State of the Art in Example-Based Motion Synthesis for Virtual Characters in Interactive Applications

T. Pejsa¹ and I.S. Pandzic¹

¹Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Abstract

Animated virtual human characters are a common feature in interactive graphical applications, such as computer and video games, online virtual worlds and simulations. Due to dynamic nature of such applications, character animation must be responsive and controllable in addition to looking as realistic and natural as possible. Though procedural and physics-based animation provide a great amount of control over motion, they still look too unnatural to be of use in all but a few specific scenarios, which is why interactive applications nowadays still rely mainly on recorded and hand-crafted motion clips. The challenge faced by animation system designers is to dynamically synthesize new, controllable motion by concatenating short motion segments into sequences of different actions or by parametrically blending clips that correspond to different variants of the same logical action. In this article, we provide an overview of research in the field of example-based motion synthesis for interactive applications. We present methods for automated creation of supporting data structures for motion synthesis and describe how they can be employed at run-time to generate motion that accurately accomplishes tasks specified by the AI or human user.

Keywords: motion synthesis, motion graphs, parametric motion, motion planning

ACM CCS: Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism Animation

1. Introduction

Fluid, natural-looking animation is a key aspect of virtual characters and achieving it has always been a challenging task, because people can perceive even subtle errors in human motion. In interactive applications this issue is compounded by the fact that characters must be controllable, that is the character must be able to quickly and accurately adapt its movement to dynamically changing input from the AI module or, for user-controlled characters, the input system.

Producing a clip of natural-looking motion does not pose an exceptional technological challenge. Animators have a variety of production tools at their disposal for manual crafting of character animations and motion capture provides a way to record movements of live actors using tracking equipment. Once prohibitively expensive, the latter technique is nowadays dominant in animation production for big- and mid-budget projects in entertainment industries. Although these methods can yield very natural and expressive motion, they have two significant disadvantages—production cost and lack of controllability. Producing significant quantities of keyframe or recorded motion requires substantial resources in terms of manpower and expensive equipment, while inherent lack of control poses a problem for interactive applications such as computer and video games.

In offline applications all movement scenarios are predefined and animators can plan every detail of the characters’ motion in advance. In interactive applications this is impossible, because actions occur dynamically, depending on factors like user input and current state of the game world. Game engines have traditionally used motion-captured or hand-crafted animations coupled with procedural translation...
of the character to desired locations, with largely unsatisfactory results. Procedural translation of the character does not respect physical constraints of the body and environment, resulting in jarring visual artefacts such as footskating and object interpenetration.

Because of the controllability issue, procedural and physics-based animation seem like attractive alternatives to recorded and hand-crafted motion clips. These methods employ a mathematical model to generate desired character motion according to high-level parameters. Because of its parametric nature, this kind of motion is intuitively controllable and suitable for interactive applications. It does have one crippling drawback, however—lack of naturalness. The underlying mathematical model is typically only a rough approximation of actual physics that govern the movement of living creatures, while more complex models are too computationally expensive to be of any use in interactive applications. For this reason procedural and physics-based methods are still typically used only in a few specific situations, for example ragdolls are used in action games to simulate character falling and death.

The fundamental idea behind example-based or data-driven motion synthesis is to combine the controllability of procedural animation with realistic appearance of recorded clips of motion. This is accomplished by employing two classes of techniques:

1. **Motion concatenation.** Concatenation of short motion segments into sequences of actions.

2. **Parametric motion synthesis.** Parametrically controlled interpolation of similar motion clips, which correspond to the same logical action.

Research in the area of example-based motion synthesis is largely focused on automatic generation of supporting data structures for motion synthesis and on employing these data structures to synthesize controllable and visually pleasing motion at run-time. In order to generate high-quality sequences of two or more motion clips, it is important to create transitions between these clips at points where character poses are similar. These points of similarity can be identified in a preprocessing step, which has the effect of organizing example motions into searchable graph structures called motion graphs. Motion graph construction and use are discussed in Section 3.

Though motion graphs provide an efficient means of sequencing motion clips, they do little to address one of the principal drawbacks of example motions—lack of fine parametric control. Parametric motion synthesis (discussed in Section 4) aims to solve this issue by offering a means to group similar motions into parametrized motion spaces and synthesize new motions by interpolating between them.

At low level, these techniques are largely based on **motion blending**, an operation which constructs a weighted combination of two or more base motions and which we briefly introduce in the preliminary Section 2.

Motion graphs and parametrized motion spaces are very useful tools for character motion synthesis, but they are not all-powerful. In practical applications they need to be utilized for synthesis of complex motion which accomplishes demanding tasks such as locomotion in composite, dynamically changing environments or interaction between multiple characters. A potential solution is to employ **motion planning**, a set of techniques originating from robotics that use smart control policies for synthesis of long motion sequences which accurately and efficiently accomplish complex high-level tasks. An overview of these techniques is given in Section 5.

