Chapter 6

Parametric Motion Graphs

This chapter presents a new approach for controlling interactive human character using a novel example-based motion synthesis data structure called a *parametric motion graph*. Like other example-based data structures, parametric motion graphs provide easy authoring of high-quality motions, but they also supply the responsiveness, precise control, and flexibility demanded by interactive applications. A parametric motion graph describes possible ways to generate seamless streams of motion by concatenating short motion clips generated through blending-based parametric synthesis. As described in Section 3.5, blending-based parametric synthesis allows accurate generation of any motion from an entire space of motions, by blending together examples from that space. For instance, parametric synthesis can generate motions of a person picking up an item from any location on a shelf by blending together a small set of example motions. While neither seamless motion concatenation nor parametric synthesis is a new idea, by combining both techniques, parametric motion graphs provide accurate control through parametric synthesis and can generate long streams of high-fidelity motion without visible seams using linear-blend transitions.

In contrast to many other automated methods for representing transitions between motions (see Section 2.2), parametric motion graphs are highly structured, facilitating efficient interactive character control. The nodes of a parametric motion graph represent entire parametric motion spaces that produce short motions for given values of their continuously valued parameters. The directed edges of the graph encode valid transitions between source and destination parametric motion spaces. This structure efficiently organizes the large number of example motions that can be blended together to produce the final motion streams. Because of this structure, I have been able

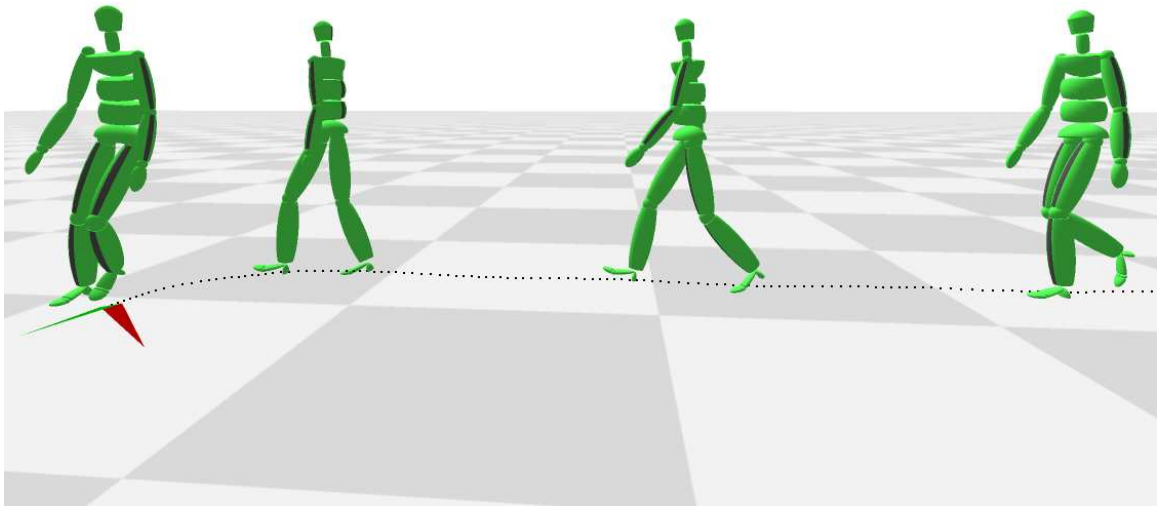

Figure 6.1 An interactively controllable walking character using parametric motion graphs to smoothly move through an environment. The character is turning around to walk in the user-requested travel direction, depicted by the red arrow on the ground.

to easily author interactively controllable characters that can walk, run, cartwheel, punch, change facing direction, and/or duck in response to user-issued requests.

While prior work on synthesis by concatenation has focused on representing seamless transitions between individual clips of motion (see Section 2.2.1), I face the problem of defining valid transitions between parametric *spaces* of motions, where it is not often possible to transition from any motion in one parametric motion space to any motion in another. For example, consider a parametric motion space representing a person taking two steps, parameterized on curvature. One can imagine that this parametric motion space can follow itself; a person can take two steps, and then take two more, and so on. However, a transition should not be generated between a motion where the character curves sharply to the right and another where the character curves sharply to the left; the resulting transition would not look realistic. Thus, the edges in a parametric motion graph must encode the *range* of parameters of the target space that a motion from the source space can transition to, as well as the correct way to make the transition between valid pairs of source and destination motions. The key challenge to parametric motion graphs is finding a good way to compute and represent these transitions. By approaching the problem from a sampling perspective, I provide an efficient way to compute and encode the edges of a parametric motion graph, allowing automated authoring and fast transition generation at runtime.

To provide parametric motion graphs as a method for interactive character control, this chapter describes how to:

Build Parametric Motion Graphs: Using a method based on sampling, I can efficiently locate and represent transitions between parametric motion spaces.

Extract Data from Parametric Motion Graphs: My representation of transitions allows fast lookup of possible transitions at runtime using interpolation.

Use Parametric Motion Graphs for Interactive Control: Because parametric motion graphs are highly structured, they facilitate the fast decision-making necessary for interactive character control. Furthermore, because all motion clips in the graph are generated using parametric synthesis, motions accurately meet relevant constraints.

The rest of this chapter is organized as follows. Section 6.1 details my methods for building and extracting information from a parametric motion graph. Then, Section 6.2 presents results from some experiments using parametric motion graphs, including controlling interactive characters in realtime. Finally, Section 6.3 concludes with a general discussion of the presented technique, including a number of the technique's limitations.

6.1 Parametric Motion Graphs

This section describes in detail the methods developed for building parametric motion graphs and extracting data from them. My methods for controlling a character using a parametric motion graph are presented later in Section 6.2.

6.1.1 Building a Parametric Motion Graph

To facilitate efficient motion synthesis at runtime, much of the needed computation for controlling interactive characters is done while building a parametric motion graph offline. A parametric motion graph only needs to be built once, resulting in a small text file representation of the graph that can be loaded at runtime.

As described at the beginning of this chapter, each node of a parametric motion graph represents a parametric motion space implemented using blending-based parametric synthesis. For all of the examples in this dissertation, blending-based parametric synthesis is performed using the method presented in Section 3.5. While the nodes of a parametric motion graph can be built using this existing technique, the key challenge is finding a way to identify and represent possible transitions between these parameterized nodes. Because the parametric motion spaces represented by the graph nodes are smooth, as described in Section 3.5, I can tackle this challenge using sampling and interpolation. The rest of this subsection describes in detail how to identify and represent edges between source and target graph nodes, $\mathbf{N}_s$ and $\mathbf{N}_t$ respectively. Throughout this description, the parametric motion space represented by node $\mathbf{N}_i$ is denoted by $\mathcal{P}^i(l)$, where l is a vector of relevant motion parameters, such as the target of a punch; a parametric motion space produces a short motion, $\mathbf{M}_i$, for any given value, l_i, of its continuously valued parameters.

6.1.1.1 Identifying Transitions Between Motion Spaces

To start, consider the case where the nodes $\mathbf{N}_s$ and $\mathbf{N}_t$ represent small motion spaces whose valid parameter ranges only include a single point. This case reduces to the traditional synthesis-by-concatenation problem; is there a frame of motion near the end of the motion generated by $\mathbf{N}_s$, $\mathbf{M}_1$, and a frame of motion near the beginning of the motion generated by $\mathbf{N}_t$, $\mathbf{M}_2$, that are similar enough to allow a linear-blend transition from one to the other over a short window centered at these frames? A good transition exists from $\mathbf{M}_1$ to $\mathbf{M}_2$ if and only if there exists a frame, t_1, near the end of $\mathbf{M}_1$ and a frame, t_2, near the beginning of $\mathbf{M}_2$ such that $\mathbf{D}(\mathbf{M}_1(t_1), \mathbf{M}_2(t_2)) \leq T_{GOOD}$, where T_{GOOD} is a tunable threshold. If the distance value of the optimal transition point found using the method presented in Section 3.3 is below T_{GOOD}, then it is possible to transition between $\mathbf{M}_1$ and $\mathbf{M}_2$ at that point, (t_o^1, t_o^2).

