Snap-Together Motion: Assembling Run-Time Animations

Michael Gleicher * Hyun Joon Shin Lucas Kovar Andrew Jepsen

Computer Sciences Department, University of Wisconsin, Madison

Abstract

Many virtual environments and games must be populated with synthetic characters to create the desired experience. These characters must move with sufficient realism, so as not to destroy the visual quality of the experience, yet be responsive, controllable, and efficient to simulate. In this paper we present an approach to character motion called *Snap-Together Motion* that addresses the unique demands of virtual environments. Snap-Together Motion (STM) preprocesses a corpus of motion capture examples into a set of short clips that can be concatenated to make continuous streams of motion. The result process is a simple graph structure that facilitates efficient planning of character motions. A user-guided process selects "common" character poses and the system automatically synthesizes multi-way transitions that connect through these poses. In this manner well-connected graphs can be constructed to suit a particular application, allowing for practical interactive control without the effort of manually specifying all transitions.

Keywords: Motion Synthesis, Virtual Environments, Motion Capture

1 Introduction

Advances in graphics hardware and rendering software have made it possible to build visually rich virtual worlds, creating possibilities for entertainment and training applications. For many of these applications, the virtual worlds must be populated with believable synthetic characters. Creating such characters is challenging. To fit with the visual richness provided by virtual environments, characters must move in realistic ways. At the same time, in order to meet interactivity demands they must also be efficiently animated and controllable by the simulation.

In this paper we introduce a methodology that allows quality motions to be synthesized in a controllable manner with little run-time overhead. A corpus of motion capture data is processed into a set of short clips that can be "snapped together" (concatenated) into seamless streams of motions at run time. This process is guided by a human author who identifies (either independently or via help from our system) character poses that appear frequently in the corpus. Each such pose serves as a jump point at which any motion that enters can be followed by any motion that leaves, as shown in Figures 1 and 2. The result is a simple graph structure that allows clips to be connected into longer motions.

*gleicher@cs.wisc.edu, http://www.cs.wisc.edu/graphics

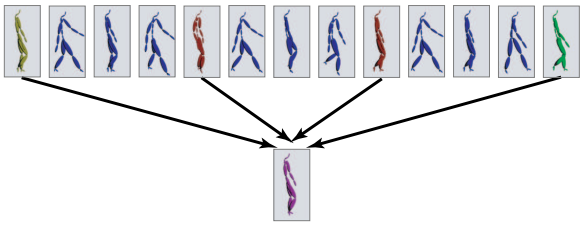

Figure 1: Schematic of the authoring process. A linear corpus of motion (here a single walking motion) has a common pose identified through a user-guided process.

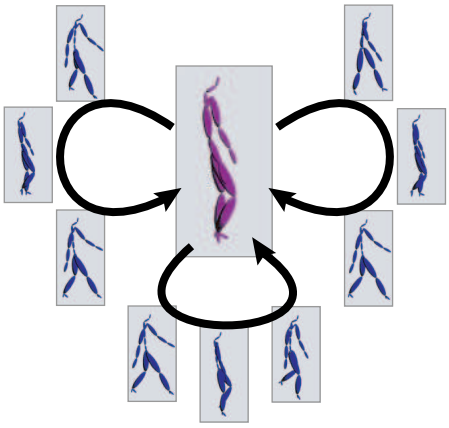

Figure 2: Transitions are generated around the common pose, forming a simple graph.

All transition generation and cleanup operations are performed automatically by our system. At run time, a character animation module need only play precomputed clips in a valid order as determined by the graph. User involvement in the graph construction process allows for the clips to connect in ways that facilitate control. That is, the animation designer guides the system into building a graph with a structure that is contrived to be easy to exploit at run time. In particular, if the designer creates a graph with a high branching factor, the run-time motion planner will have the flexibility to choose from several options when a new action must be taken.

Our approach is closely related to previous non-linear animation methods. In particular, our final graph structures are akin to the *move trees* common in computer games. The key difference is that our graphs are constructed opportunistically based on a provided data corpus and some user guidance on how to form a usable graph. In contrast, traditional move trees use specially contrived motions and hand-crafted transitions. In a sense, we provide a new approach for constructing the data structures used by existing approaches to real-time animation synthesis. The increased automation of our approach reduces the planning, effort, and skill required to author the graph structures, and it is possible to author graphs with a degree

of connectivity that would be extremely tedious to construct using traditional methods.

Our work involves two main contributions, each of which facilitate the authoring of character motion for virtual environments. First, we provide an improved authoring methodology where candidate transition points are identified automatically. This aids in the creation of graphs with a small number of *hub* nodes containing a large number of edges. We speed the process of adding clips to the graph by allowing an author to add entire hubs to a graph at a time, and we can further simplify construction by automatically suggesting hubs based on the original motion data.

The second main contribution of our work is to provide methods for generating multi-way transitions. Our framework allows *cut transitions*, which involve simply concatenating two clips together without further processing. This is done by adjusting the original motions such that these transitions are seamless, i.e., they are C_1 smooth and satisfy the appropriate constraints. The advantages of such an approach are that it keeps the resulting graph compact and allows efficient generation of motions at run time. The challenge is to connect multiple motions in a manner that avoids visual artifacts.

The remainder of this paper is divided into five sections. First, we clarify in Section 2 the limitations of current tools for constructing graphs and how we propose to address these limitations. In Section 3 we discuss related work. In Section 4 we describe our algorithms and explain how they fit into the overall process of building a graph. Finally, in Section 5 we present some example results and then we conclude in Section 6 with a discussion of the advantages and drawbacks of our approach.

2 Issues with Current Practices

In order to create streams of high-quality motion, current applications assemble static clips of motion created with traditional animation techniques such as motion capture or keyframing. The assembly process requires making transitions between motions. These transitions may be difficult to create, such as a transition between a running clip and one where the character is lying down, or trivial, if the end of one clip is identical to the beginning of the next. In practice, simple techniques such as linear blends are capable of creating transitions in cases where the motions are similar.