2. Preliminaries

2.1. Motion Specification

A character is animated via its skeleton. The character’s skeleton is a hierarchical structure composed of bones connected at joints. Each joint specifies a 3D transformation which is inherited by the joints that lie below it in the hierarchy. It is typically composed of position, which we represent with a 3D vector, and orientation, which we represented either with a rotation vector—3D vector representing a three-DOF (degree of freedom) rotation—or a quaternion. Most practical systems also support scale (specified as a 3D vector).

Motion may be formally defined as a continuous function $\mathbf{M}(f) = (p(f), q_1(f), \ldots, q_k(f))$, where $f$ is frame index, $p$ position of the character’s root joint at frame $f$ and $q_i$ joint orientations at frame $f$. In other words, motion is a continuous function that maps frame indexes (evenly spaced in time) to poses of the character’s skeleton.

2.2. Motion Blending

Motion blending is an operation that constructs a weighted combination of two or more base motions. The resulting motion is called a **blend**. It may be formally defined as $\mathbf{B}(t)$, where $\mathbf{B}(t)$ is constructed from $N$ motions ($\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_N$) and an $N$-dimensional weight function $\mathbf{w}(t)$. Each weight determines the influence of the corresponding example clip on the final motion. For example, if $w_1(t_0) = 1$ and $w_1(t_0) = 0$ for every $i \neq k$, then frame $\mathbf{B}(t_0)$ will be identical to the corresponding frame in $\mathbf{M}_k$. A **transition** is a blend of two motions where $w_1(t)$ smoothly changes from 1 to 0 while $w_2(t)$ smoothly changes from 0 to 1 (so that $w_1 + w_2$ always equals 1). A blend of $N$ motions where $w_i(t) = \text{const.}$ is referred to as **interpolation**.
Root joint positions are blended by computing the weighted average of root positions or velocities in base motions:

\[ p_{\text{blend}} = w_1 p_1 + w_2 p_2 + \cdots + w_n p_N. \] (1)

The preferred representation for joint orientations are unit quaternions, because they are non-singular; that is, every unit quaternion \( q \), along with its antipode \( -q \), maps to exactly one orientation (see [Lee08] for a more detailed overview of the issue). Linear interpolation (as in Equation 1) is not the correct way to average quaternions, as it interpolates along a line instead of an arc. The weighted average of two quaternions \( q_1 \) and \( q_2 \) is called spherical linear interpolation (slerp) [Sho85] and computed as

\[ q_{\text{blend}} = \text{slerp}(q_1, q_2, t) = \frac{\sin(1 - t)\theta}{\sin\theta}q_1 + \frac{\sin\theta}{\sin\theta}q_2 \] (2)

where \( \theta = \arccos(q_1 \cdot q_2) \) and \( t \) is the interpolation weight.

Slerp may be used when computing a blend of two motions, but the case of blending \( N \) motions is much less clear-cut. The most widely accepted definition of the weighted average \( q_{\text{blend}} \) of \( N \) quaternion orientations \( q_i \) is given in [BF01] as

\[ \sum_i w_i \log(q_{\text{blend}}^{-1} q_i) = (0, 0, 0) \] (3)

where \( \log \) is the logarithm map operator which maps a unit quaternion to its corresponding three-DOF rotation vector. \( \log(q_{\text{blend}}^{-1} q_i) \) represents a displacement rotation vector between orientations represented by \( q_i \) and \( q_{\text{blend}} \). Assuming that all quaternions \( q_i \) are given in the same hemisphere, the aforementioned equation has a unique solution that unfortunately must be solved for iteratively. Park et al. [PSS02] propose a blending technique where all orientations \( q_i \) are transformed into displacement vectors with respect to a reference orientation \( q_* \), like this:

\[ v_i = \log(q_*^{-1} q_i) \] (4)

where \( v_i \) are the displacement vectors. \( q_* \) is chosen so that it is as close as possible to all blended quaternions \( q_i \) and may be computed using a least-squares method. Once displacement vectors have been determined, they are blended linearly:

\[ v_{\text{blend}} = w_1 v_1 + w_2 v_2 + \cdots + w_N v_N. \] (5)

The blended quaternion is then derived by computing the exponential map of the displacement vector \( v_{\text{blend}} \) back into quaternion space and applying it to the reference quaternion:

\[ q_{\text{blend}} = q_* \exp(v_{\text{blend}}). \] (6)

### 3. Automatically Organizing Motion into Graph-like Structures

A central problem of motion synthesis is automatic generation of larger motion sequences through concatenation of individual motion clips. This is a two-fold issue:

1. Appropriate motion clips must be selected that achieve the desired goal.
2. Transitions between consecutive clips must be seamless and natural-looking.

To use a relevant example from interactive applications, a fairly standard paradigm for game AI agents is to model their behaviour with finite state machines (FSM). As game agents transition from state to state (e.g. idle to walking, or walking to running) and perform actions associated with these states, the animation system ensures seamless transitions between animation clips that correspond to these states. Transitions are constructed as smooth blends at points where the motion are similar.