Now consider the general case where $\mathbf{N}_s$ and $\mathbf{N}_t$ represent larger spaces. For any sufficiently large space, it is unlikely that the motions represented by the space look similar enough to be treated like a single motion. For instance, in the walking example discussed at the beginning of this chapter, the walking character can only transition to other walking motions where the character walks at a similar curvature to its current one. However, since each parametric motion space represents an infinite number of motions, it is infeasible to compare all possible pairs of motions represented by each of the parameterized nodes. One possible approach is to reduce each parametric motion space to a discrete number of motions chosen from the full space. To find and represent good transitions between all pairs of motions from a source set of size m and a target set of size n, I would need to repeat the technique described above mn times. Unfortunately, by transforming a continuous motion space into a discrete set of motions, I lose much of the accuracy that parametric synthesis provides; accuracy can be increased by adding more motions to these sets but this results in a combinatorial explosion in the number of required comparisons and the amount of space needed to store the possible transitions.

Yet, in a smooth parametric motion space, motions generated for any local neighborhood of parameter space look similar. For example, consider a parametric motion space representing motions of a person punching, parameterized on the location of the punch. Two motions in this space

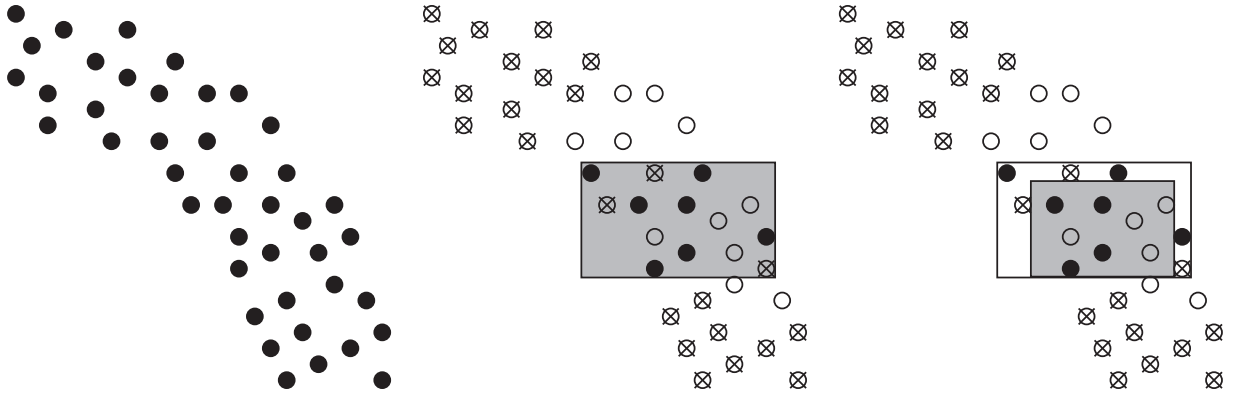

Figure 6.2 Process of determining the valid transition region in target parameter space. (a) A set of randomly chosen samples from the target space. (b) Darkened circles produce good transitions, crossed out circles produce bad transitions, and empty circles produce neutral transitions. The shaded box encloses all good samples but also includes some bad samples. (c) The adjusted, shaded box excludes all bad samples. In practice, little to no adjustment is usually made to the bounding box.

where the punches land $1mm$ apart look similar. In this case, I can compute the possible transitions from one of these motions and use the result for both. This observation leads me to approach the problem of identifying and representing transitions between parametric motion spaces using sampling, extending the method presented in Section 3.3 for locating possible linear blend transitions between individual motions.

6.1.1.2 Building a Parametric Motion Graph Edge

An edge between source and target nodes, $\mathbf{N}_s$ and $\mathbf{N}_t$ respectively, maps any point, l_i^s, in $\mathcal{P}^s$ to the subspace of $\mathcal{P}^t$ that can be transitioned to from $\mathbf{M}_i^s = \mathcal{P}^s(l_i^s)$. It also supplies the time at which that transition should occur. Assuming it is possible to transition from every point in $\mathbf{N}_s$ to some subspace in $\mathbf{N}_t$, we can build an edge between these nodes using sampling. I start by generating two lists of random parameter samples, $\mathbf{L}^s = \{l_1^s, \ldots, l_{n_s}^s\}$ and $\mathbf{L}^t = \{l_1^t, \ldots, l_{n_t}^t\}$ (see Figure 6.2a). In order to accurately capture the variations in the target space, n_t should be large. The exact number depends on the size of the parameter space, but I have found 1000 samples to be more than enough for all of the cases I have tried, even for parametric motion spaces that have three

parameters. In contrast, n_s should be small, while still covering the extremes of the source space, as this number affects the amount of storage needed for an edge as well as performance efficiency of the graph when used to produce motion at runtime (see Section 6.2.4). For the examples in this dissertation, n_s ranged from 4 to 200.

Now consider a sample from $\mathbf{L}^s$, l_1^s. This sample corresponds to the motion $\mathbf{M}_1^s = \mathcal{P}^s(l_1^s)$. I can determine if $\mathbf{M}_1^s$ can transition to each motion represented by the parameter samples in $\mathbf{L}^t$ by computing the optimal transition point with each motion $\{\mathbf{M}_1^t, \ldots, \mathbf{M}_{n_t}^t\}$ using the method presented in Section 3.3. Samples from $\mathbf{L}^t$ that produce good transitions are added to the list of parameter samples $\mathbf{L}_{GOOD}^t$.

Using the observation that motions close in parameter space look similar, I can assume that any parameter vector for $\mathcal{P}^t$ whose nearest parameter samples from $\mathbf{L}^t$ appear in $\mathbf{L}_{GOOD}^t$ can also be transitioned to from $\mathbf{M}_1^s$. Thus, the list $\mathbf{L}_{GOOD}^t$ defines the subspace of $\mathcal{P}^t$ to which $\mathbf{M}_1^s$ can transition.

Unfortunately, I cannot represent the subspace of $\mathbf{N}^t$ that can be transitioned to from $\mathbf{M}_1^s$ by listing the points in $\mathbf{L}_{GOOD}^t$ because, as described at the beginning of Section 6.1, I plan to determine what transitions are possible at runtime using a simple and efficient interpolation scheme (as shown in Figure 6.3); interpolating between potentially different numbers of uncorrelated points in a meaningful way is difficult, if not impossible. So, instead, I represent each subspace as a simple shape that can always be interpolated (i.e., bounding boxes, spheres, triangles). I have found axis-aligned bounding boxes work well for my data; I use axis-aligned bounding boxes to represent all of the transition parameter subspaces.

Using simple, easily interpolated shapes to represent transition regions introduces a considerable problem. Any simple shape that contains all points in $\mathbf{L}_{GOOD}^t$ could also contain other points from $\mathbf{L}^t$ that were not deemed good transition candidates (see Figure 6.2b). To guarantee that bad transitions are not included in the transition subspace of $\mathbf{N}^t$, I take a conservative, double threshold approach. First, while constructing the list $\mathbf{L}_{GOOD}^t$, I also form a list, $\mathbf{L}_{BAD}^t$, containing all samples from $\mathbf{L}^t$ that generate motions whose optimal transition point distance is greater than T_{BAD}, where $T_{BAD} \geq T_{GOOD}$. Next, I compute the bounding box of all parameter samples in $\mathbf{L}_{GOOD}^t$. Finally,

I consider each sample in $\mathbf{L}^t_{BAD}$; if the sample falls within the subspace defined by the bounding box, I make the minimal adjustment to the dimensions of the bounding box so that the sample falls at least ϵ away, where $\epsilon > 0$. In this way, I construct a bounding box that contains many, if not all, of the samples from $\mathbf{L}^t_{GOOD}$ without including any of the samples from L^t_{BAD}. Neutral samples from $\mathbf{L}^t$ whose optimal transition point distance falls between T_{GOOD} and T_{BAD} are considered good enough if they fall within the transition subspace of $\mathbf{N}^t$ but will not be explicitly included in the space (see Figure 6.2c). In practice, the system makes few bounding box adjustments to remove bad samples and in most cases makes none at all.

I also compute a single transition point from $\mathbf{M}^s_1$ to any of the motions located in the subspace of $\mathbf{N}^t$ defined by the computed bounding box. In Section 3.3, I described the optimal transition point of two motions as the pair of frames where the two motions are most similar. For computing a generic transition point for the entire subspace, it is useful to normalize these frame numbers to the range 0 to 1. Again, because nearby motions in a motion space look similar, the optimal transition points are likely to be at similar normalized times. So, I average the normalized optimal transition points for each sample of $\mathbf{L}^t_{GOOD}$ that falls inside the adjusted bounding box to calculate the normalized transition point for the subspace.

