Chapter 3

Background

The work in this dissertation builds directly on existing work in the area of motion synthesis. This chapter describes in more detail some of the techniques and concepts unique to motion synthesis that are used extensively throughout the rest of the document. In particular, this chapter briefly describes how motions in this dissertation are represented, the metric used to measure the similarity between two pieces of motion, the method used to append two motions together, the process for aligning two motions in time, and the blending-based parametric synthesis method employed. It should be noted that none of the methods presented in this chapter are novel. All of the described techniques are well described elsewhere. They are reviewed here to provide explanation and motivation for the methods I build upon and to introduce important terms and notation that are used throughout the rest of this document. Please refer to the original papers on these techniques for more details.

3.1 Motion Representation

The raw data collected by a standard motion capture system consists of the 3D locations of a set of points that are rigidly attached to the actor's body. Because of the ease of animating a character using a set of hierarchically organized joints, called a *skeleton*, the raw data from a motion capture shoot usually goes through the process of being skeletonized (see [BRRP97, OBBH00, ZH03] for more information on how this process is accomplished).

While the work presented in Chapter 5 uses raw motion capture data directly, I primarily represent the human body as a rigid-body skeleton. Each joint of the skeleton has exactly one parent

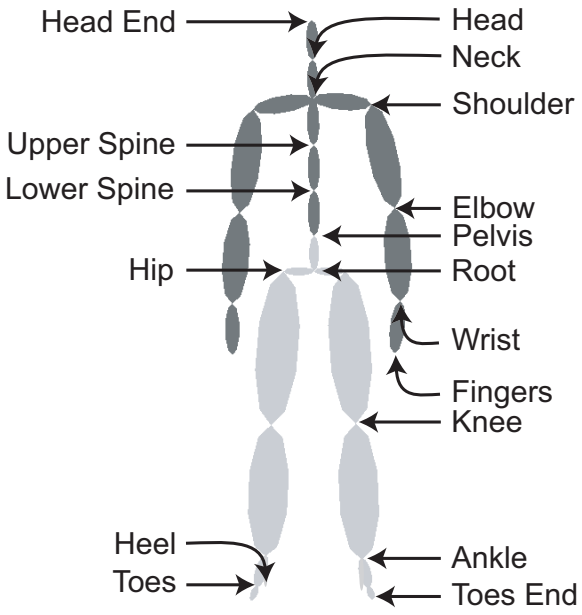


Figure 3.1 Standard Skeleton Hierarchy

joint, with the exception of the *root* joint that has no parent. A skeleton can be defined by its joint hierarchy and initial pose, or the offset of each joint from its parent in local coordinates. Figure 3.1 shows the standard skeleton hierarchy used in all of the experiments presented in this dissertation. Note that the articulated skeleton can only control the body motion of the character; the skeleton does not include joints that represent fingers, toes, or facial features. In general, this dissertation is only concerned with a character's overall body motion, with the notable exception of eye movement as discussed in Chapter 5.

Given a character's skeleton, a motion is defined as a continuous function over time:

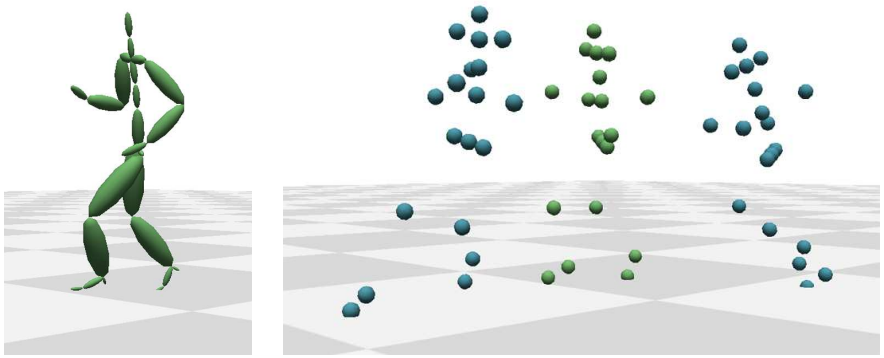
$$\mathbf{M}(t) = \{\mathbf{p}(t), \mathbf{q}_1(t), \ldots, \mathbf{q}_k(t)\} \quad [3.1]$$

where $\mathbf{p}$ is the position of the root with respect to the origin and $\mathbf{q}_j$ is the relative orientation of the j^{th} joint with respect to its parent. In the case of the root, which does not have a parent joint, the orientation is denoted with respect to the global coordinate system at the origin.