Traditionally, designers would identify the possible states and state transitions of game characters and specify in detail the animations that correspond to these states. Animators would craft and record these animations according to strict specifications and transitions that were needed. By defining transitions, animators essentially connect motion clips into graph-like structures known as move trees [MBC01].

Because a typical game character can have dozens, even hundreds of different animations, the process of manually constructing move trees is quite arduous and time-consuming. Moreover, even minor changes to agent behaviour on part of designers necessitate a large amount of corrections to animation clips and move trees on part of animators. If the process of constructing move trees were automatic, it would save animators a great deal of time and effort.

Moreover, many companies nowadays have large databases of recorded motion accumulated through years of capturing motion for different projects and applications. In most cases, this motion data cannot be directly reused in a new project since it is unlikely to match the exact specifications of new motion. If these motions could be automatically organized into a searchable graph-like structure, it would be easier to reuse them even in applications and scenarios for which they were not originally created.

Most early work on generating streams of motion is focused on concatenating procedural motion [Per95, PG96, FvdPT01] and first papers on automatically organizing example motion into graphs for efficient motion synthesis appear around 2002. Arikan et al. [AF02] construct a hierarchical motion graph and employ randomized search to synthesize novel motion sequences, while Kovar et al. [KGP02] build a flat graph structure and use local search with branch and bound algorithm. Some
implementations [LCR’02, LWS02] use a two-layer statistical model, where the lower layer captures fine details of motion, while the upper layer generalizes the motion data with a probabilistic model for more intuitive control.

3.1. Building a Motion Graph

Motion graph is a directed graph where edges correspond to segments of motion, while nodes serve as choice points for connecting the segments, that is each outgoing edge is potentially the successor to any incoming edge. An edge is either a portion of one of the example clips from the motion dataset or a transition between two such clips. A node corresponds to a single frame in one of the example clips. A graph walk represents a possible motion sequence. In a well-connected graph it is possible to generate a graph walk between any two nodes. Figure 1 shows an example motion graph.

Motion graph is constructed as follows:

1. Each example motion clip is compared with every other clip (and itself) frame-by-frame using a suitable frame distance metric. For each pair of clips the comparison yields a 2D grid of frame distances (Fig. 2).
2. The frame distance grid is searched for local minima; if a minimum is below a user-specified threshold, then the corresponding frame pair constitutes a transition point (Fig. 2). On each transition point, two edges are created, representing two transitions centered on the transition point—one from the first motion to the second motion and another from the second motion to the first motion.
3. The constructed graph is pruned to ensure that it is well-connected. The aim of the pruning process is to eliminate poorly connected nodes (dead ends and sinks). Tarjan’s algorithm is used to compute strongly connected components in the graph (SCCs) and any node that is not a part of the largest SCC is eliminated.

Most work on the subject of motion graphs does not explicitly consider the problem of choosing the motions that should be included in the motion graph. Building a motion graph over an entire motion dataset is seldom a feasible approach, as it would result in a massive, inefficient graph containing lots of superfluous motion. On the other hand, manually choosing which motions to include is tedious and may result in a graph that is too small or poorly connected and therefore unsuited for motion synthesis. Zhao et al. [ZNKS09] propose a semi-automatic method for constructing a minimum-size motion graph with just enough motion to yield good motion sequences that match user expectations. First, all motion in the dataset is organized into a large motion graph. Next, the user is expected to manually select a small set of key motions that are representative of what is required by the application. Finally, an iterative algorithm is used to determine a minimum-size subgraph that contains the user-selected motions with good connectivity. Furthermore, this approach can be complemented with earlier work of Zhao and Safonova [ZS09] which aims to automatically increase connectivity in a motion graph by introducing a set of interpolated poses (actually blends of the original poses in the motion dataset) and building a well-connected motion graph (wcMG) that incorporates these poses, resulting in much better connectivity and smoother transitions than a regular motion graph.

3.1.1. Frame Distance Metric

The frame distance metric used for motion comparison is adapted from the work on video textures of Schödl et al. [SSSE00] It does not merely compute the difference between character poses at the two frames, but takes into account the following factors as well:

- Joint velocities and accelerations at both frames. Given two frames $i$ and $j$, the distance metric compares windows
of frames of fixed length, centered on frames $i$ and $j$. This way, higher-order derivatives are incorporated into distance computation.

- **Joint influences.** Not all joints affect the motion’s appearance equally. For example, rotation of the spinal joints or pelvis has more effect on the character’s pose than rotation of the wrists or fingers. The frame distance metric therefore gives different weights $w$ to different joints.

- **Position and orientation in the 2D plane.** Motion similarity is invariant to position and orientation of the character in the 2D floor plane. For example, there is no difference between two walking motions rotated by 180 degrees in the 2D plane. Frame distance metric therefore does minimization with respect to a 2D transformation $T_{\theta_{x_0}, z_0}$ which aligns the second motion to the first one.

Taking all these factors into account, frame distance computation can be formulated as

$$D = \min_{\theta_{x_0}, z_0} \sum_i w_i \| p_i - T_{\theta_{x_0}, z_0} p'_i \|^2 \quad (7)$$

where $p_i$ and $p'_i$ are joint world positions. The minimization problem can be solved analytically using the equations given in [KGP02].

