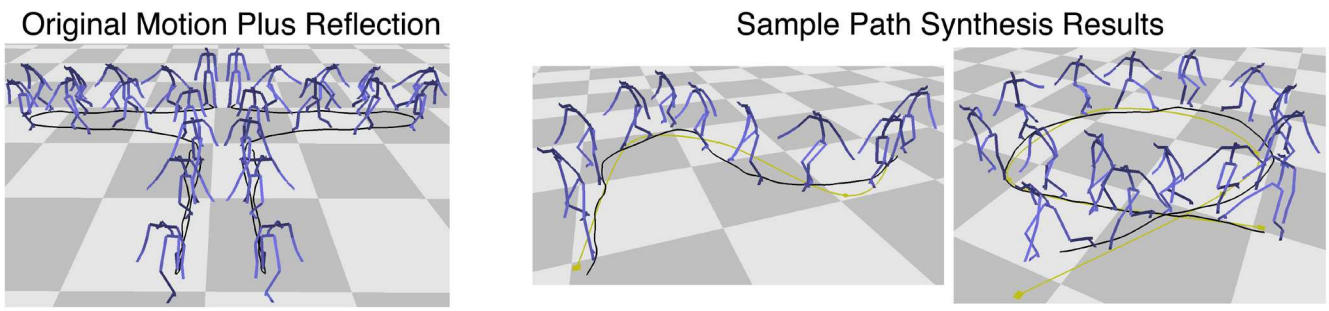


Figure 6: The leftmost image shows the original motion and its reflection and the following images show motion generated by our path synthesis algorithm. The thick yellow lines were the paths to be fit and the black line is an approximation of the actual path of the character. Note how we are able to accurately fit nontrivial paths despite the limited variation in the path of the original motion.

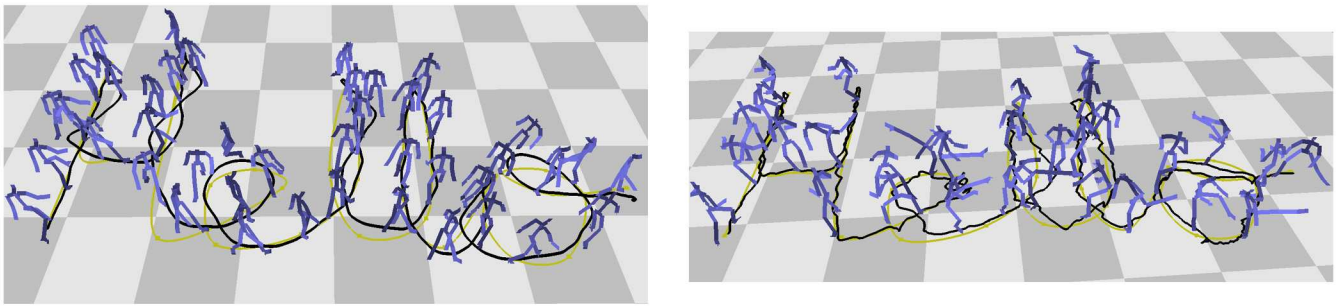


Figure 7: The left image shows a walking motion generated to fit to a path that spells "Hello" in cursive. The right image shows a karate motion fit to the same path. The total calculation time for the walking motion was 58.1 seconds and the animation itself is 54.9 seconds. The 87.7-second karate motion was computed in just 15.0 seconds. All computation was done on a 1.3gHz Athlon.

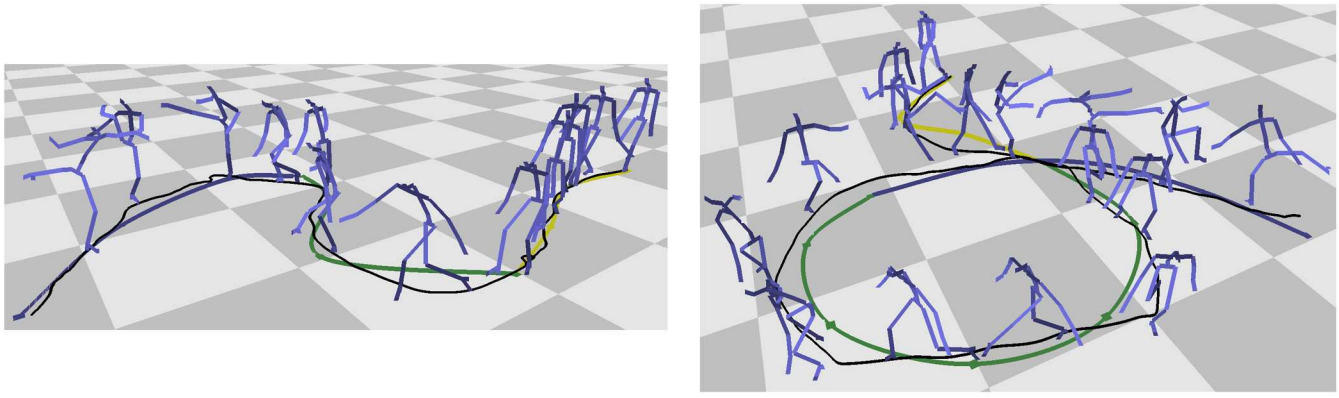


Figure 8: These images are both fits to paths wherein the character is required to walk, then sneak, and finally perform martial arts moves. The desired transition points are indicated by where the curve changes color. Note that the character both fits the path accurately and switches to the appropriate motion type close to the desired location.