In practice, the continuous function $\mathbf{M}(t)$ is represented as a set of samples, or *frames*, taken at regular time increments, $\mathbf{M}(t_1), \ldots, \mathbf{M}(t_n)$. Values of $\mathbf{M}$ in between frames are computed by using linear interpolation of the root position and spherical linear interpolation on the joint orientations represented as unit quaternions [Sho85]. While there are a number of other ways to represent rotations, unit quaternions are particularly well-suited for interpolation (see [Gra98]), making them the rotation representation of choice for many motion researchers.

3.2 Motion Similarity

Much of my work depends on computing the similarity between two frames of motion, or, conversely, to compute the difference between two frames of motion. One could compute the difference between two frames of motion, $\mathbf{M}(t)$ and $\mathbf{M}'(t')$, by directly comparing the sampled motion vectors, $\{\mathbf{p}(t), \mathbf{q}_1(t), \ldots, \mathbf{q}_k(t)\}$ and $\{\mathbf{p}'(t'), \mathbf{q}'_1(t'), \ldots, \mathbf{q}'_k(t')\}$, a method employed by others studying human motion [LZWP03] as well as by researchers in other domains with high-dimensional spaces [BBK01]. There are two problems with this type of direct comparison:



a **b**

Figure 3.2 Point Cloud Representation of a Frame of Motion. (a) shows a frame of running motion in skeletal form. (b) shows the same frame of motion in point cloud form. Notice how the point cloud representation contains information about where each of the joints of the skeleton are located in 3D space on the frame in question as well as the frames surrounding it.

1. It fails to account for differences introduced solely due to a motion's global orientation in the environment.

2. It does not take into account differences between the dynamics of the two motions.

Thus, I compute the difference between two frames of motion, $\mathbf{D}(\mathbf{M}(t), \mathbf{M}'(t'))$, using a metric originally introduced by Kovar et al. [KGP02]. I chose to use this algorithm for two reasons, each of which addresses one of the two problems associated with direct motion vector comparison:

1. The metric is invariant to translations along the ground plane and rotations about the up-axis.

2. The metric can take into account joint velocities and accelerations using finite differences.

The metric works by first representing each frame of motion as a cloud of points, henceforth called a *point cloud*. The points of a point cloud correspond to the locations of relevant skeletal joints over a small window of time surrounding the frame. Figure 3.2 shows an example of a point cloud generated from a frame of running motion. Using the following closed-form solution from [KGP02], I can compute the rotation about the vertical axis, θ, and translation along the floor

plane, (x_0, z_0), that best aligns corresponding points, $\mathbf{p_i}$ and $\mathbf{p'_i}$, in the two point clouds.

$$\theta = \arctan \frac{\sum_i w_i(x_i z'_i - x'_i z_i) - \frac{1}{\sum_i w_i}(\overline{x}\overline{z'} - \overline{x'}\overline{z})}{\sum_i w_i(x_i x'_i + z_i z'_i) - \frac{1}{\sum_i w_i}(\overline{x}\overline{x'} + \overline{z}\overline{z'})} \quad [3.2]$$

$$x_0 = \frac{1}{\sum_i w_i}(\overline{x} - \overline{x'}\cos(\theta) - \overline{z'}\sin\theta) \quad [3.3]$$

$$z_0 = \frac{1}{\sum_i w_i}(\overline{z} + \overline{x'}\sin(\theta) - \overline{z'}\cos\theta) \quad [3.4]$$

where i is an index over the number of points in each cloud, w_i is a weighting term for point $\mathbf{p_i}$ in the point cloud, x_i and z_i are the x and z coordinates of point $\mathbf{p_i}$, and all barred terms correspond to the weighted sum of the barred variables over the index i.

Using this optimal alignment, the distance between the two frames is computed as:

$$\mathbf{D}(\mathbf{M}(t), \mathbf{M}'(t')) = \sum_i (\mathbf{p_i} - \mathbf{\Phi}_{\theta,\mathbf{x_0},\mathbf{z_0}}(\mathbf{p'_i}))^\mathbf{2} \quad [3.5]$$

where the function $\Phi_{\theta,x_0,z_0}(\mathbf{p})$ applies the optimal point cloud alignment transform, $\{\theta, x_0, z_0\}$, to the point $\mathbf{p}$.