Putting all the pieces together, an edge can be defined between $\mathbf{N}^s$ and $\mathbf{N}^t$ as a list of transition samples, one for each parameter vector in $\mathbf{L}^s$. Each sample includes:

- The value of the parameter vector l^s_i
- The computed transition bounding box for l^s_i
- The average, normalized transition point for l^s_i

I could also store the average alignment transform between the motion $\mathbf{M}^s_i$ and each of the motion samples in $\mathbf{L}^t_{GOOD}$ but recomputing this alignment using the method presented in Section 3.3 is fast; instead I save storage space by computing the alignment transform for each transition at runtime.

Up until this point, I have assumed that I can transition from every point in $\mathbf{N}_s$ to some subspace of $\mathbf{N}_t$. I define that a transition exists between nodes $\mathbf{N}_s$ and $\mathbf{N}_t$ if and only if for any motion

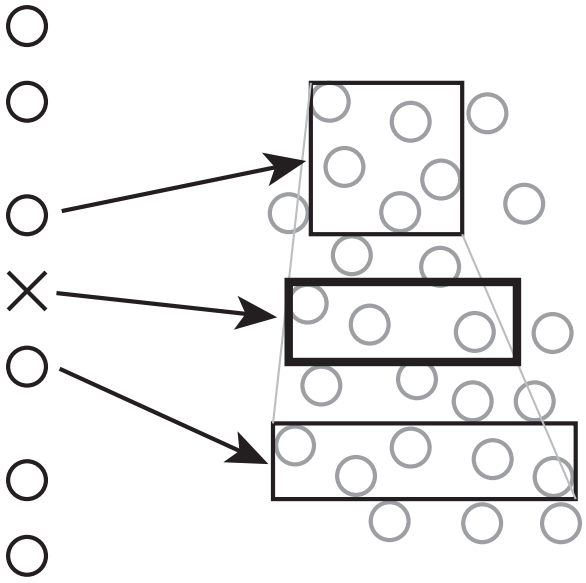

Figure 6.3 Mapping a parameter vector, depicted by the X, from the 1-D parameter space on the left, to a valid transition region in the 2-D parameter space on the right. X's bounding box is the weighted average of the bounding boxes for its 2-nearest neighbors.

contained in $\mathbf{N}_s$ there exists *some* subspace in $\mathbf{N}_t$ that it can transition to. Thus, if I find any sample in $\mathbf{L}^s$ whose adjusted bounding box is empty, I cannot create an edge between $\mathbf{N}_s$ and $\mathbf{N}_t$

6.1.2 Extracting Data from a Parametric Motion Graph

Synthesizing motion using a parametric motion graph is quick and efficient. The data that is stored in each node of the graph allows fast lookup for possible transitions. In particular, given the node, $\mathbf{N}_s$, and relevant parameter vector, $\tilde{l}^s$, for a motion clip, I can determine what subspaces of other parametric motion spaces can be transitioned to as well as when that transition should occur.

For each outgoing edge of $\mathbf{N}_s$, begin by finding the k-nearest neighbors to $\tilde{l}^s$ from the transition sample list, in terms of Euclidean distance, where k is normally one more than the number of dimensions of $\mathcal{P}^s$. Call these neighbors $l^s_1, \ldots, l^s_k$, ordered from closest to farthest from $\tilde{1}^s$. Following the work of Allen et al. [ACP02] on skinning human characters using k-nearest neighbor interpolation and on Buehler et al.'s work on rendering lumigraphs using k-nearest neighbor

interpolation [BBM+01], each l_i^s is associated with a weight, w_i:

$$w_i = \frac{w_i'}{\sum_{j=1}^{k} w_j'} \quad [6.1]$$

$$w_i' = \frac{1}{\varepsilon(\tilde{l}^s, l_i^s)} - \frac{1}{\varepsilon(\tilde{l}^s, l_k^s)} \quad [6.2]$$

where ε gives the Euclidean distance between parameter samples. This method of determining weights has two relevant advantages over using a linear map constructed as a best fit optimization over weights and motion parameters. First, this method does not introduce large negative weights. Using a linear fit method, these large negative weights often appear in order to artificially produce a better fit, even though the quality of the results suffer. Second, computing weights using this algorithm is fast as it does not require a costly global optimization and scales well with the number of example motions in the database.

For any outgoing edge of $\mathbf{N}_s$, calculate the subspace of the target node, $\mathbf{N}_t$, that can be transitioned to, $\mathbf{B}(\mathbf{N}_s, \mathbf{N}_t)$, as follows:

$$\mathbf{B}(\mathbf{N}_s, \mathbf{N}_t) = \sum_{i=1}^{k} w_i * \beta(l_i^s) \quad [6.3]$$

where $\beta(l_i^s)$ gives the value of the bounding box for the sample l_i^s, represented by the location of the box's center and its width in each dimension, as stored in the edge (see Figure 6.3). Similarly, compute the normalized transition point as a weighted sum of the average, normalized transition points for each l_i^s stored in the edge.

6.2 Results

This section provides details for some of the example parametric motion graphs I designed for interactive character control. Following the description of these graphs, I present the results of a number of experiments for testing the usefulness of these graph structures in interactive applications.

Graph Name	# of Nodes	# of Edges	# of Example Motions
Walking	1	1	44
Running	1	1	198
Cartwheeling	1	1	10
Walking and Running	2	4	242
Many Everyday Actions	7	14	256
Boxing	3	9	275

Table 6.1 Size and make-up of the parametric motion graphs in this dissertation. Each line provides the name of the parametric motion graph, the number of nodes in that graph, the number of edges connecting those nodes in the graph, and the total number of example motions organized by the graph.

6.2.1 Graphs

I have constructed six different parametric motion graphs in order to show the utility of the technique. These graphs are described throughout this section. Refer to Table 6.1 for a summary of the size of these graphs in terms of number of nodes, edges, and example motions.

The process of building parametric motion graphs is highly automated. An author starts by choosing the parametric motion spaces needed for the graph from an available motion space database built using the blending-based parametric synthesis technique described in Section 3.5. These parametric motion spaces then appear as disconnected nodes in the graph.

Next, the author chooses two nodes to generate an edge between and specifies values for $\mathbf{T}_{GOOD}$, $\mathbf{T}_{BAD}$, n_s, and n_t. While it is possible to set the values of $\mathbf{T}_{GOOD}$ and $\mathbf{T}_{BAD}$ without user input, the ability to adjust these values allows an author to determine where to set the tradeoff between motion quality and flexibility discussed later in this section. In practice, it took two or three iterations in order to tune the parameters $\mathbf{T}_{GOOD}$ and $\mathbf{T}_{BAD}$ for each edge. Empirically, setting $\mathbf{T}_{GOOD}$ to $.5$ and $\mathbf{T}_{BAD}$ to $.7$ served as a good starting point. For my example graphs, the amount of time it took to generate a single edge varied from $2 - 147$ seconds, depending on the complexity of the source and target parametric motion spaces.

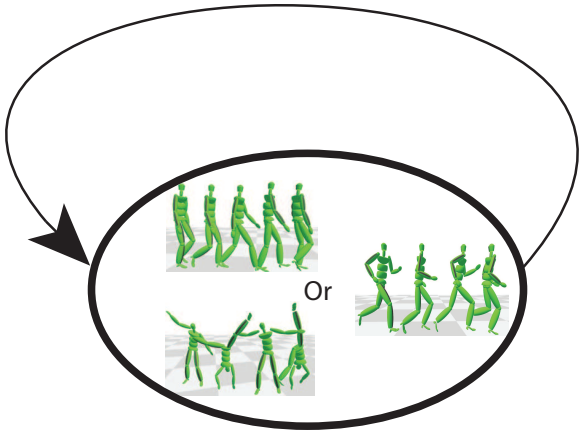

Figure 6.4 Graph for walking, running, or cartwheeling.