### 3.1.2. Executing Transitions

Executing transitions at run-time with a motion graph is fairly simple. Transition of length $L$ is executed by blending between motions starting $(L - 1)/2$ frames before the transition point and ending $(L - 1)/2$ frames after the transition point. The second motion must first be rotated in the 2D plane to align it with the first motion, otherwise unnaturally fast changes in character orientation and position may occur during blending. Root positions are blended using linear interpolation and spherical linear interpolation can be used for joint orientations. For every transition frame $p$, root position and joint orientations are given as

$$P_{\text{root}, p} = \alpha(p)P_{\text{root}, i + p} + (1 - \alpha(p))P_{\text{root}, j - L + 1 + p} \quad (8)$$

$$q_p = \text{slerp}(q_{i + p}, q_{j - L + 1 + p}, \alpha(p)) \quad (9)$$

where $i$ and $j$ are start- and end-frame indexes, $p$ is relative frame index in the transition window ($0 \leq p < L$) and $\alpha(p)$ specifies the blend weights. A possible formulation of $\alpha(p)$ is

$$\alpha(p) = 2 \left( \frac{p + 1}{L} \right)^3 - 3 \left( \frac{p + 1}{L} \right)^2 + 1. \quad (10)$$

The simple transition scheme based on linear interpolation and slerp is suitable for short transition intervals (Kovar et al. use $L \approx 0.33s$). Other researchers propose more complex transitions schemes, often with longer or variable transition lengths. In [RGBC96, ZMCF05] and [AFO05] character dynamics are taken into account when generating transitions, whereas Wang and Bodenheimer [WB08] develop new methods for computing optimal blend weights and durations and propose a scheme for evaluating transition naturalness. Registration curves of Kovar and Gleicher [KG03] encapsulate information necessary for correct blending of $N$ motions, so they can also be employed to generate high-quality transitions between two motions. In some implementations [LCR’02, AF02, PB02, GSKJ03] displacement mapping [WP95, BW95] is applied to stitch motion together at transition points—this method does a good job preserving fine details of motion even over longer transition periods (Lee et al. use $1s \leq L \leq 2s$), though it is more computationally expensive than linear interpolation and slerp.

During the transition it is possible for one or both motions to have active physical constraints on end-effectors (e.g. left or right footplants in locomotion). Since transitions are typically short, it is sufficient to enforce constraints on the motion with greater blend weight and ignore those on the other motion. This simple scheme is used in [KGP02] and [AW01]. The analytical inverse kinematics algorithm of Kovar et al. [KSG02] can be used for computing correct end-effector configurations.

### 3.2. Synthesizing Streams of Motion

Unlike move trees, which are hand-crafted and tailored to a specific purpose, motion graphs in their basic form are unstructured, so the only way to generate streams of controllable motion is to search the graph for appropriate motion clips. The search criteria are specified by higher-level modules, namely AI and user input system, and depend on the task. The search process stops when stopping conditions are met, yielding the required motion sequence.

There are two main categories of search methods: *local search* and *global search*. Local search methods generate the motion sequence incrementally, a limited number of graph edges at a time, until the stopping conditions are met. A simple heuristic is typically used to evaluate the neighbourhood of the current edge and choose the next edges to append. On the other hand, global search methods attempt to synthesize the whole motion sequence at once. In general, local search is better suited for run-time motion synthesis than global search—not only is the latter computationally taxing, but it is also pointless to build the entire motion sequence at once when it is likely to become invalidated quickly due to dynamic nature of interactive applications.

However, local search does have a significant disadvantage—it does not generate globally optimal motion and may even fail to achieve the desired objective. Better motion can be synthesized by expanding the search horizon at the expense of performance. Some researchers attempt
to improve motion synthesis by using precomputed search results [SMM05], by having smaller, specialized graphs for different parts of the virtual environment [LCL06] or by making graphs themselves more structured [GSKJ03]. However, none of these methods are actually able to yield optimal motion sequences. To do that we must employ motion planning, discussed in detail in Section 5.

### 3.2.1. Local Search Methods

In their paper on motion graphs [KGP02], Kovar et al. propose using the branch and bound algorithm for motion synthesis. The algorithm explores the motion graph up to a specific horizon and generates multiple graph walks simultaneously. Each graph walk \( w \) has a strictly increasing error value \( f(w) \) associated with it. As the graph is explored, each edge is evaluated using a user-supplied, task-specific function \( g(w) \). By appending an edge to the graph walk, its \( f(w) \) increases. If any graph walk has worse \( f(w) \) than the current best graph walk \( w_{opt} \), it is discarded. Because it is important to compute a \( w_{opt} \) with a low value of \( f(w_{opt}) \) as quickly as possible, edges are chosen using a simple greedy heuristic. The search ends when a user-specified halting condition is satisfied. Kovar et al. demonstrate how their search algorithm can be used to generate motion sequences that have the character move along a path or reach a target location.