A set of motion clips and transitions between them form a graph where the edges are pieces of motions and the nodes are choice points connecting motions. A graph of this type called a *move tree* is common in computer games [14, 15]. Move trees are constructed by pre-planning movements such that the initial clips have similar beginning and end points. An artist then chooses the exact points in the clips where transitions are to occur and creates the transition motions. Most commercial motion editing tools, such as Character Studio, Softimage XSI, Diva, or Messiah:Animation, provide some support for applying simple transition methods (e.g., linear blends) at identified points.

The structure of a graph can have a significant impact on its usefulness. In general, the more well-connected a graph is, the more controllable the animation will be. Ideally, all clips of motions will connect, allowing any action to take place at any time. In practice, good transitions between radically different clips are prohibitively difficult to create. Tradeoffs must therefore be made between the quality of the transitions and the connectivity of the graphs.

While it may not be possible to have all clips connect directly, well-constructed graphs nonetheless typically have nodes with many incoming and outgoing edges. We call such nodes *hubs*. Hubs are desirable because they offer advantages in both flexibility and simplifying the problem of generating motion that meets high-level requirements. For example, a particular hub might contain several different kinds of punches and kicks, in which case a character could easily string together a sequence of strikes according to a high-level reasoning module (e.g., he should throw a punch combination since the opponent's guard is down). Similarly, there might be a "walking" hub that has several outgoing edges which each correspond to taking a step in some direction. Combined with jogging and running hubs, a planning module could move characters in the virtual environment simply by specifying a speed and direction.

Graphs containing hubs are difficult to construct. Authors must find places in the motion corpus where several motions come together and devise multi-way transitions, a much more difficult problem than making just two clips join smoothly. Current tools offer little support for the creation of hubs. Our framework, in contrast, explicitly supports the creation of hub nodes. Instead of having to hand-select a set of clip boundaries that are conducive to quality transitions, we are able to automatically provide the user with sets of clips whose starting and ending frames are "close". Moreover, given the desired transition locations we automatically modify the original database so cut transitions are possible. Specifically, at every transition the clips join seamlessly and any constraints in the motion (such as that a foot must planted on the ground) are enforced even if these constraints exist across transition boundaries.

In computer games and other virtual environments, move trees have demonstrated the utility of synthesizing motion based upon a handcrafted graph. The main limitation of this technique is in the difficulty of the authoring process: the necessary manpower limits the complexity of the graphs and the range of applications that can afford to build them. Our framework provides an alternative to manual authoring that alleviates this problem.

3 Alternate Approaches

The computer animation literature provides a number of ways of generating motion for synthetic characters. Since virtual environments require continuous streams of motion, some approaches are clearly inappropriate. Two obvious examples are keyframing and motion capture, which only create individual, static clips. Similarly, while motion capture editing [4, 11, 3, 23, 19, 5] and multi-target motion interpolation [22, 20] allow one to adapt a motion to new circumstances, these methods are still only capable of producing individual clips.

Procedural approaches have the advantage, in principle, of being able to generate flexible motions of arbitrary length. Perhaps the largest class of such approaches is physically-based motion synthesis. While physically-based methods have been successful for many natural phenomena, they have failed to scale to the complexities of character motions, with the exception of a few particular actions such as running [6] and jumping [13]. More ad hoc procedural methods have succeeded at a larger range of character motions [17, 18], but they require each new motion to be generated by hand and often do not produce realistic results.

Some recent approaches to motion synthesis involve constructing mathematical models based on a set of motion capture data. In particular, researchers have used hidden markov models [2] and switched linear dynamic systems [12] to create new motion. Such methods provide a straightforward way of generating arbitrarily-long streams of motion, but as yet it is unclear how they can be adapted to provide the high-level control required for virtual environments.

A number of graph-based approaches to motion synthesis have recently been developed that fully automate the graph construction process [1, 8, 10]. These methods allow graphs to be constructed quite quickly at the expense of providing severely limited control over the graph structure; indeed the generated graphs were *unstructured*.

In contrast to the explicitly designer-structured graphs of the previous section, unstructured motion graphs have no pre-determined connections between movements, and can make no guarantees about how quickly one motion can be reached from another. The path between two motions might be complex. Therefore, methods for synthesizing motions from unstructured graphs rely on search. By looking ahead, the search algorithms make choices that not only meet current needs, but have paths to future goals.

Unstructured graphs are inappropriate for interactive applications for several reasons. Interactive systems preclude lookahead and therefore search algorithms. Another problem is that it is difficult to know what motions are possible in an unstructured graph, since the connectivity is complex. For example, if a designer knows that a certain set of transitions will be required for a character's actions, there is no way to ensure that they are contained in the graph. Third, the control approaches currently used in interactive applications rely on known structure. For these reasons, we believe that interactive applications demand designer control over the graph structure.

Recently the graph-based approach has been extended to manually-constructed graphs in which the fundamental unit is not a static clip of motion, but rather a set of carefully-chosen clips that can be interpolated [16]. Parameterizing these interpolations appropriately can given one a finer degree of control over a character; for example, in the work cited one could specify locomotion in a continuum of directions and speeds, rather than from a discrete set of choices. At present it is unclear how readily this approach generalizes to larger, more expressive sets of motions.

4 Constructing Graphs

We assume the user has a database of motion capture data in a standard skeletal format. The number of motions in the database is irrelevant; it might contain many short clips or a single long clip. Each frame of motion is represented by a vector of parameters $(\mathbf{p}, \mathbf{q_1}, \ldots, \mathbf{q_n}, \mathbf{o_1}, \ldots, \mathbf{o_n})$, where $\mathbf{p}$ is a three-vector specifying the position of the root joint in world coordinates, $\mathbf{q_i}$ is a quaternion specifying the orientation of the i^{th} joint in its parent's coordinate system, and $\mathbf{o_i}$ is a three-vector indicating the offset of the i^{th} joint in its parent's coordinate system[1]. We assume that there is some linear indexing of the corpus, so a particular frame's vector is denoted by $\mathbf{F_i}$ for frame i of the corpus.