6.2.1.1 Single Node Locomotion Graphs

While other researchers have dealt specifically with generating controllable streams of locomotion in realtime (see Section 2.2.4 for a review of these methods), I chose to create several single-node locomotion graphs because it is easy to see artifacts in this commonly performed activity. In my first graph, I encoded streams of walking motion that only contain smooth turns. This graph consists of a single node representing a parametric motion space of a character walking for two steps at different curvatures. The parametric motion space maps the angular change in the character's travel direction from the beginning to the end of the motion (between -131 degrees and 138 degrees) to synthesized motions. Similarly, I built a running graph as a single node representing a parametric motion space with a valid angular travel direction change between -120 degrees and 99 degrees.

Since my technique requires little authoring effort, it is possible to experiment with non-obvious motions. I also built a parametric motion graph that encodes locomotion control through cartwheeling. Like the graphs for walking and running, my cartwheel locomotion graph contains only a single node. This node represents a parametric motion space of a character doing a cartwheel, rotating towards the right by varying amounts on one foot, and then doing a cartwheel

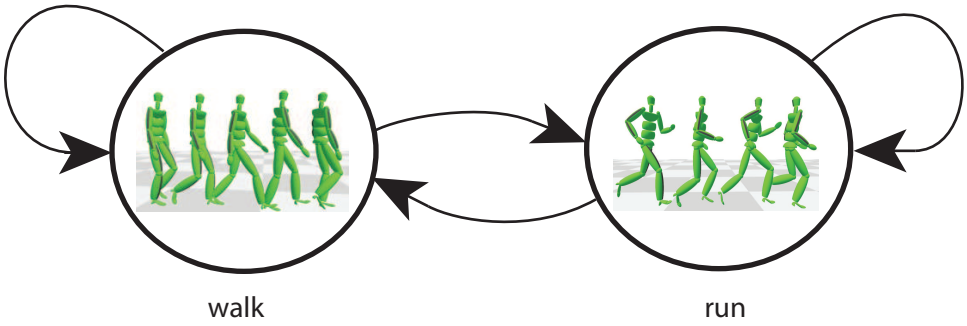

Figure 6.5 A locomotion graph for walking and running.

in another direction. Again, the parametric motion space maps the angular change in travel direction of the character from the beginning of the motion to the end (between -13 degrees and 157 degrees) to synthesized motions.

Each of these single node locomotion graphs take less than 5 minutes to build from beginning to end using my unoptimized system.

6.2.1.2 General Graphs

In addition to single-node locomotion graphs, I have also built several larger graphs. The simplest is a two-node graph that combines the walking and running nodes described earlier (see Figure 6.5). This graph can control the travel direction of a character that can both run and walk.

I have also built a seven-node, fourteen-edge graph containing motions for a number of different everyday actions: walking and running at different curvatures, sitting down and standing up from chairs of heights between 1ft and 1.9ft tall, stepping up onto and stepping off of platforms of heights between $.8$ft and 1.8ft tall, and leaping over distances between 2 and 3ft (see Figure 6.6). It takes about 11 minutes to build this graph. The final graph organizes a total of 256 example motions so that they can be blended to produce continuous streams of controllable animation.

In order to show that my technique works when controlling a number of different non-locomotion actions, I built a parametric motion graph that encodes the motions of a boxer punching, ducking, and "dancing" from one foot to the other. The boxing graph consists of three nodes. The first node represents all motions of a boxing character punching to some location in a 6ft wide, 2ft tall, and 5ft

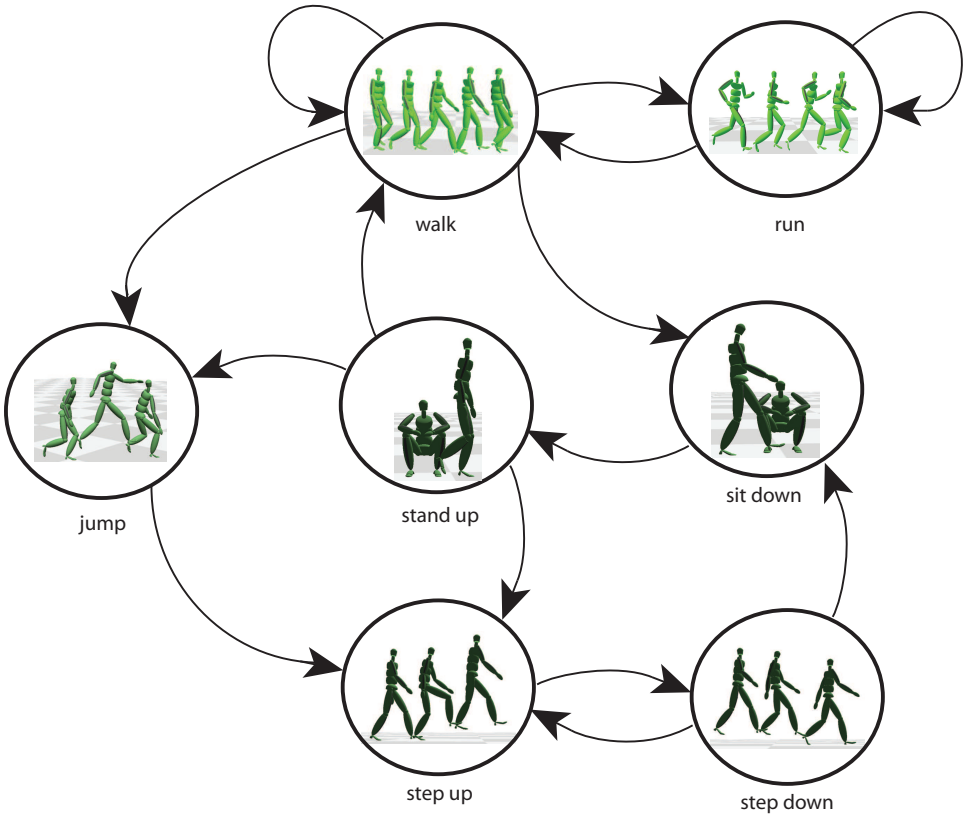

Figure 6.6 A graph for controlling a number of different everyday actions.

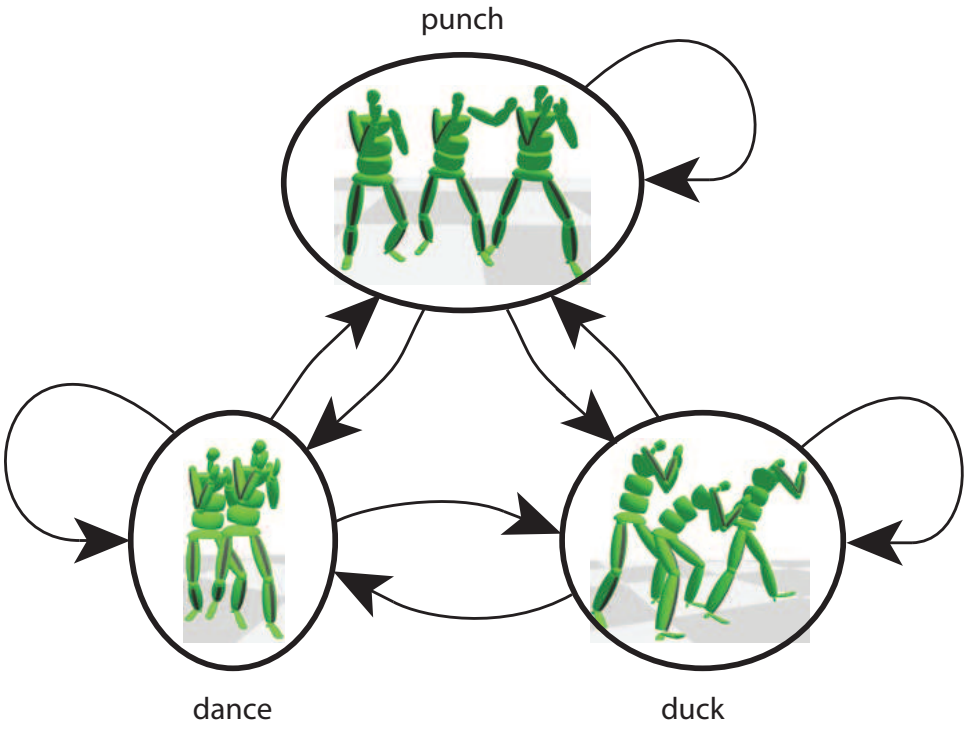

Figure 6.7 A boxing graph.

deep space. The parametric motion space maps desired punch locations in relation to the starting configuration of the root to synthesized punching motions. The second node of the boxing graph represents motions of a boxing character ducking below different heights (between 3.4ft and 5.6ft from the ground) and is parameterized on how low the character ducks. The third and final node encodes motions of a character "dancing" from one foot to another while maintaining a boxing ready stance. When "dancing", the character rotates by different amounts (between -27 and 46 degrees). Thus, the "dancing" motion space maps the change in facing direction from the beginning of the motion to the end of the motion to synthesized "dancing" motions. In total, the parametric motion spaces used for these graph nodes blend between 275 different motion-captured examples. A discrete motion transition graph, like those described in Section 2.2.2 and Section 2.2.3, that represents transitions between this number of motions would be large and unwieldy. In contrast, the final parametric motion graph (Figure 6.7) contains only nine edges, one connecting every pair of nodes. It takes approximately 7 minutes and 40 seconds to build the graph.