Acknowledgements

We would like to acknowledge Andrew Gardner, Alex Mohr, and John Schreiner for assisting in video production, proofreading, and other technical matters. We also thank the University of Southern California's School of Film and Television for their support and the reviewers for their many useful suggestions. Our work was made possible through generous motion data donations from Spectrum Studios (particularly Demian Gordon), House of Moves, and The Ohio State University. This work was supported in part by NSF grants CCR-9984506 and IIS-0097456, the U.S. Army[2], the Wisconsin Alumni Research Fund's University Industrial Relations program, equipment donations from IBM, NVidia, and Intel, and software donations from Discreet, Alias/Wavefront, and Pixar.

References

ARIKAN, O., AND FORSYTHE, D. 2002. Interactive motion generation from examples. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, ACM SIGGRAPH.

BOWDEN, R. 2000. Learning statistical models of human motion. In *IEEE Workshop on Human Modelling, Analysis, and Synthesis, CVPR 2000*, IEEE Computer Society.

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

BRUDERLIN, A., AND CALVERT, T. 1996. Knowledge-driven, interactive animation of human running. In *Graphics Interface*, Canadian Human-Computer Communications Society, 213–221.

BRUDERLIN, A., AND WILLIAMS, L. 1995. Motion signal processing. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, ACM SIGGRAPH, 97–104.

FALOUTSOS, P., VAN DE PANNE, M., AND TERZOPOULOS, D. 2001. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH 2001*, Annual Conference Series, ACM SIGGRAPH, 251–260.

GALATA, A., JOGNSON, N., AND HOGG, D. 2001. Learning variable-length markov models of behavior. *Computer Vision and Image Understanding Journal 81*, 3, 398–413.

GLEICHER, M. 1998. Retargeting motion to new characters. In *Proceedings 0f ACM SIGGRAPH 98*, Annual Conference Series, ACM SIGGRAPH, 33–42.

GLEICHER, M. 2001. Motion path editing. In *Proceedings 2001 ACM Symposium on Interactive 3D Graphics*, ACM.

HODGINS, J. K., WOOTEN, W. L., BROGAN, D. C., AND O'BRIEN, J. F. 1995. Animating human athletics. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, ACM SIGGRAPH, 71–78.

KOVAR, L., GLEICHER, M., AND SCHREINER, J. 2002. Footskate cleanup for motion capture editing. Tech. rep., University of Wisconsin, Madison.

LAMOURET, A., AND PANNE, M. 1996. Motion synthesis by example. *Computer animation and Simulation*, 199–212.

LEE, J., AND SHIN, S. Y. 1999. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of ACM SIGGRAPH 99*, Annual Conference Series, ACM SIGGRAPH, 39–48.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, ACM SIGGRAPH.

LEE, J. 2000. *A hierarchical approach to motion analysis and synthesis for articulated figures*. PhD thesis, Department of Computer Science, Korea Advanced Institute of Science and Technology.

LI, Y., WANG, T., AND SHUM, H.-Y. 2002. Motion texture: A two-level statistical model for character motion synthesis. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, ACM SIGGRAPH.

[2]This paper does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred

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*, IEEE Computer Society, 137–142.

MULTON, F., FRANCE, L., CANI, M.-P., AND DEBUNNE, G. 1999. Computer animation of human walking: a survey. *The Journal of Visualization and Computer Animation 10*, 39–54. Published under the name Marie-Paule Cani-Gascuel.

PERLIN, K., AND GOLDBERG, A. 1996. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of ACM SIGGRAPH 96*, ACM SIGGRAPH, 205–216.

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

PULLEN, K., AND BREGLER, C. 2000. Animating by multi-level sampling. In *IEEE Computer Animation Conference*, CGS and IEEE, 36–42.

PULLEN, K., AND BREGLER, C. 2002. Motion capture assisted animation: Texturing and synthesis. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, ACM SIGGRAPH.

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

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

SCHÖDL, A., SZELISKI, R., SALESIN, D., AND ESSA, I. 2000. Video textures. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, ACM SIGGRAPH, 489–498.

SUN, H. C., AND METAXAS, D. N. 2001. Automating gait animation. In *Proceedings of ACM SIGGRAPH 2001*, Annual Conference Series, ACM SIGGRAPH, 261–270.

WASHBURN, D. 2001. The quest for pure motion capture. *Game Developer* (December).

WILEY, D., AND HAHN, J. 1997. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Application 17*, 6, 39–45.

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