We also assume the motions are annotated with relevant constraints on end-effector positions. In this paper we limit our attention to footplant constraints, which specify that either the heel or ball of a particular foot must be planted over some set of frames (hence a total of four possible constraints may exist on a given frame). These types of constraints are by far the most common in motion capture data, and so this restriction is minor.

In our framework each edge in a graph is a clip of motion and each node is defined by a group of frames at which transitions are to occur. This group of frames is called a *match set* and each element of the match set is a *match frame*. If the original database has n motions, then the corresponding graph has a trivial structure with $2n$ nodes and n edges; refer to Figure 3. Each match set naturally partitions the database into shorter clips, which in turn correspond to edges in the graph that attach at a common node.

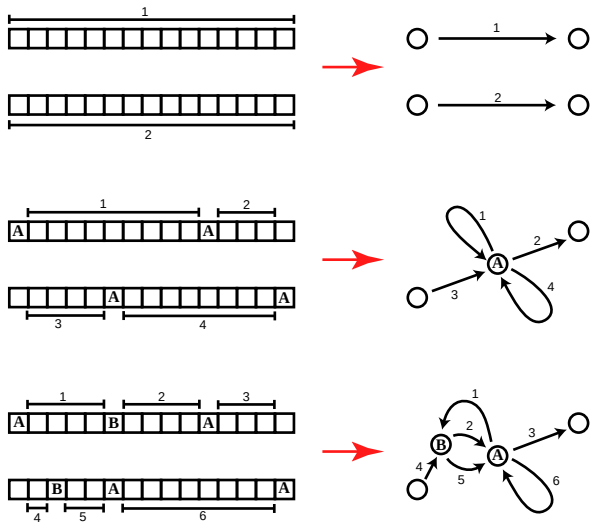

Figure 3: The top diagram schematically represents an initial database with two motions; on the left it is represented as two groups of frames and on the right is the corresponding graph. The middle diagram shows the result of making a match set out of four frames. This breaks the database into smaller clips and adds a new node to the graph. The bottom diagram demonstrates the addition of a second match set.

In our system graphs are built one node at a time by choosing match sets. If desired, an author can simply select the match frames manually. The author may also specify a particular frame and have the system automatically build a match set out of a group of similar frames. Finally, the author can have the system create a match set out of the largest collection of similar frames in the database.

Once the graph designer has finished creating match sets, our system automatically adjusts the motions so the corresponding transitions can be executed with simple cuts. This requires choosing a *common pose* for the match set, so that each match frame can be replaced by a rigid transformation of the common pose, and then transforming the surrounding frames such that this replacement is seamless. Any motion leading into the pose can then be followed by any of the motions exiting it, creating a multi-way transition.

The remainder of this section details our method. We first explain our process for helping a graph designer build match sets, then we describe our method for adjusting the original motions to generate seamless cut transitions, and finally we discuss the details of actually generating motion with the final graph.

4.1 Choosing Match Frames

Our system helps an author create match sets (and therefore nodes in the graph) by finding collections of frames that are similar to one another. This is accomplished through a scalar function $D(\mathbf{F_i}, \mathbf{F_j})$ that defines the distance between two frames $\mathbf{F_i}$ and $\mathbf{F_j}$. We use the same distance function as in [8], which has the advantage of automatically choosing a common coordinate system for $\mathbf{F_i}$ and $\mathbf{F_j}$. That is, since a motion is fundamentally unchanged by a rotation about the vertical axis and a translation along the floor plane, $\mathbf{F_j}$ needs to be "aligned" with $\mathbf{F_i}$ before the distance can be computed.

[1] Most motion capture processing systems assume perfectly rigid skeletons, in which case $\mathbf{o_i}$ is not explicitly represented. We use this more general skeleton representation since we employ the constraint solver described in [9], which adds small length changes to bones.

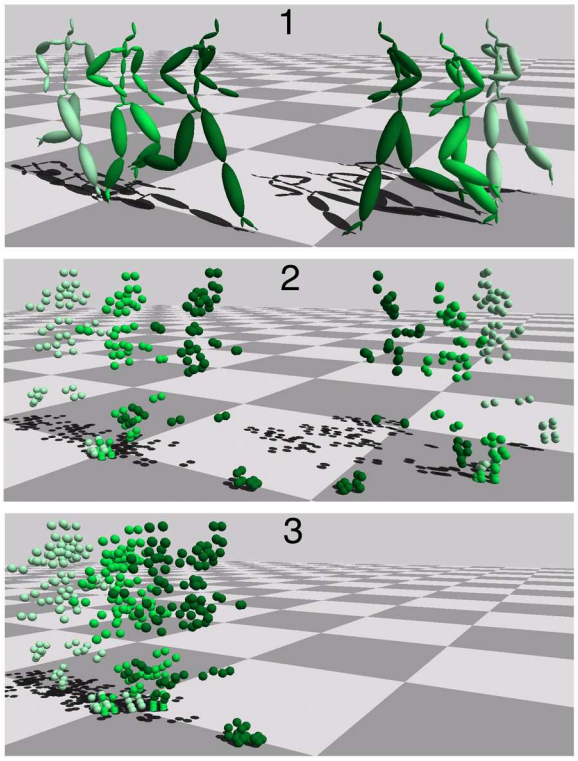

Figure 4: The distance between two frames $\mathbf{F_i}$ and $\mathbf{F_j}$ is calculated as follows. (1) Small neighborhoods of frames are extracted about $\mathbf{F_i}$ and $\mathbf{F_j}$. (2) These sets of frames are converted into two point clouds. (3) The optimal sum of squared distances between corresponding points is computed given that each point cloud can be rotated about the gravity axis and translated in the floor plane.