6.2.2 Applications

I implemented a number of different applications to test the usefulness of my technique. In this section, I describe these applications in detail and provide an overview of my results.

6.2.2.1 Random Graph Walks

My first application shows that parametric motion graphs can generate *seamless, high-fidelity motion streams in realtime*. For each of the graphs described in Section 6.2.1, I can produce a random stream of motion by taking random walks on the graph.

I start by choosing a random node and parameter vector from the graph. When the parametric motion space associated with the node is supplied with the chosen parameter vector, I can render a motion that matches this parameter request in realtime using the method presented in Section 3.5. While playing the motion, when I reach the possible transition region, I randomly choose an edge from those leaving the current node. The node that this edge points to is the new target node. Using the method described in Section 6.1.2, I compute the optimal transition point and the parameter

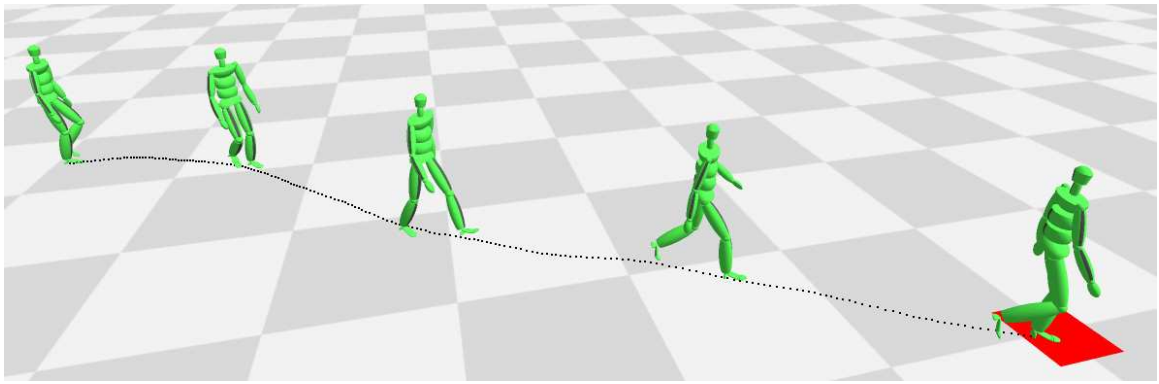

Figure 6.8 Using parametric motion graphs, this character walks to a specified location, depicted by the red square on the ground. The path on the ground plane is not prespecified. It is shown only to illustrate the path the character takes to the target.

subspace of the target node that I can transition to from my current parameter vector. I then randomly choose a new target parameter vector enclosed in this subspace. Finally, when I reach the blending window centered at the optimal transition point, I can append my current motion to my newly chosen motion using a linear blend transition, as described in Section 3.3. This process is then repeated indefinitely to produce an infinitely long stream of motion.

By randomly generating long streams of motion, I can confirm that my technique produces continuous motions and avoids poor transitions. I can also show that the algorithm for synthesizing new motion with a parametric motion graph is efficient enough to be used in an interactive application.

6.2.2.2 Target Directed Control

My second application tests whether my walking character can *accurately* reach a target location using a greedy graph search similar to ones used for locomotion control [SMM05] and crowd control [SKG05]. For this application, I generate a motion stream in the same way as for random graph walks (see Section 6.2.2.1), except that when it is time to choose a new parameter vector from the target bounding box, I choose the parameter vector that best adjusts the character's travel direction towards a target. Figure 6.8 shows that the walking character is able to accurately reach a target location without wandering by using this simple control algorithm.

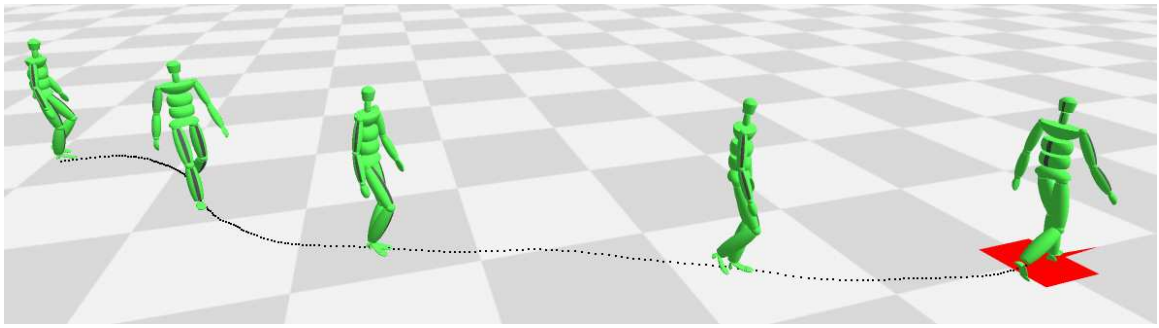

Figure 6.9 Using parametric motion graphs, this character walks to a specified location, and arrives while oriented in the requested direction. The red box and arrow on the ground depict the desired location and orientation respectively. The path on the ground plane is not prespecified. It is shown only to illustrate the path the character takes to the target.

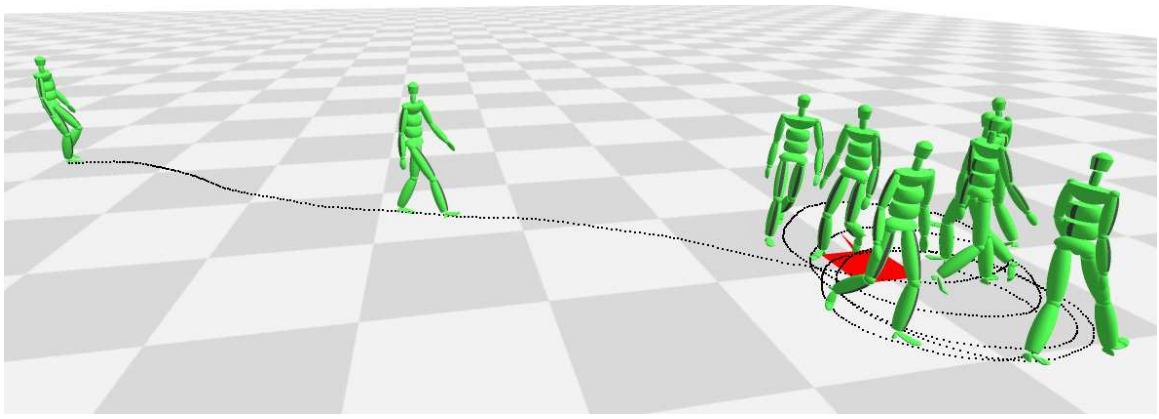

Figure 6.10 Using parametric motion graphs, this walking character cannot arrive at the specified location while oriented in a particular direction. The turning radius of the character is too small.