Depending on the length of the window over which the point cloud is built, this distance metric can implicitly take into account relative joint positions, joint velocities, and joint accelerations when measuring similarity. Refer to [KGP02] for a more detailed description of this point cloud distance metric.

3.3 Transitioning Between Two Motions

It is often useful to append two motions together with a transition. Unfortunately, appending motions together by simply appending motion frame vectors is likely to cause discontinuities in the motion. Instead, as described in Section 2.2.1, it is common to append two motions together by using a linear blend transition between the two motions over a small window of frames centered at the transition point.

Appending a motion, $\mathbf{M}_2$, to another motion, $\mathbf{M}_1$, using a linear blend consists of three steps:

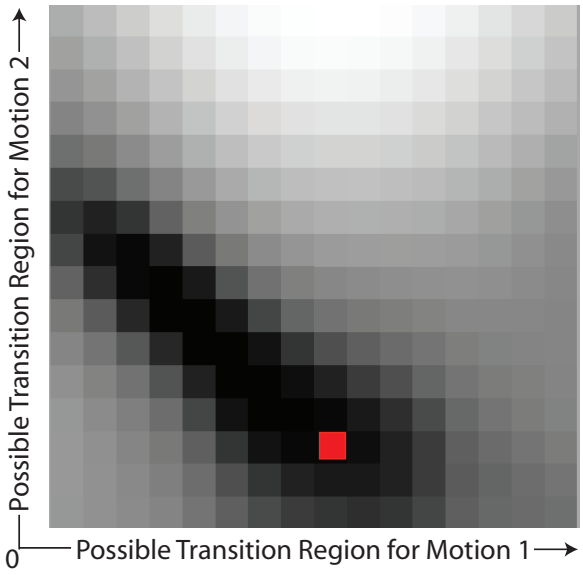


Figure 3.3 A distance grid. Darker regions denote greater similarity between frames. The light red dot marks the optimal transition point.

1. **Choosing a transition point between the two motions.** While a transition point can be chosen arbitrarily between the two motions, linear blend transitioning works best if the motions look as similar as possible. For this reason, my methods choose a transition point between the two motions by locating the point where the two motions look the most similar to each other over the possible transition region. The point cloud distance metric presented in Section 3.2 is used to rate the similarity of different frame pairs to determine the transition point. My method for finding the transition point starts by calculating the distance between every pair of frames in the possible transition regions, forming a grid. The pair of frames corresponding to the grid cell with the minimum distance value, (t_o^1, t_o^2), is called the optimal transition point. Figure 3.3 shows an example of this distance grid computation between two motions.

2. **Aligning the two motions at the transition point.** Because the global position and orientation differs between $\mathbf{M}_1$ and $\mathbf{M}_2$, it is necessary to align $\mathbf{M}_2(t_o^2)$ to $\mathbf{M}_1(t_o^1)$. This can be done by applying the optimal alignment transform associated with $\mathbf{D}(\mathbf{M}_1(t_o^1), \mathbf{M}_2(t_o^2))$, Φ_{θ,x_0,z_0}, to the motion $\mathbf{M}_2$.

3. **Synthesizing the appended motion through blending.** The final step necessary to append motion $\mathbf{M}_2$ to motion $\mathbf{M}_1$ is to blend the motions over time to produce the final appended motion, $\mathbf{M}_{1+2}$:

$$\mathbf{M}_{1+2}(t) = \left\{ \begin{array}{ll} \mathbf{M}_1(t) & \text{if } t < t_o^1 - \frac{w}{2} \\ \mathbf{M}_2(t_o^2 + t - t_o^1) & \text{if } t > t_o^1 + \frac{w}{2} \\ (1 - \alpha(t)) * \mathbf{M}_1(t) + \alpha(t) * \mathbf{M}_2(t_o^2 + t - t_o^1) & \text{otherwise} \end{array} \right\}$$

$$\alpha(t) = \frac{t - t_o^1}{w}$$

   This blending process is illustrated in Figure 3.4.