The distance calculation, motivated in [8], is shown in Figure 4. It works on clouds of points to avoid scaling issues in angle computations. First, small neighborhoods of frames are extracted around both $\mathbf{F_i}$ and $\mathbf{F_j}$. Two point clouds are then formed by attaching markers to the skeletons. Finally, the optimal weighted sum of squared distances is computed given that rigid 2D transformations may be applied to each point cloud. That is, we calculate

$$D(\mathbf{F_i}, \mathbf{F_j}) = \min_{\theta, x_0, z_0} \sum_k w_k \|\mathbf{p_{i,k}} - \mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}} \mathbf{p_{j,k}}\|^2, \quad (1)$$

where $\mathbf{p_{i,k}}$ is the k^{th} point in the cloud generated from frame i and $\mathbf{T}_{\theta, \mathbf{x_0}, \mathbf{z_0}}$ is a linear transformation consisting of a rotation of θ degrees about the y (vertical) axis followed by a translation of (x_0, z_0). The weights w_i sum to 1 and are chosen to give the most importance to $\mathbf{F_i}$ and $\mathbf{F_j}$ and less importance to frames toward the edges of the neighborhoods.

This optimization has the following closed-form solution:

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

$$x_0 = (\overline{x} - \overline{x'}\cos(\theta) - \overline{z'}\sin\theta) \quad (3)$$

$$z_0 = (\overline{z} + \overline{x'}\sin(\theta) - \overline{z'}\cos\theta), \quad (4)$$

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

For every pair of frames in the database there are two possible transitions, one that connects frames preceding $\mathbf{F_i}$ to frames following $\mathbf{F_j}$ and one that connects frames preceding $\mathbf{F_j}$ to frames following $\mathbf{F_i}$. D allows one to assign to each of these transitions a quality estimate and a coordinate transformation that aligns the ending motion with the starting motion. To speed interaction with our system, the distances and aligning coordinate transformations are precomputed for every pair of frames in the database.

Given a particular frame $\mathbf{F}$ and a user-defined threshold, we find a match set $S = \{\mathbf{F_1}, \ldots, \mathbf{F_n}\}$ as follows. For each motion in the database, we can form a 1D function by considering the distances between $\mathbf{F}$ and every frame of this motion. The local minima of these functions correspond to locally optimal transition points. We form a set S' of the frames corresponding to local minima whose values are below the threshold. These frames satisfy the similarity requirement for being match frames, but there is one more condition that must be met. Each match frame is associated with a displacement map that smoothly introduces the corresponding transitions into the motion database. As will be discussed more fully in Section 4.2, to create these displacement maps we require match frames to be at least w_{min} frames apart. So, in order of lowest distance to $\mathbf{F}$, we add to S the frames from S' that are at least w_{min} frames from every existing match frame.

If the graph designer wants $\mathbf{F}$ to serve as a hub node in the graph, then S determines the transitions that connect to this hub. By interactively choosing different thresholds the designer can determine an appropriate tradeoff between the number of edges attached to the hub and the quality of the resulting motions. The designer may also want to create a node based on the largest group of similar frames in the database. This can be found simply by forming S for every frame in the database and returning the one with the most elements. Ties are broken based on the lowest average distance between frames in S and the frame used to generate S.

4.2 Creating Transitions

Once the graph designer has finished creating match sets $S_1, \ldots, S_n$, our system adjusts the original database so the motions join seamlessly at all transitions points. Since transitions always occur between frames of a match set, it is sufficient to adjust the original motions such that the match frames are all identical, i.e., the values and velocities of each skeletal parameter are the same. If there are no constraints on the motions, this is accomplished solely through adaptation of displacement mapping techniques [23, 3]. If constraints *are* present, then matters are more complicated. Applying displacement maps will violate constraints, and if we subsequently use existing methods [4, 11, 9] to enforce them, the motions may change such that the match frames are no longer identical. We consider both of these cases, first treating transition generation in the absence of constraints and then when constraints exist.

4.2.1 Transitions Without Constraints

If constraints aren't present, then for each match set S_i our system creates an "average" frame $\mathbf{F_{S_i}}$ with a skeletal pose that is representative of the poses in the match frames. This pose is called the *common pose*. Our system then applies displacement maps that transform each match frame to have the common pose.

Figure 5 depicts our algorithm. In the original database the match frames are scattered about in a global reference frame. If we are to compute an average pose, the match frames must first be aligned. As discussed in Section 4.1, every pair of match frames $\mathbf{F_j}, \mathbf{F_k} \in \mathbf{S_i}$ has a rigid 2D transformation that aligns them for the

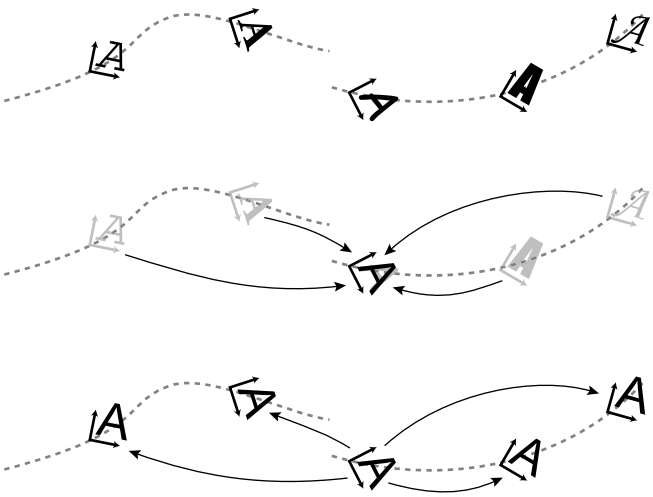

Figure 5: (top) In the original motions, match frames are scattered in the global coordinate system. (middle) We choose a particular match frame, align the others to it, and compute an average skeletal posture to serve as the common pose. (bottom) Using a set of displacement maps, each match frame is altered to have this common pose.