Lee et al. [LCR*02] expand the motion graph structure with an additional layer. They model motion as a first-order Markov process and assign higher probabilities to better-quality transitions. Then they cluster similar frames in the base motion graph into groups and for each frame construct a cluster tree, which encodes all clusters and cluster transitions that are reachable within a given search depth. Local search of such a structure is more efficient, because it works by evaluating and assembling cluster paths rather than low-level motion transitions. The selected cluster paths are mapped to most probable (i.e. best-quality) motion sequences.

Li et al. [LWS02] identify recurring patterns in example motions called textons (e.g. walk cycles) and represent them as linear dynamic systems which capture dynamic properties of the patterns. At the upper layer they represent transition probabilities between textons with transition matrices. Motion synthesis is done by specifying start and end textons of the motion sequence and computing a low-cost path between them. Smooth transitions between adjacent textons are ensured using optimization with constraints.

### 3.2.2. Global Search Methods

Global search methods such as those of Arikan and Forsyth [AF02, AFO03] are suitable for motion authoring, but of little use for online motion synthesis. In [AF02] Arikan et al. use a slightly different motion graph representation, where nodes represent example motions and edges are transitions between these motions. They create a hierarchical motion graph by first grouping together edges with same source and target nodes and then creating additional levels of the graph structure, where each level has halved cluster sizes compared to the level above it. Randomized search similar to MCMC search [Gil95] is then employed to generate the best possible motion sequence that satisfies externally specified constraints.

In [AFO03] search is not done on the motion graph, but directly on the dataset of annotated example motions. A coarse motion is first generated by randomly choosing 32-frame motion segments from the motion graph, upon which dynamic programming is iteratively applied to refine the motion.

### 3.2.3. Structured Motion Graphs

Generating motion from motion graphs in a controllable manner can be difficult due to their inherent lack of structure. Gleicher et al. [GSKJ03] attempt to address this by giving animators control over motion graph construction. In their approach motion graphs are built around user-defined or automatically identified common poses that serve as hub nodes. Displacement mapping [BW95, WP95, Gle98] is applied to ensure smooth transitions between motion clips that meet at the common poses. If there are constraints present, they are enforced using the IK technique from [KSG02]. This type of graph structure naturally partitions example motions into short clips and ensures synthesis of motion sequences with correct transitions while facilitating easier motion control due to the fact that all motion segments begin and end at familiar character poses.

### 3.2.4. Precomputing Search Results

Local search gives better results with a deeper search horizon, at the expense of performance. To enhance performance, Srinivasan et al. [SMM05] precompute search results and use them at run-time to synthesize the motion sequence. They derive a state-action model from the motion graph, where graph nodes (i.e. character poses) correspond to states (S) and edges map to actions (A) which change the character’s state. They precompute shortest possible paths between all pairs of states using Dijkstra’s shortest-path algorithm and store them into an all pairs shortest path (APSP) matrix. For each state a mobility map is then computed from the APSP matrix, holding a list of states that can be reached from that state within a fixed number of action-steps (25 in the example implementation), as well as corresponding state-action sequences that reach each state. Run-time motion synthesis then relies on repeated greedy evaluation of a cost function (provided at run-time) to choose the state-action sequences which bring the character closer to satisfying externally specified constraints.
4. Parametric Motion

Parametric motion synthesis is a method of synthesizing new motion by interpolating between motion clips that are visually similar and correspond to the same logical action. For example, if we have a walking motion and running motion, we can blend them together to create a jogging motion. The influence of each example motion is specified by its blend weight. The example motions essentially define a continuous motion space, allowing us to generate a near-infinite number of similar motions by simply specifying appropriate blend weights. We can map high-level parameters such as movement speed or locomotion curvature to blend weights and use them to control blending of the base motions—hence the term “parametric motion.” In the walking and running example, we may wish to parametrize the motion by movement speed.

Likely the first research paper on parametric motion is the “Verbs and Adverbs” paper by Rose et al. [RCB98]. They classify sets of similar motions as verbs and parametrize them by adverbs. For example, typical verbs are walk, run, sneak, etc., while adverbs may be normal, brisk, tired, etc. Verbs are connected into “verb graphs” for seamless transitioning. Verbs and adverbs enable simple parametrization for controlling motion style. In later research more complex parametrizations would be introduced, facilitating better motion control [PSS02, KG04]. Researchers have also developed real-time, example-based inverse kinematics systems [KG04, RPpSC01, ES03, GMHP04] which use parametrically controlled interpolation and are capable of generating more natural-looking motion than numeric and analytical IK solvers employed in 3D modelling software like 3ds Max and Alias Maya.

Animation systems in video game engines have featured parametric motion [Eds03] for years now, but these have largely relied on manual labour of animators to author spaces of evenly spaced, perfectly aligned animations. In this section, we present methods for automatically generating supporting data structures for parametrization and motion blending and explain how these data structures are employed at run-time. Such motion synthesis schemes are finding use in high-profile computer and video games as well, notable examples being Assassin’s Creed, Grand Theft Auto IV and Crysis. Research has confirmed that parametric motion generated in this way not only looks subjectively natural, but is close to being physically correct [SH05].