I chose to use this method for appending two motions together using a linear blend transition because it is a simple method. The limitation is that linear blend transitions only work reliably when the motions that are being appended are already close to one another in terms of the similarity

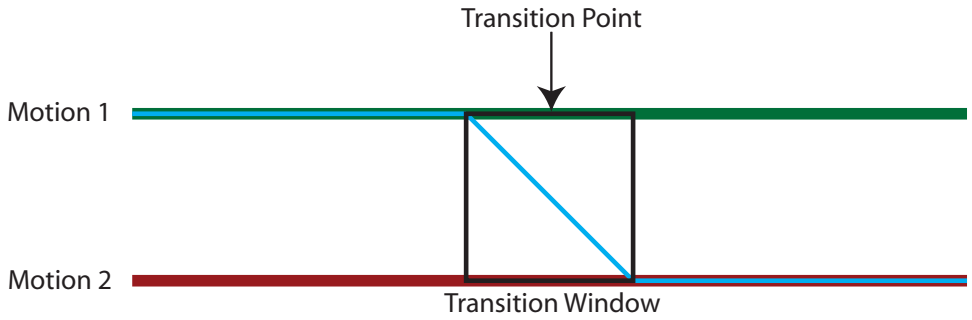


Figure 3.4 Appending two motions through linear blend transitioning. The green bar (top) and red bar (bottom) correspond to the original two motions. The thinner, light blue bar shows the weighting for the blend of the motions over time. Notice how the weights go from being fully on motion 1 to fully on motion 2 over the transition window centered at the transition point.

metric presented in Section 3.2. Rather than developing a more complex method for appending or transitioning between motions, I use this simple method but ensure that I only transition between motions with sufficient similarity.

3.4 Motion Time Alignment

Two motions, $\mathbf{M}_1$ and $\mathbf{M}_2$, may be logically similar in that they represent different motions of the same action. For instance, both motions might be instances of a person walking, following the same footstep pattern. Yet, these logically similar motions might have very different timing details. For instance, each step in $\mathbf{M}_1$ might take twice as long as each step in $\mathbf{M}_2$. For some applications, it is desirable to find a time alignment between logically similar motions such that frames in one motion can be mapped to time-corresponding frames in the other motion. This continuous, strictly increasing mapping is called a *time alignment curve*.

When computing a time alignment curve between two motions, $\mathbf{M}_1$ and $\mathbf{M}_2$, the goal is to find a mapping between frames of $\mathbf{M}_1$ and frames of $\mathbf{M}_2$ that minimizes the average distance between corresponding frames, where distance is computed using the algorithm in Section 3.2. First, the distance between every pair of frames is calculated, forming a grid as in Section 3.3. Then using the dynamic programming method of Kovar and Gleicher [KG03], a continuous, monotonic, and non-degenerate path is computed for every cell in the grid that connects to the lower-left corner, while minimizing the sum of its cells. This optimal path from the lower-left corner to the upper-right corner of the grid provides a discrete, monotonically increasing time alignment, as shown in Figure 3.5. To generate the final time alignment curve, a strictly increasing, endpoint-interpolating B-spline is fit to the optimal path. See [Kov04] for more information on how to increase the speed of this time alignment algorithm.

The dynamic timewarping method that I use in this dissertation to compute time alignment curves is based on the work of Kovar and Gleicher [KG03], but there are several other published methods that would work as well. In particular, the timewarping methods of Bruderlin and Williams [BW95], Dontcheva et al. [DYP03], and Hsu et al. [HPP05] also use dynamic programming to find a temporal alignment between motions. Hsu et al.'s method [DYP03] builds on