purposes of executing a transition. Let $\mathbf{T_{jk}}$ be the transformation that when applied to $\mathbf{F_j}$ aligns it with $\mathbf{F_k}$. Since each transformation was computed independently via equations $(2)-(4)$, in general they will be inconsistent in the sense that $\mathbf{T_{jk}T_{kl}} \neq \mathbf{T_{jl}}$. We could attempt to find a set of coordinate transforms that *are* consistent by, for example, adjusting Equation (1) to optimize simultaneously over several coordinate transforms. However, for more than two point clouds there is no simple closed form solution and an expensive nonlinear optimization would be necessary. On the other hand, we observe that if the match frames in S_i are sufficiently similar then the coordinate transformations will be *approximately* consistent. Hence we may simply select one particular match frame to define the coordinate transforms for every other match frame. Say we select $\mathbf{F_{j_{base}}}$. Then we redefine the $\mathbf{T_{pq}}$ to be

$$\mathbf{T_{pq}} := \mathbf{T'_{pq}} = \mathbf{T_{j_{base}p}T^{-1}_{j_{base}q}}. \quad (5)$$

These new coordinate transforms guarantee that $\mathbf{T_{pq}T_{qr}} = \mathbf{T_{pr}}$. We can now align the k^{th} match frame in S_i with $\mathbf{F_{j_{base}}}$ by applying the transformation $\mathbf{T_{kj_{base}}}$.

In practice $\mathbf{F_{j_{base}}}$ is not chosen arbitrarily. Rather, our system attempts to choose the match frame that is closest to being in the "center" of the other frames. This corresponds to choosing the match frame with the smallest sum of distances to the other match frames.

Once we have chosen $\mathbf{F_{j_{base}}}$, our system computes $\mathbf{F_{S_i}}$ by aligning the match frames into the coordinate system of $\mathbf{F_{j_{base}}}$. The root position, joint offsets, and joint orientations of $\mathbf{F_{S_i}}$ are the average of the corresponding quantities in the match frames. The average joint orientation is computed as in [16].

We can now form displacement maps that replace each $\mathbf{F_k} \in S_i$ with $\mathbf{T^{-1}_{kj_{base}}F_{S_i}}$ (Figure 5). Since each match frame is identical, motion is guaranteed to be continuous at transitions. This use of displacement maps is similar to previous work [10, 1] which used displacement maps to guarantee C_0 continuity at transitions. However, for motions with very different velocity characteristics C_0 continuity may be insufficient (Figure 6). For this reason we extend previous efforts by building displacement maps that preserve C_1 continuity. For each skeletal parameter we compute the average velocity over all match frames. We then construct displacement maps such that motions pass through the common pose with these parameter velocities. Since the motions are represented as discretely sampled signals, care must be taken in computing derivatives. Because continuity is most important at a scale greater than a single frame, we estimate derivatives by calculating finite differences at each point in a small window and filtering the results.

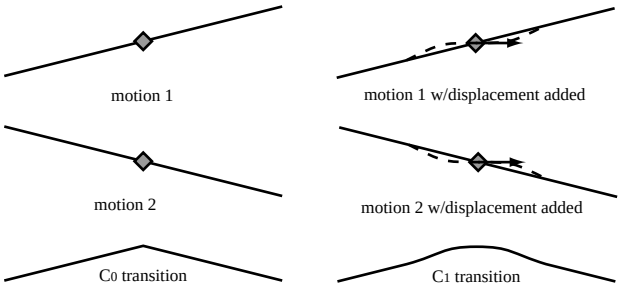

Figure 6: C_0 transitions can still cause discontinuities if motions have very different velocities. For this reason we use C_1 smooth displacement maps.

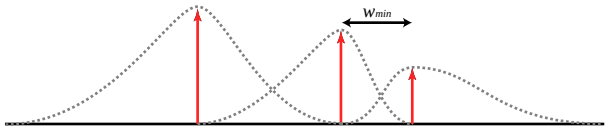

Figure 7: At each match frame a displacement map is used to smoothly alter the motion so as to facilitate transitions; this figure depicts a motion with three match frames and the corresponding displacement maps. On each side the displacement map extends up to either the next match frame or the motion boundary, whichever comes first. Displacement maps are required to extend at least w_{min} frames on either side, so match frames must be at least w_{min} frames apart.

Each side of a displacement map extends to either the nearest match frame or a boundary of the motion, whichever comes first (Figure 7). To ensure that changes do not occur too rapidly, we require match frames to be spaced at least w_{min} frames apart. If there are n joints in the skeleton, then the displacement map consists of $2n+1$ splines: one for the root position, n for the joint offsets, and n for the joint orientations. The ends of each spline have zero value and derivative and the center is chosen to map the relevant parameter to the target value and derivative. We construct these splines out of two Hermite cubic segments; for orientations we construct quaternion splines using the method in [7].

4.2.2 Transitions With Constraints

If displacement maps are applied to the original motions, then any constraints on those motions are likely to be violated. We now consider how to create smooth multi-way transitions while simultaneously preserving constraints. We focus on the most common kinds of constraints, which are footplant constraints. A footplant constraint specifies that either the left heel, right heel, left ball, or right ball must be fixed on the ground. To enforce a footplant constraint, two things must be done: 1) positions must be chosen for each constrained joint and then 2) the motion must be smoothly adjusted so the constrained joints are in these positions. We use the method of [9] to enforce footplant constraints. This algorithm has the important property that one can ensure that a particular frame is not altered by constraining the root, heels, and balls of the feet to remain in their current positions. We refer to this as *locking* the frame.