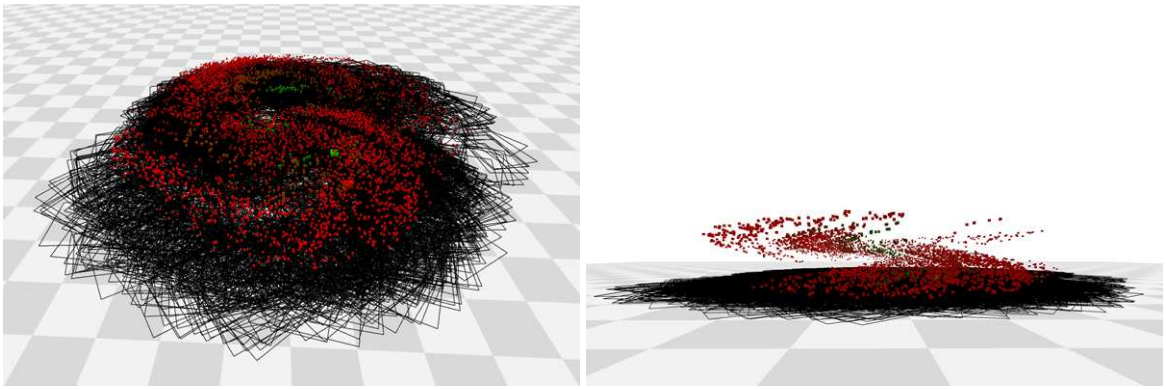

Figure 6.11 Locations that the walking character can reach in a short period of time. Height corresponds to possible orientations that the character can be in when it reaches the location. This experiment shows that most locations on the ground plane can be reached by the walking character but that there are only a small number of orientations in which the character can be in when they arrive at each of these locations.

I also allow a user to request that the character reach the target location oriented in a particular direction. For this case, I choose the parameter vector that both adjusts the character's travel direction towards the target and orients the character towards the desired facing direction. I place more weight on the orientation component of this optimization function as the character gets closer to the target. In several cases, the walking character can perform the requested action well (see Figure 6.9). But I find that in others, the character approaches the target and then turns in circles trying to orient itself (see Figure 6.10). This result is anticipated as I know that the character's minimum turning radius is quite large.

Inspired by the work of Reitsma and Pollard [RP04, RP07], I used a discrete, brute force method to embed the walking parametric motion graph in the environment in hopes of better understanding this problem. The embedding made it clear that the walking character could meet location constraints within a reasonable radius but that for most locations, there were only a few orientations that the character could be in when they arrived. Figure 6.11 shows the results of this embedding.

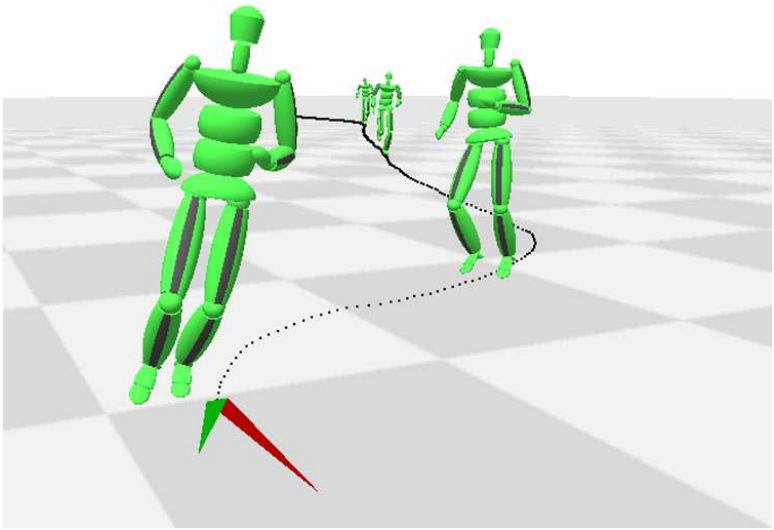

Figure 6.12 An interactively controllable running character using parametric motion graphs to smoothly move through an environment. The character has changed running direction in order to travel in the user-requested direction depicted by the red arrow.

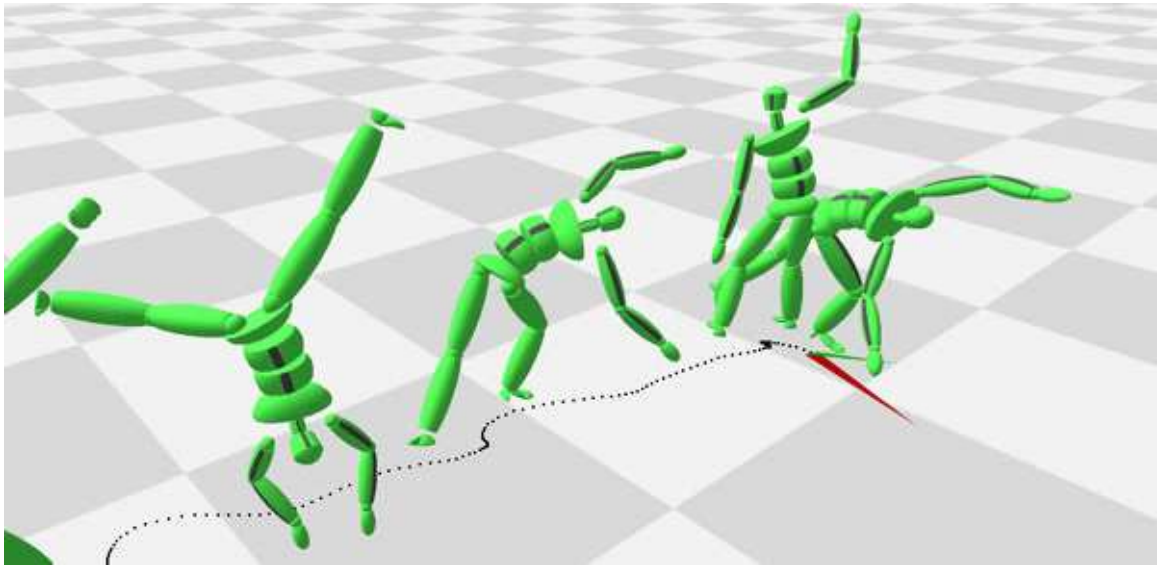

Figure 6.13 An interactively controllable cartwheeling character using parametric motion graphs to smoothly move through an environment. The character has changed cartwheeling direction in order to travel in the user-requested direction depicted by the red arrow.

6.2.2.3 Interactive Character Control

My last and most important application allows users to interactively control a character, testing all of the necessary characteristics of interactive applications (see Section 1.1). To do this, I attach a function to each node that translates user requests to parameters. For example, for walking and cartwheeling, I wanted a user to control the travel direction of the character by specifying the desired travel direction using a joystick. So, I attached a function to each of these nodes that could compute the angular change between the character's current direction of travel and desired direction of travel.

With these translation functions in place, I can again generate motion streams as I did when generating random graph walks (see Section 6.2.2.1) except that when it is time to choose a parameter vector from the target bounding box, I query the user's current request. Then I use the translation function for the requested node to compute a parameter vector. These parameter values are adjusted so that they fall within the target bounding box if they were not within bounds already.

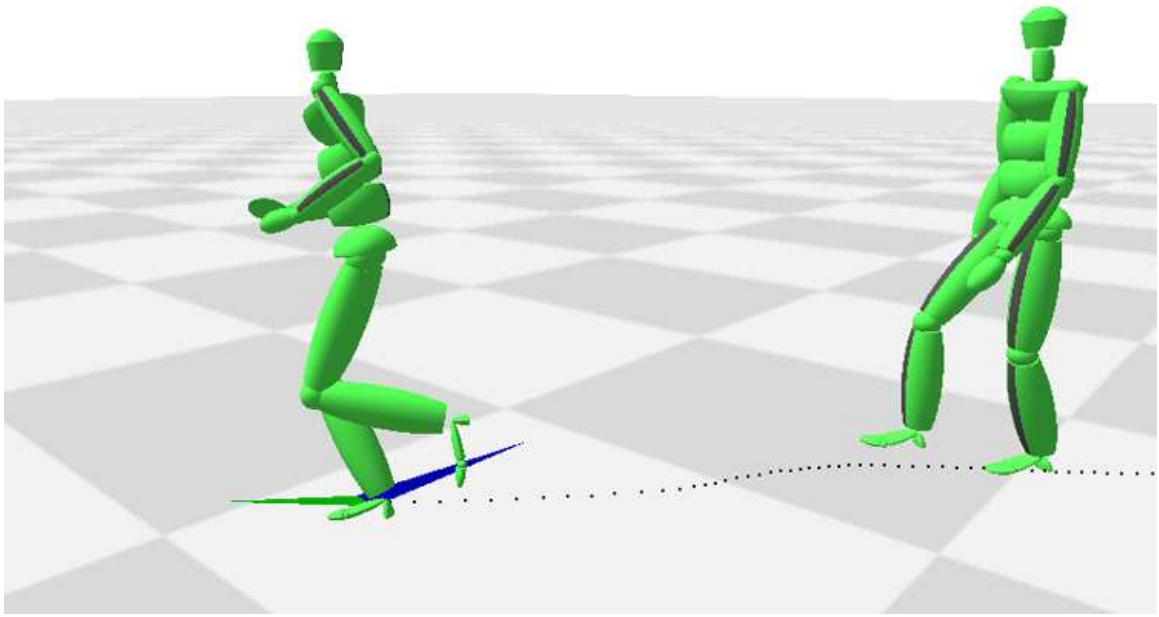

Figure 6.14 An interactively controllable character using parametric motion graphs to smoothly move through an environment by walking or running. The character has just transitioned from walking to running, and is now changing his travel direction in order to meet the user-requested direction depicted by the blue arrow.