Research in the area of parametric motion has largely focused on the following issues:

- **Motion retrieval.** A motion space is defined by a set of similar example motion clips. Because motion capture databases tend to contain vast quantities of unlabeled and uncategorized motion, automated methods need to be employed for searching the database and identifying similar pieces of motion.

- **Parametrization.** Example motions comprising the motion space need to be parametrized and a scheme needs to be developed for mapping arbitrary parameter values to motion blend weights. This is non-trivial, because example motions are typically irregularly distributed in the parametric space, so some type of scattered data interpolation must be employed to obtain correct blend weights.

- **Timewarping.** Example motions need to be synchronized in time during blending, that is logically correspondent events in the example motions must occur at the same time as the motions are blended, otherwise the blend will look subpar. This is not a problem for transitions, which are typically short (about half a second or less) and centered on a point where the two motions are similar. However, with parametric motions the blend extends for the entire duration of the blended motions, so the motions must be sufficiently similar at every point of the blend. This is most obvious in our example of walking and running motion—if leg cadence in these two motions is not synchronized during blending, the resulting motion will have severe artefacts and not look natural at all.

- **Root alignment.** Root joint position and orientation in example motions need to be aligned during blending. For example, if we have two walking motions going in opposite directions, blending them directly will cause root translations to cancel each other out, resulting in an unnatural-looking blended motion where the character walks in place.

- **Constraint handling.** Example motions in general do not have the same constraints on corresponding frames, so constraints will get violated in blends. For example, in a walking motion one of the feet must always be planted to the ground, but that is not the case with running motions. A scheme needs to be developed to decide which constraints need to be enforced and when.

- **Transitions.** There is no trivial way to concatenate parametric motions into continuous streams. Motion graphs specify smooth transitions between motions at points of numeric similarity, but they are applicable only to discrete clips and not motion spaces. New types of graph structures are needed that can encode information on parametric transitions.

Proposed solutions to these issues are discussed in the subsections that follow.

4.1. Motion Retrieval

The quantity of data in motion capture databases is literally a growing issue for the animation industry. As motion accumulates over the years, it becomes impractical to manually search the database for specific motions. Instead, it is desirable to employ automated systems which efficiently and accurately search the dataset and retrieve motion clips which match the user’s specifications.
Fundamentally, motion retrieval is based on the idea that the user supplies a query motion and the search system matches motion clips in the database against the query motion and returns those which are visually and logically similar. The usefulness of such a system is obvious, as the retrieved motion clips can then be used to construct a parametrized motion space. Because it would be inefficient to search the motion data sequentially, the original motion data is usually first transformed into a more compact representation and indexed for faster retrieval. In the remainder of this section, we give a brief overview of the most popular motion data representations and indexing techniques, along with a discussion on the closely related issue of measuring motion similarity.

4.1.1. Motion Dimensionality Reduction

Motion data has high dimensionality, consisting of dozens of channels that correspond to different joints of the human skeleton. However, not all of this data is of equal significance for describing motion and most motion retrieval methods exploit this fact by transforming original motions into a low-dimensional feature space in a preprocessing step. The transformed data captures certain important features of the original motion without significant loss of fidelity, while facilitating faster, more efficient operation. In addition to that, the new representation must be intuitively indexable, which precludes the use of arbitrary compression schemes.

Motion retrieval is closely related to the more general problem of similarity-based search of time-series data, which has attracted a great deal of interest in the database community in the last decade, starting with seminal papers by Agrawal et al. [AFS93] and Faloutsos et al. [FRM94] Early time-series data retrieval systems use the Discrete Fourier Transform (DFT) to transform data into the frequency domain and retain only a handful of Fourier coefficients, thus preserving the general shape of the data, but suppressing higher-level harmonics that encode brief and sudden events such as abrupt changes or noise spikes. Chan and Fu [CF99] instead propose using the Discrete Wavelet Transform (DWT), which is supposedly better at preserving the finer details in the original data (though later research found that difference in overall matching accuracy between DFT- and DWT-based systems was ultimately negligible [WAEA00]). Some researchers use Principal Component Analysis (PCA) [KJF97, FF05, BvdPPC08] to linearly transform all data into a space of features (principal components) and cull the principal components that describe low data variance. Piecewise representations such as Piecewise Aggregate Approximation (PAA) [KCPM00, YF00] and Adaptive Piecewise Constant Approximation (APCA) [CKMP02] segment the original data into pieces of uniform (PAA) or varying length (APCA) and record the mean value for each piece. They are regarded as more intuitive to understand and easier to implement than the more exotic DFT, DWT and PCA.