As in the previous section, our basic strategy is to construct a representative frame $\mathbf{F_{S_i}}$ for each match set S_i and use displacement maps to make the match frames identical to $\mathbf{F_{S_i}}$. We define a con-

straint to exist in $\mathbf{F_{S_i}}$ if and only if it exists on a majority of the match frames[2], which means that individual match frames may end up gaining and/or losing constraints.

Since constraints must be enforced in the final motion, $\mathbf{F_{S_i}}$ must satisfy all of its constraints, i.e., the constrained joints must be on the ground. Assume this is true. As in Section 4.2.1 we can apply displacement maps such that for all match sets each match frame is identical to the appropriate common pose. If we then lock each match frame and apply the constraint enforcement algorithm, our database of motions will have the desired properties: all constraints will be enforced and each match set will contain identical frames.

While we could choose the common poses using the same algorithm as in Section 4.2.1, this method fails to take into account constraint information. This is problematic since by locking each match frame, we are forcing the motions returned by the constraint solver to pass through the common poses. For example, say the left heel is *unconstrained* on some match frame that is only a few frames away from a region where the left heel must be planted. If the left heel happens to be far from the ground in the common pose, then the constraint solver will be forced to generate a motion where the foot leaves the ground with unnatural speed.

Intuitively, we would like to select the $\mathbf{F_{S_i}}$ such that when we replace each match frame with the appropriate common pose and lock it, the locking has as little effect as possible. That is, if we imagine *not* doing this locking and enforcing constraints, we would like the match frames to nonetheless remain unchanged. In light of this we use the following two-step iterative procedure for determining a particular $\mathbf{F_{S_i}}$. We start out by creating a "working set" that initially contains copies of the match frames as they appear in the original motions. Each iteration estimates the common pose by averaging the working set, and creates a variant of each motion that passes through this common pose using the same displacement map technique described in Section 4.2.1. This possibly violates constraints. Next, we apply the constraint enforcement algorithm to the modified motions, possibly adjusting the matched frames. After this the matched frames, which may no longer be identical, are copied back into the working set. Each iteration begins with the motion from the original database and evolves the common pose. At the end of the final iteration we set the common pose $\mathbf{F_{S_i}}$ to be the average of the poses in the working set. In our experiments only a small number of iterations (3-5) were necessary.

The $\mathbf{F_{S_i}}$ generated through the above algorithm will not necessarily satisfy their constraints. We can correct this by choosing positions for the constraints and applying inverse kinematics. However, constraint positions in general can not be found independently for each $\mathbf{F_{S_i}}$. In particular, if two common poses share a constraint and border a clip that has this constraint on each frame, then the constraint positions for the common poses must be chosen such that in this clip they are in the same location. This issue arises in many common situations, such as if the character stands in place. We describe a solution to this problem in the Appendix.

4.3 Generating Motion at Run Time

Each transition involves two pieces of information: the clip we're transitioning to and the coordinate transformation that aligns it with the current clip. At run time these coordinate transformations are the only information that needs to be kept track of. That is, to play the current clip, we simply adjust the root of every (precomputed) frame by the current coordinate transformation, and whenever we make a transition we update this transformation.

[2]We have found that requiring all match frames to have the same constraint state, as suggested by [10], forces us to exclude too many good potential matches.

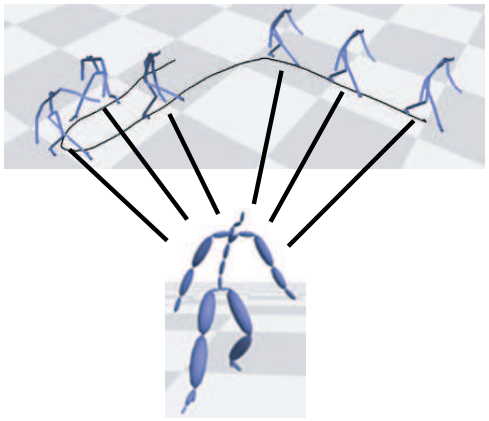

Figure 8: On top are the five frames of a match set generated automatically for a short sneaking motion. On the bottom is the corresponding common pose.

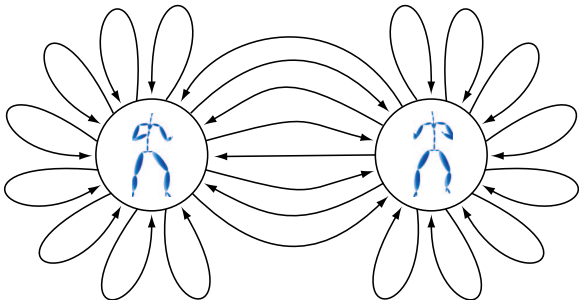

Figure 9: A schematic of the two-node martial arts graph generated with our system. Our algorithm for creating match sets automatically selected left and right "ready" stances as the hubs of the graph.

As discussed in previous graph-based approaches to motion synthesis [21, 10, 8], certain nodes of the graph may be dead ends in the sense that they are not part of any cycle. Once such a node is entered, there is a limit to how much further animation can be produced. This is unacceptable for virtual environments, since characters must be animated for arbitrarily long amounts of time. Our system notifies the graph designer of possible dead ends by finding the nodes that are not part of the largest strongly connected component [10, 8]. The designer may then decide to either add new transitions or remove these nodes.

5 Results

We have implemented a system based on the methods in Section 4 and applied it to a number of motion datasets. Figure 10 is a screenshot of the window seen by the graph designer. In the upper right is a visualization of the distance function; pixel (i, j) represents $D(\mathbf{F_i}, \mathbf{F_j})$, with darker pixels corresponding to lower distances. On the far upper right is a slice of the $2D$ distance function showing the distances between a frame selected by the user and the other frames in the database. The bottom of the window shows a schematic of the graph given the current match sets. The horizontal black lines represent original motions and the vertical lines indicate match frames. All frames of the same match set are the same color. Clicking on a segment in this schematic causes the corresponding clip to be displayed in the upper left window.