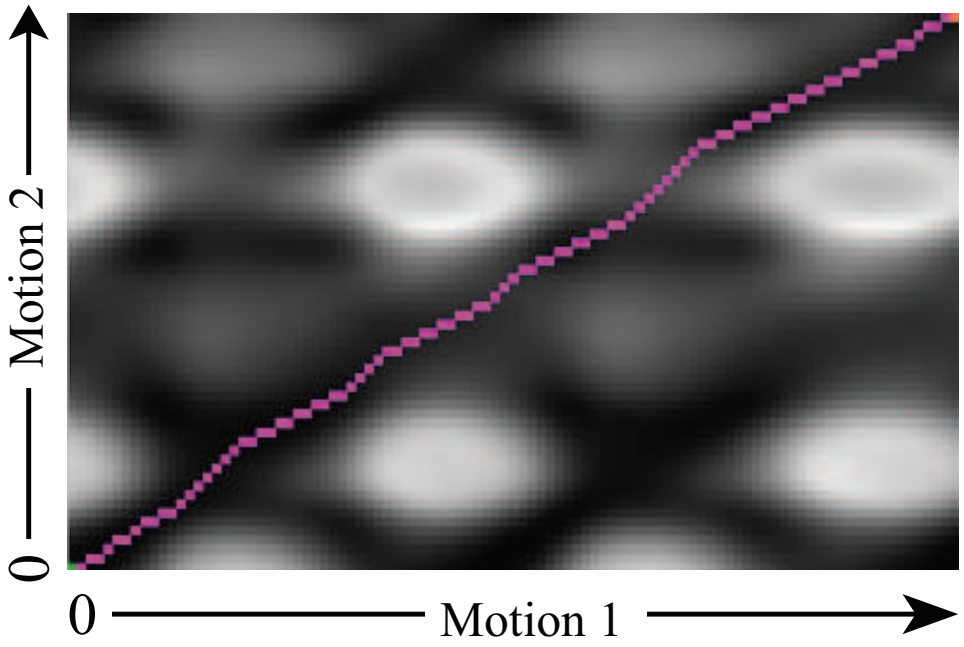


Figure 3.5 A grid depicting the difference between Motion 1 and Motion 2. Dark areas denote similarity between the corresponding frames. An optimal time alignment is shown by the path through the grid.

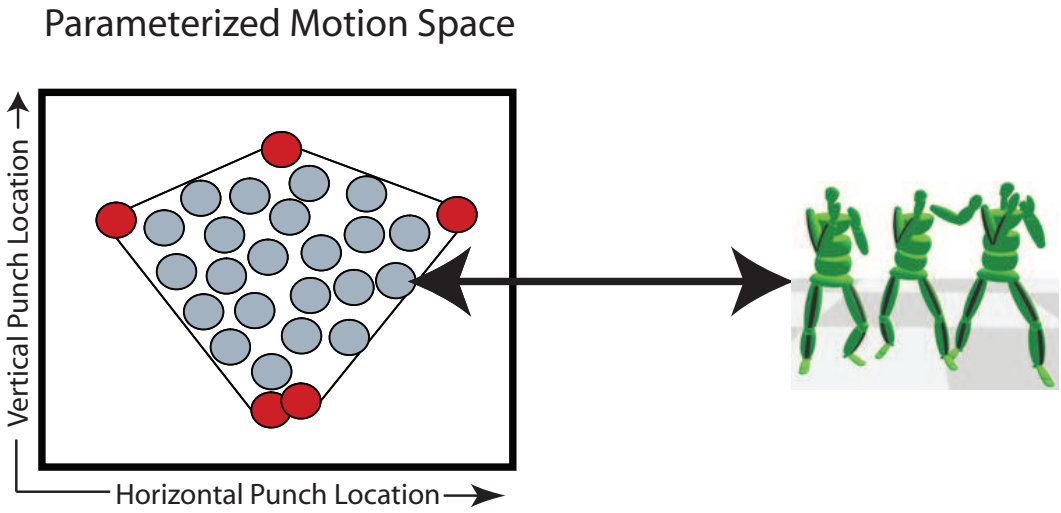


Figure 3.6 A parametric motion space representing punching motions, parameterized on the location of a punch. Each point in this parametric motion space maps to an entire punching motion, not just a single pose.

dynamic timewarping to iteratively converge on a time alignment that explicitly takes into account pose variations. This technique can be used to improve timewarping results but requires a considerable increase in computation time. A simple, greedy timewarping algorithm can also work in cases where the motions are very similar. But I have found that greedy search does not work well for many of the more complex cases of timewarping in this dissertation, and I recommend using a method based on dynamic programming.

3.5 Blending-Based Parametric Synthesis

Many different types of motion are easily described using a small number of *parameters*. For instance, a punching motion might be described by the location of a punch, or a stair climbing motion might be described by the height of the steps. It can be convenient to use these high-level parameters as a way of describing what motions are requested by a user or application. *Parametric synthesis* describes the set of techniques that can generate motions based solely on these high-level parameter vectors. The infinite set of motions that can be generated using a parametric synthesis method can be mapped to the parameter space, forming a *parametric motion space*. Each point

in a parametric motion space maps from a parameter vector to an entire motion that meets those parameters (see Figure 3.6).

One popular method for parametric synthesis is to synthesize new motions that meet requested parameters by blending together example motions from that space (see Section 2.1.2.2 for a review of these techniques). In this dissertation, I perform blending-based parametric synthesis using the method of Kovar and Gleicher [KG04]. Because Kovar and Gleicher's method is so essential to my work on gaze control (Chapter 5) and parametric motion graphs (Chapter 6), I will use this section to provide an in depth overview of their approach.