4.1.2. Motion Indexing and Search

Sequential scanning of motion data is unfeasible for a large dataset, even when data has been transformed into a more compact representation. Most systems therefore employ some form of indexing to achieve reasonable search performance. Time-series data retrieval systems have generally relied on spatial access methods (SAM) for indexing. Most implementations use a variant of R-trees [Gut84], tree data structures which organize space into a hierarchy of minimum bounding rectangles (MBRs). For example, the well-known GEMINI framework of Faloutsos et al. [FRM94] employs R*-trees, which ensure better performance than traditional R-trees by reducing coverage and overlap between different MBRs. The index can then be queried with a user-specified data sequence; however, the quality of the result will depend not on the index itself, but on the transformation applied to the original data, as well as the choice of the distance metric employed for sequence matching.

Not all motion retrieval systems are based on those exact concepts and many researchers working in the field of motion retrieval have developed more domain-specific indexing techniques. For example, Liu et al. [LZWP03] use a motion index tree (based on the character’s joint hierarchy) as a classifier to determine a subset of motion data that is most likely to contain similar motions. To simplify distance computation, they use a key-frame extraction algorithm to reduce the number of frames by clustering similar frames together. However, their method works only on whole motion sequences. On the other hand, Kovar and Gleicher [KG04] first compare all motions in the dataset using the distance metric from [KGP02] and, for each motion pair, construct a match web, a data structure which encodes segments where the motions are similar. Given a query motion segment $M_q$, the match webs are used to quickly find numerically similar motion segments. However, this method does not scale well with the size of the motion dataset and, more importantly, the query motion must be one of the existing motions in the dataset (i.e. it may not be a novel motion), otherwise match webs have to be rebuilt with the new motion incorporated.

More recently Beaudoin et al. [BvdPPC08] have introduced the concept of motion-motif graphs, motion graphs in which similar motion clips are clustered together into common nodes. They are constructed as follows: (1) the whole motion dataset is transformed into reduced-dimensionality space using PCA, (2) similar poses are clustered together using $k$-means clustering and each pose cluster is assigned a letter, (3) motions are transformed into string representation, (4) frequently repeating substrings (motifs) are identified in the dataset, (5) each motif represents a cluster of similar motion sequences that becomes a node of the motion-motif graph. Such a graph can then be used as the basis of a fast motion retrieval system.
4.1.3. Motion Comparison

Initially time-series database search techniques employed simple Euclidean distance for comparison of data sequences. This is inadequate for human motion, because, as mentioned earlier in the section, motion can vary in speed and logically correspondent events are not guaranteed to occur synchronously in the compared motions. The solution is to warp the motions in time prior to comparison using dynamic time warping (DTW), so that corresponding events in the motions are synchronized. Originally used in speech recognition, dynamic timewarping was first applied to the motion comparison problem by Bruderlin et al. [BW95] and has since been employed in various motion synthesis applications, notably motion blending (see Section 4.3). Berndt and Clifford were the first to apply it to time-series database search [BC94] and it has since been used by numerous researchers for time-series data matching [YJF98, KP00, CKH02, wKPC01, KR05, FF05]. Its high computational cost has been addressed by using lower bounding [YJF98, wKPC01, KR05] and by performing timewarping on coarser (mainly piecewise linear) approximations of the original data [CKH02, KP00, PLC99]. Spurious and incorrect matches due to noise and presence of superfluous events in the compared motions can be avoided by employing longest common subsequence (LCSS), a distance measure which skips over frames that cannot be reasonably matched to any frame of the motion in time prior to comparison using dynamic time warping (DTW), so that corresponding events in the motions are synchronized. Originally used in speech recognition, dynamic timewarping was first applied to the motion comparison problem by Bruderlin et al. [BW95] and has since been employed in various motion synthesis applications, notably motion blending (see Section 4.3). Berndt and Clifford were the first to apply it to time-series database search [BC94] and it has since been used by numerous researchers for time-series data matching [YJF98, KP00, CKH02, wKPC01, KR05, FF05]. Its high computational cost has been addressed by using lower bounding [YJF98, wKPC01, KR05] and by performing timewarping on coarser (mainly piecewise linear) approximations of the original data [CKH02, KP00, PLC99]. Spurious and incorrect matches due to noise and presence of superfluous events in the compared motions can be avoided by employing longest common subsequence (LCSS), a distance measure which skips over frames that cannot be reasonably matched to any frame of the other motion [DGM97, VHGBK3], and by performing uniform scaling of motions [KPZ04] to effectively eliminate superfluous events interfering with DTW.

A drawback of most motion comparison techniques is that they measure only numerical similarity. However, two motions can be numerically dissimilar and still represent the same logical action. The motion retrieval technique based on match webs [KG04] attempts to address this by performing iterative search of the match webs, with matches from the previous iteration being used as query motions for the next iteration, until no more matches are found. Deng et al. [DGL09] attempt to further increase search accuracy by decomposing motions by body parts and extracting common motion patterns. Any motion clip may be represented as a string of pattern indices, so motion retrieval amounts to simple string matching. The method is more accurate than that of Kovar and Gleicher (see Fig. 4 in [DGL09]) and unlike the latter, it does not require index recomputation for novel query motions. A related method are the aforementioned motion-motif graphs of Beaudoin et al. [BvdPPC08], also based on string matching.