We created a set of graphs by having the system automatically create nodes based on the largest sets of match frames. To test the system, we started with a single motion of someone sneaking for thirteen seconds and built a graph with a single node and 7 clips; see Figure 8. We then moved on to larger data sets, constructing graphs and driving their input with a video game controller. We first built a two-node graph out of a dataset containing 900 frames (30 seconds) of martial arts motions (Figure 9). The common poses generated automatically by the system corresponded to two "ready" stances, one with the left foot forward and one with the right foot forward. We then mapped the clips to buttons on a gamepad, allowing a user to interactively direct the character to punch, kick, dodge, shuffle-step, and switch stances. We next built a one-node graph out of 3000 frames (100 seconds) of walking data. This graph allowed a user to guide a character by specifying the curvature of its path, where the options ranged from a gentle arc to a sharp about-face. Finally, we combined these two datasets into a larger graph that allowed all of the previous operations plus the ability to switch between walking and fighting modes.

The semi-automatic nature of our system makes it possible to produce graphs quite quickly. The total amount of time necessary to build the martial arts graph — from raw data to being able to interactively control a character — was about 12 minutes, and the walking graph took about 20 minutes. Most of this time was spent deciding how to map the clips to the gamepad.

6 Discussion

In this paper we have described a framework for synthesizing character motions in virtual environments by assembling clips built from a corpus of motion capture data. We meet the visual quality demands of virtual environments by preserving the fidelity of the original motions. We meet performance demands by performing all processing of the motions at authoring time, so at run time clips can simply be concatenated in appropriate orders. Finally, we meet controllability and responsiveness demands by allowing the user to guide the graph building process to ensure that the graph has a usable structure. Specifically, we support and encourage the creation of hub nodes that allow many different actions to be reachable from a common point.

Our approach automates tedious portions of the graph construction process and makes it possible to use data more opportunistically. This can allow graphs to be created from a wide range of data that was not specifically captured for graph construction, and it can also enable designers to build graphs of a scope that would otherwise be too expensive to produce.

The authoring tool described in this paper required several new techniques to be developed:

1. We automatically identify potential hub nodes, allowing a graph designer to avoid tedious parts of the construction process.
2. We introduce C_1 displacement maps as a means of creating higher quality cut transitions.
3. We provide a method for satisfying constraints as a preprocess, allowing the complexity of constraint satisfaction to be avoided at run time.

The run-time execution of our approach is intentionally similar to current (and successful) methods that use manually constructed graphs. We believe this will make it easier to apply our methods in practical virtual environments. Moreover, by reducing the effort required to construct graphs suitable for run-time synthesis, we hope to make run-time animation accessible to a broader array of applications.

Acknowledgements

We would like to thank everyone in the UW graphics group for their help with this project. Motion data was generously provided by Demian Gordon and House of Moves Studios. This research was supported in part by a Wisconsin University-Industry Relations Grant, NSF grants CCR-9984506 and CCR-0204372, and equipment donations from Intel. Lucas Kovar is supported by an Intel Foundation Fellowship.

References

[1] Okan Arikan and D.A. Forsythe. Interactive motion generation from examples. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, July 2002.

[2] Matthew Brand and Aaron Hertzmann. Style machines. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 183–192, July 2000.

[3] Armin Bruderlin and Lance Williams. Motion signal processing. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 97–104, August 1995.

[4] Michael Gleicher. Retargeting motion to new characters. In *Proceedings 0f ACM SIGGRAPH 98*, Annual Conference Series, pages 33–42, July 1998.

[5] Michael Gleicher. Motion path editing. In *Proceedings 2001 ACM Symposium on Interactive 3D Graphics*, March 2001.

[6] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 71–78, August 1995.

[7] Myoung-Jun Kim, Myoung-Soo Kim, and Sung Yong Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *Proceedings of ACM SIGGRAPH 1996*, Annual Conference Series, pages 369–376, August 1996.

[8] Lucas Kovar, Michael Gleicher, and Fred Pighin. Motion graphs. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, July 2002.

[9] Lucas Kovar, John Schreiner, and Michael Gleicher. Footskate cleanup for motion capture editing. In *ACM Symposium on Computer Animation 2002*, July 2002.

[10] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, July 2002.

[11] Jehee Lee and Sung Yong Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of ACM SIGGRAPH 99*, Annual Conference Series, pages 39–48, August 1999.

[12] Yan Li, Tianshu Wang, and Heung-Yeung Shum. Motion texture: A two-level statistical model for character motion synthesis. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, July 2002.

[13] C. Karen Liu and Zoran Popović. Synthesis of complex dynamic character motion from simple animations. In *Proceedings of ACM SIGGRAPH 2002*, Annual Conference Series, July 2002.

[14] Alberto Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Academic Press, San Diego, CA, 2000.

[15] Mark Mizuguchi, John Buchanan, and Tom Calvert. Data driven motion transitions for interactive games. In *Eurographics 2001 Short Presentations*, September 2001.

[16] Sang Il Park, Hyun Joon Shin, and Sung Yong Shin. On-line locomotion generation based on motion blending. In *ACM Symposium on Computer Animation 2002*, July 2002.

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

[18] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of ACM SIGGRAPH 96*, pages 205–216, August 1996.

[19] Zoran Popović and Andrew Witkin. Physically based motion transformation. In *Proceedings of ACM SIGGRAPH 99*, Annual Conference Series, pages 11–20, August 1999.