In [KG04], Kovar and Gleicher described how to:

1. *Automatically find and extract logically similar motions*, or motions where the character is performing the same basic action, from a motion database. Their technique uses an iterative method that locates motions that look similar to a query motion, where similarity is defined by the metric presented in Section 3.2, and then repeats the process on each of the similar looking motions.

2. *Map these logically similar motions to a parametric motion space*. Each of the logically similar motions are registered to each other in both time and space using methods similar to those presented in Sections 3.3 and 3.4. These registered motions become example motions in the motion parameter space by mapping the motion to its relevant parameter vector. For instance, the algorithm might identify the location in space where the character punches. Figure 3.7a shows these example motions mapped in parametric motion space for a space of punching motions.

3. *Sample blends from the space to build a parametric motion*. New logically similar motions can be generated by blending together the base example motions. Yet, because of the non-linearity of human motion, the parameter vectors associated with these new blended motions are unlikely to be a similarly proportioned blend of the base example parameter vectors. So, Kovar and Gleicher sampled the set of motions that can be blended together from these base

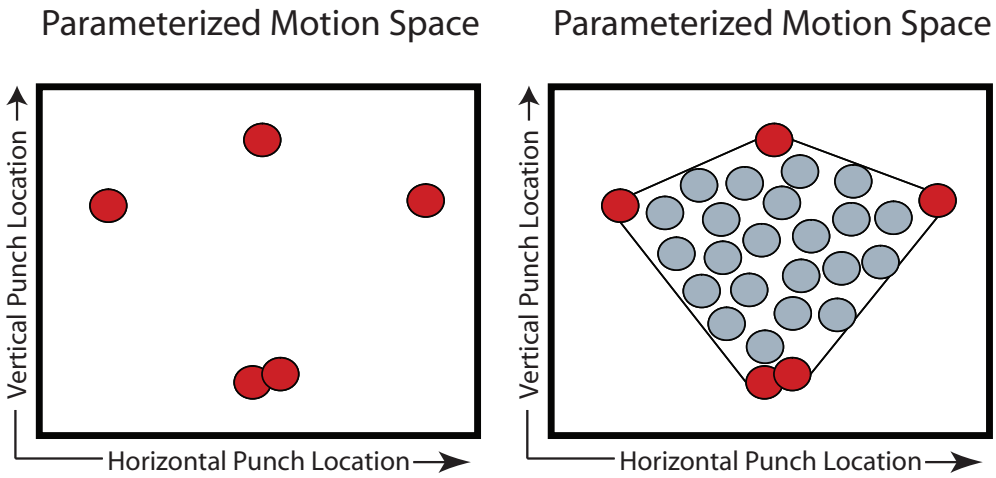


Figure 3.7 A sampled parametric motion space of punching motions, parameterized on the location where a character punches. (a) Logically similar punching motions from a motion database are mapped to the parametric motion space (shown with dark, red circles) (b) Blends between the example punching motions are sampled, producing additional data points (shown with light, gray circles) in parametric motion space.

example motions. Each of these sampled blends is then analyzed, just as the original examples were, to produce an associated parameter vector. The blend information then becomes a new data point in parametric motion space at this computed parameter vector. Figure 3.7b shows these blend samples mapped in the punching parametric motion space.

4. *Use the samples from the parametric motion space to synthesize new motions that accurately meet user-requested parameter vectors*. Once the samples in parametric motion space are sufficiently close together, k-nearest neighbor interpolation is used to synthesize new motions that meet specified parameter vectors. Other methods for blending-based parametric synthesis use a linear fit model to perform this scattered data interpolation instead of k-nearest neighbor interpolation. But k-nearest neighbor interpolation has a number of advantages: it constrains interpolation weights to reasonable, positive values, resulting in more realistic-looking motion; it is computationally efficient for large data sets and does not require a costly optimization to calculate; and it projects all outlier parameter vector requests back into the space enclosed by the original example motions.

I chose to use the method of Kovar and Gleicher to perform parametric synthesis because it produces high-quality results, allows for quick experimentation with many different types of motion, provides a simple and efficient method for producing motion clips at runtime, and results in parameteric motion spaces that are *smooth*. A parameteric motion space is considered smooth if small changes in the input parameters produce small changes in the generated motion. This smoothness property is important to my work on parametric motion graphs in Chapter 6.