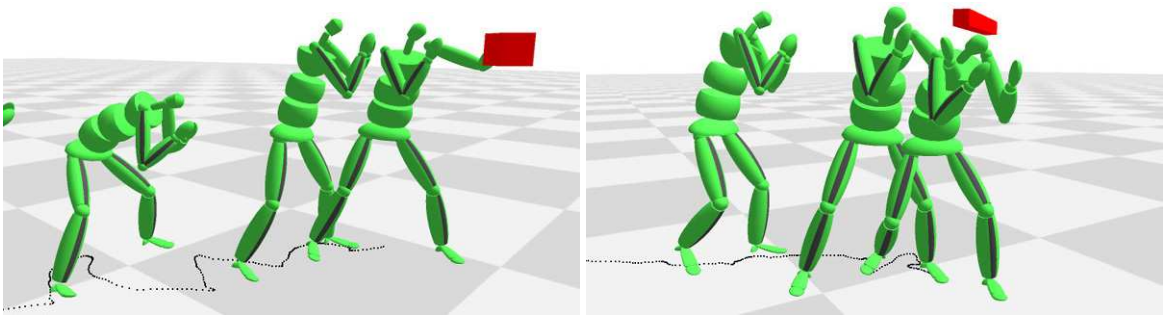

Figure 6.15 An interactively controllable boxing character that uses parametric motion graphs. The character is punching towards a user-requested target in the top image. In the bottom image, the character is ducking below a user specified height.

This process has the effect of creating interactive characters that perform requested actions as accurately as possible without introducing poor transitions between motion clips. By limiting the transitions to good ones, our characters occasionally miss targets; in these cases, the character still "reacts" to the target by choosing a good transition that gets closest to meeting the request. For instance, the boxing character shown in Figure 6.15 occasionally misses its punching target when the target appears on the periphery of the region in front of his body. This is because it is not possible to hit the target without producing a bad transition. So, instead, the character will rotate his body as much as possible and punch near to the target, thus "reacting" to the request but not quite meeting it.

Using this technique, I have produced:

1. a *walking* character whose travel direction can be controlled (see Figure 1.5b).

2. a *running* character whose travel direction can be controlled (see Figure 6.12).

3. a *cartwheeling* character whose travel direction can be controlled (see Figure 6.13). Note that the cartwheeling character is an interesting one because the character only knows how to turn to the right. So, when the character is asked to turn to the left, he makes two large right hand turns. This reaction is not "programmed" into the control structure, instead it happens naturally because of the way parametric motion graphs work.

4. a character who can either *run or walk* in a desired travel direction (see Figure 6.14).

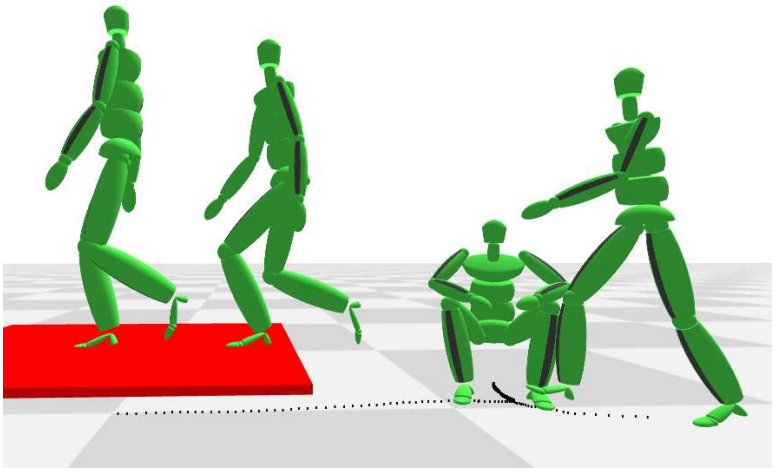

Figure 6.16 An interactively controllable character using parametric motion graphs. The character has just stepped up onto a platform after sitting down in a chair.

5. a *boxing* character that is able to change facing direction while "dancing", punch towards specified 3D locations, and duck below a specified height (see Figure 6.15).

6. a character that can perform *everyday actions* - walking or running in a desired direction, stepping onto and off of platforms, sitting down and standing up from chairs, and leaping over distances (see Figure 6.16)

6.2.3 Comparison with Fat Graphs

Parametric motion graphs are similar to another method for constructing structured motion graphs called fat graphs [SO06]. As described in Section 2.2.3, a fat graph is constructed by first identifying key poses within a motion database that appear many times. These poses are then represented as "hub" nodes within the graph. The edges represent parametric motion spaces of motions that can transition from the same two key poses. Like parametric motion graphs, this structure explicitly combines parametric synthesis with synthesis-by-concatenation methods to produce a structured representation of motion transitions that can be used efficiently at runtime to accurately control a human character.

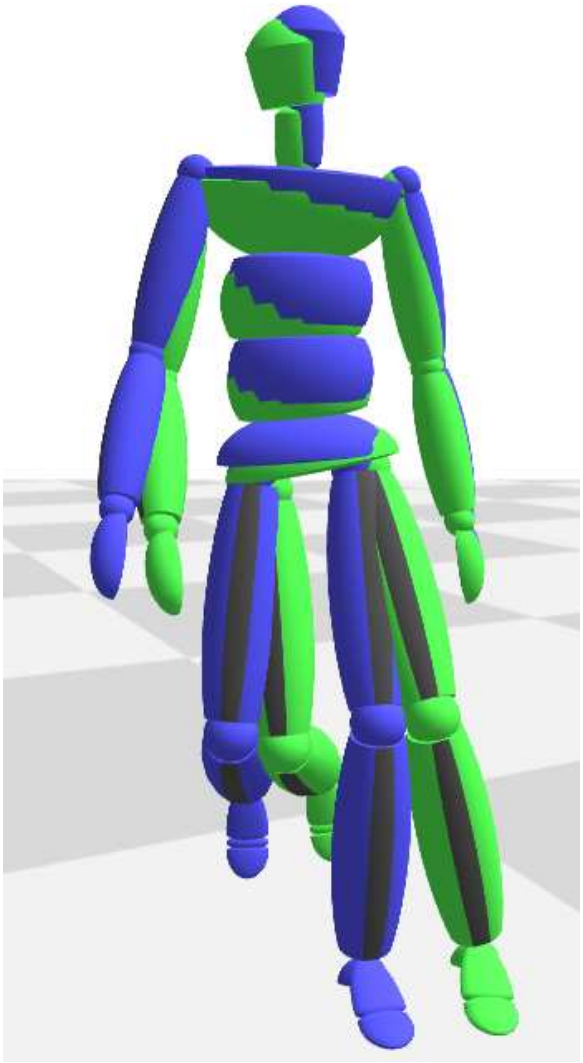

Figure 6.17 At the transition point between two motions of a character turning towards the right, the character using a parametric motion graph remains leaning into the turn (as shown in green) while the character using a fat graph must return to the common transition pose with no lean (as shown in blue), causing the character to “bob” as it goes around the turn.

Yet, parametric motion graphs have a number of advantages over fat graphs. Because all motions representing the same logical action, such as walking or dodging, are explicitly grouped together, parametric motion graphs provide additional logical structure to the graph. A graph author can easily see the logical connections between motion types, allowing the graphs to be designed easily for specific applications.

Parametric motion graphs also represent continuously changing transition points and ranges within a single type of motion. Like Snap-Together Motion, the technique fat graphs are based on (see Section 2.2.3), a fat graph must use more than one "hub" node in order to capture some of this complexity. For instance, in a fat graph representation, a walking parametric motion space might be divided across three parametric motion spaces in order to avoid transitioning from a sharp righthand curvature walking motion to a sharp lefthand curvature walking motion: one where the character is curving a lot towards the right, one where the character is curving a lot towards the left, and one where the character is curving mostly forward. And even if these motions are grouped into these three separate parametric motion spaces, each motion within a single parametric motion space will have exactly the same possible target transition motions. Parametric motion graphs represent continuously changing transition points using sampling.