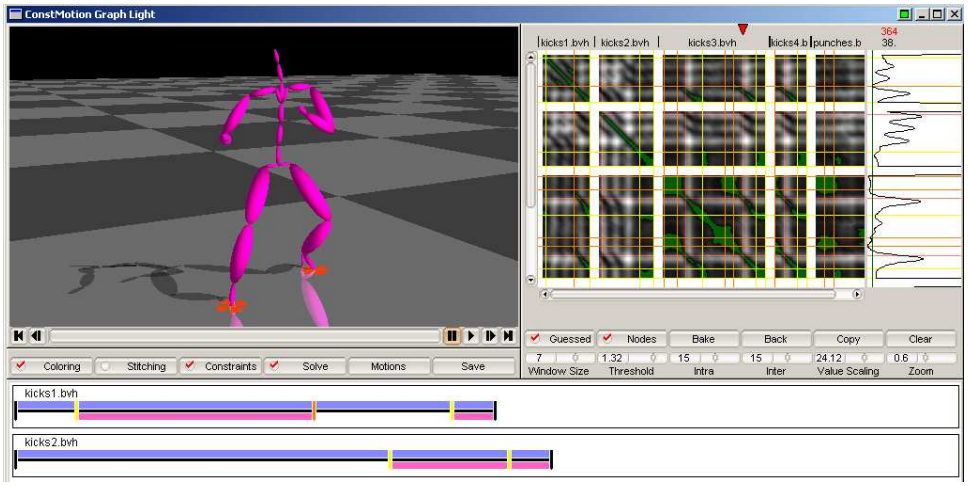

Figure 10: A screenshot of our graph-building interface.

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

[21] Arno Schodl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video textures. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 489–498, August 2000.

[22] D. Wiley and J. Hahn. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Application*, 17(6):39–45, 1997.

[23] Andrew Witkin and Zoran Popović. Motion warping. In *Proceedings of ACM SIGGRAPH 95*, Annual Conference Series, pages 105–108, August 1995.

Appendix

The $\mathbf{F_{S_i}}$ generated through the iterative algorithm of Section 4.2.2 will in general not satisfy their constraints unless these constraints are explicitly enforced. This amounts to identifying positions for each constraint and then performing inverse kinematics to ensure the relevant joints reach these positions. This process is complicated by the fact that the choice of constraint positions can not be made independently for each $\mathbf{F_{S_i}}$. Consider the case where $\mathbf{F} \in S_i$ and $\mathbf{F}' \in S_j$ share some constraints and border a clip that also has these constraints on every frame. For the resulting motion to be continuous, we require that $\mathbf{F_{S_i}}$ and $\mathbf{F_{S_j}}$ (when transformed to be aligned with $\mathbf{F}$ and $\mathbf{F}'$) place the constrained joints in the same location; $\mathbf{F_{S_i}}$ and $\mathbf{F_{S_j}}$ are *linked* on these constraints. Since constraints can exist anywhere in the original motions, common poses can be linked arbitrarily.

Linked constraints are not an artifact of having a bizarre set of motions. On the contrary, they occur in quite ordinary datasets. Consider, for example, a set of motions of someone waiting around impatiently. The character might shuffle its feet, tap its toes, and make subtle shifts in posture to redistribute its weight. In the likely event that constraints exist on every frame of the dataset, *every* common pose will have linked constraints with every other common pose.

To ensure continuous motion, linked constraint positions of $\mathbf{F_{S_i}}$ and $\mathbf{F_{S_j}}$ need only be identical up to a $2D$ rigid transform. Recall that when making a transition we first align the starting and ending motions so the match frames are in the same position and orientation. Section 4.2.1 explained how to determine these coordinate transformations based on the values we computed for D (Section 4.1). However, we are free to pick different transformations, and in particular we can select ones specifically to align constraint positions.

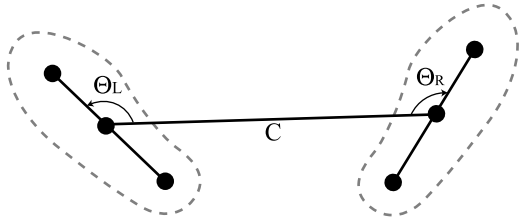

Figure 11: Up to a rigid $2D$ transformation, the configuration of two feet that are flat on the floor is uniquely defined by the distance between the centers of the feet and the orientation of each foot relative to the line connecting the centers.

So: the problem is to ensure that any linked constraint positions are identical up to a $2D$ rigid transformation, or *rigidly similar*. We can determine how common poses are linked simply by looking at every clip and determining whether the bordering match frames share constraints that exist throughout the clip. If the only linked constraints are on joints of the same foot, then since the foot is rigid the constraint positions are automatically rigidly similar. If linked constraints exist for all four joints on the feet, then let $\mathbf{C_1}$ and $\mathbf{C_2}$ be the segments connecting the centers of the feet in, respectively, the starting common pose and the ending common pose (Figure 11). Also, let Θ_{L_1} and Θ_{R_1} be the orientations of the left and right foot in the starting common pose relative to $\mathbf{C_1}$, and let Θ_{L_2} and Θ_{R_2} be defined similarly. To ensure rigid similarity it is sufficient to require $\|\mathbf{C_1}\| = \|\mathbf{C_2}\|$, $\Theta_{L_1} = \Theta_{L_2}$, and $\Theta_{R_1} = \Theta_{R_2}$. If there are only two or three linked constraints and they exist on joints of different feet, then the situation can be reduced to the four-joint case by rotating any foot with only one linked joint about that joint such that it is flat on the floor.

We can divide common poses into equivalence classes via constraint linkage. Each common pose in an equivalence class has linked constraints on both feet with at least one other common pose in that same class. For each equivalence class, we find the average foot orientations and distance between the foot centers. Each common pose is then adjusted to have these average parameters.

We have now ensured that every set of linked constraint position are rigidly similar. However, the coordinate transformations that align clips (as computed in Section 4.2.1) may not align the constraints positions. This can be addressed by redefining these coordinate transformations such that the constraint positions are identical for the last frame of the starting clip and the first frame of the ending clip.