Fat graphs are also limited in the quality of their results by the use of "hub" nodes; motions are constantly forced to return to the same average pose at each transition point. For instance imagine a character who is walking at a curvature that is very sharp to the right; when that character reaches the "hub" node, he must transition to the average pose of all of the walking motions represented by the parametric graph edge, even if the character continues to walk at a curvature that is very sharp to the right. By forcing motions to return to an average pose at "hub" nodes, motion streams often exhibit repetitive "bobbing" artifacts, as illustrated in Figure 6.17. On the other hand, parametric motion graphs handle natural variations in the transition poses of similar motions.

6.2.4 Algorithm Performance

In this section, I describe how parametric motion graphs perform in each of the six categories described in Section 1.1.

Node Name	Mean Clip Length	Min Clip Length	Max Clip Length
Walking	1.6s	1.4s	1.8s
Running	1.0s	0.8s	1.1s
Cartwheeling	2.9s	2.5s	3.5s
Stepping Up	1.9s	1.9s	2.0s
Stepping Down	1.9s	1.8s	1.9s
Sitting Down	4.8s	4.3s	5.3s
Standing Up	3.0s	2.9s	3.1s
Jumping	1.5s	1.4s	1.6s
Punching	1.1s	0.5s	1.9s
Ducking	1.6s	1.1s	2.6s
Boxing Dance	0.9s	0.6s	1.1s

Table 6.2 Example motion length information for each parametric motion space. Each line in the table contains the name of the parametric motion space, the average length of a motion clip in the parametric motions space, the minimum length of a motion clip in the parametric motion space, and the maximum length of a motion clip in the parametric motion space.

Efficient Synthesis The examples in this paper were computed on a laptop computer with a $1.75GHz$ Pentium M Processor, $1GB$ of RAM, and an ATI Mobility Radeon X300 graphics card. All of the generated motions were sampled at $30Hz$. Each of the generated parametric motion graphs can synthesize and render streams of motion at more than 180 frames per second, consistently. Because I use k-nearest neighbor interpolation and the weighting algorithm presented in Section 6.1.2, synthesis times are nearly independent of the number of example motions in the database. It should be noted however that computation time is dependent on the number of examples being blended together. Because Kovar and Gleicher's method for blending-based parametric synthesis [KG04] limits the number of motions that can be blended together at each of the sample points in a parametric motion space, this time is effectively bounded.

Efficient Data Storage Even for my largest graph, it is possible to store the graph's structure and edge information in a plain text file requiring less than $50KB$ of space. The storage required for this graph structure scales linearly with n_s, or the number of samples from the source motion space (see Section 6.1.1.2). Thus, it is useful to keep n_s small. Since it is unnecessary to densely sample the source nodes, in general n_s can remain low; however, n_s scales exponentially in relation to the dimensionality of the space being sampled [1]. Though, because blending-based parametric synthesis requires a more densely sampled space, it will fail due to high-dimensionality well before the limit of n_s is met for a parametric motion graph. In Section 6.3, I will discuss a possible way to alleviate the dimensionality problem using decoupling methods.

Low Latency or Response Time In terms of responsiveness, my method is limited by my ability to transition between motions only at one point near the end of a clip. Similarly, I do not adjust the parameter vector while generating a motion. These limitations mean that for motion spaces that represent long motions, it may take time for the character to react to

[1] This phenomenon of an exponentially increasing volume caused by adding dimensions to a mathematical space is a well studied problem in statistics. It is often referred to as the *curse of dimensionality*, a term coined by Richard Bellman [Bel57]

user requests. This problem can be lessened by choosing parametric motion spaces that represent short motion clips, as the maximum length of an example clip acts as an upper bound on the amount of time that the user must wait for a transition to a new motion (see Table 6.2 for detailed timing information about the length of the example clips used for this dissertation). Another possible way to improve the response time for some motions is to use a representation of blending-based parametric motions that is specifically designed for continuous, parameter vector changes, such as that in [TLP07].

I advocate improving response times for motions that are necessarily longer in natural ways. For instance, while it may take time for a human to begin accelerating, a gaze cue, such as those that can be generated using my technique in Chapter 5, can be used to indicate that the character is about to start moving; a simple change of gaze can easily be perceived as a response to a motion request.

Accurate Motion Generation and Visual Quality Because the motion clips in a parametric motion graph are generated using existing blending-based parametric synthesis methods, it is possible to produce motion clips that are of high-quality and that accurately meet user requests. However, a tradeoff exists between the quality of the transitions produced between motion clips and the accuracy of those clips. By setting T_{GOOD} to be high, synthesized motions are more likely to be accurate since the graph edge will allow a much larger range of transitions, but this might cause the transitions to look less good. For instance, by setting T_{GOOD} high when building the walking parametric motion graph, it might be possible for a character to transition from any walking motion to any other, allowing very fast changes in curvature. The synthesized motion streams would react accurately to each request for a curvature change as all curvature changes are possible, but the motion would look unrealistic during the transition. By allowing a user to set T_{GOOD} and T_{BAD} manually, I allow the user to explicitly manage this tradeoff.

Automated Authoring The process of authoring a parametric motion graph is highly automated. As described at the beginning of this section, an author must simply choose which existing

parametric motion spaces to connect together, set the tunable thresholds, T_{GOOD} and T_{BAD}, for determining the tradeoff between motion quality and accuracy, and decide how many samples to take from the source and target nodes. It is also possible for my system to identify all possible links between chosen parametric motion spaces automatically, but in practice, the information an author supplies about which types of motions can transition to which other types of motions is invaluable for minimizing the complexity of the graph, allowing more efficient synthesis of controllable motion streams.

6.3 Discussion

As presented, parametric motion graphs are able to produce seamless, controllable motion streams in realtime. The authoring process is highly automated, making parametric motion graphs useful for interactive applications that would not normally have the resources to build the structures necessary for accurate character control.

While I use the method of Kovar and Gleicher [KG04] to produce parametric motion spaces, my methods do not require that motions be generated with any particular parametric motion synthesis method. However, parametric motion graphs do require smooth parametric motion spaces (see Section 3.5); my sampling and interpolation methods depend on nearby motions in parameter space looking similar (see Section 6.1.1). While I have not provided an example, my method should work just as well using a procedural parametric synthesis method, as long as it produces smooth motion spaces.

One limitation to the technique in this chapter is that it cannot represent transitions between two nodes if there is any motion in the source node that cannot transition to the target node (see Section 6.1.1.2). For example, consider two nodes that represent a person walking at different curvatures where the first allows a much wider range of curvatures than the other. Because the extreme motions of the first node do not look like any of the motions in the second node, I will be unable to create an edge between the nodes.

One possible solution to the problem of building edges between partially compatible nodes is to dynamically add additional nodes to the graph when large enough continuous pieces of a source

node can transition to the target node. This new node would represent the same parametric motion space as the first except that its range would be limited to the range of parameters that have valid transitions to the target node. This solution has the drawback of adding greater complexity to the parametric motion graph, a characteristic that should be avoided in order to facilitate fast decision making at runtime. But there may be a way to balance the tradeoff between graph complexity and overall transition representation.

Another extension to this work that could lead to better methods for interactive control is to develop better local search methods than the greedy one in Section 6.2.2.2. Planning a long motion that consists of a series of motion clips using a parametric motion graph is at the moment an unexplored area of research. This lack of planning is in part designed into parametric motion graphs; the ability to make quick decisions without planning allows the graph to react to changing user requests efficiently and often. However, in some circumstances, it could be useful to know how to reach a specific motion outcome within the graph that requires synthesizing multiple motion clips before the outcome can be reached.

This chapter shows that motions for interactive characters can be designed in an automated way, allowing fast, accurate, high-fidelity motion generation in realtime. My method gains the benefits of accurate motion generation using parametric synthesis as well as the ability to make good transitions between clips using a continuous representation of transitions between parameterized spaces of motion. This technique can decrease the amount of time it takes to author interactive characters, increase the accuracy and efficiency of these characters at runtime, and provide high-fidelity motion in a reliable way.