Müller et al. go the furthest in abstracting motion details by introducing the concept of relational features [MRC05], boolean features which describe geometric relationships between various body parts (e.g. whether the left foot is in front of or behind the skeleton root). Moreover, they introduce motion templates [MR06], compact matrix representations which encode how various relational features change over time for a particular class of motions, essentially capturing semantic properties of motion. Matching with motion templates is therefore an efficient and intuitive way of determining logical similarity of two motion sequences, though its accuracy diminishes when motions are very short and non-descript.

4.2. Motion Parametrization

A parametrized motion space is defined by a set of example motions with known parameter values. During animation, blend weights are computed from parameter values associated with example motions. If example motions are distributed regularly in the parametric space, then a simple interpolation scheme can be employed, such as the one used in [GR96, WH97]. In general, however, example motions are irregularly distributed in the motion space, so scattered data interpolation needs to be used. Further we give an overview of the most relevant interpolation schemes.

4.2.1. Scattered Data Interpolation with RBFs

Rose et al. [RCB98] employ a scattered data interpolation scheme based on radial basis functions (RBFs), more efficient variants of which are proposed in [SRC01, PSS02] and [RPpSC01]. Given a vector \( p \) of parameter values, the weight \( w_i(p) \) of the \( i \)th example motion is given as

\[
w_i(p) = \sum_{j=0}^{N_p} a_{ij}A_j(p) + \sum_{j=0}^{N} r_{ij}R_j(p)
\]

where \( N_p \) is the number of parameters, \( N \) the number of example motions, \( A_j(p) \) and \( a_{ij} \) linear basis functions and their coefficients, \( R_j(p) \) and \( r_{ij} \) RBFs and their coefficients. The values of basis function coefficients are initially unknown and must be computed for given \( p \).

4.2.2. k-Nearest-Neighbours Interpolation of Densely Sampled Motion Spaces

Scattered data interpolation with RBFs suffers from several issues: (1) it can be inaccurate if the motion space is sparsely or non-uniformly sampled, (2) it does not constrain blend weights to reasonable values and results in poor-looking blends for extrapolation (i.e. when input parameter values are far from sample values), (3) performance scales poorly with the number of samples.

To overcome these issues, Kovar and Gleicher [KG04] propose using k-nearest-neighbours interpolation on densely sampled motion spaces. They use the method of match webs to retrieve motion segments similar to a query motion \( M_q \). A data structure called registration curve [KG03] is then computed for these segments, encapsulating timewarping and spatial alignment information needed for correct blending.
Finally, a dense sampling of the parameter space is generated by blending between registered segments for random parameter values (see Fig. 3 for examples).

With a densely sampled parameter space (see Fig. 3 for examples), $k$-nearest-neighbours interpolation can be used to derive interpolated blend weights $\tilde{w}$ at run-time. This interpolation scheme not only ensures that $\tilde{w}$ is always projected into attainable parameter space, but is also more computationally efficient than other schemes, because it takes into account only a small subset of sample motions—the size of which is independent of the total number of samples—while all the other samples receive zero blend weights.

\subsection{Extrapolation with PCA}

As an alternative to the previous methods, PCA can be used to reduce the dimensionality of motion data and create high-level parametrizations of locomotion. In the work of Glardon et al. [GBT04b, GBT04a], example motions of $n$ subjects are expressed as a linear combination of principal components or eigenvectors of the whole motion space. From this main PCA space, $n$ level 1 sub-PCA spaces are formed for the $n$ subjects in the first PCA space, and for each type of locomotion in the sub-spaces, two new PCA level 2 sub-spaces are created, one containing the standing posture and the other with motions captured at various speeds. High-level parameters—speed, locomotion type (walking to running) and personification (combining locomotion styles of different subjects)—can then be used for motion control. Speed is mapped to the lowest PCA level and least-squares method is used to compute PCA coefficients. At the next level, linear interpolation between these coefficients is used to compute that level’s coefficients from locomotion type, and personification is applied similarly via highest-level coefficients. In addition to parametric control, this method allows extrapolation, smooth transitions and motion retargetting to arbitrary characters.

\subsection{Extrapolation with SGPLVM}

Grochow et al. [GMHP04] developed a dimensionality reduction method which employs a Scaled Gaussian Process Latent Variable Model (SGPLVM) to model the probability of motion capture poses. SGPLVM associates each pose $f_i$ from a dataset of example motions to a vector $x_i$ in a low-dimensional (usually 3D) latent space. Learning the model amounts to numeric optimization of an objective function $L_{GP}$, yielding a set of model parameters and $x_i$ values for the input poses. Moreover, each input pose is associated with a feature vector $y$, which specifies features of character poses that the learning algorithm should be sensitive to—joint angles, global orientation, velocity and acceleration.

Once the SGPLVM has been estimated, new poses can be synthesized at interactive framerates by performing optimization of an objective function $L_{IK}(x, y(f))$, which gives the likelihood of a new pose given the SGPLVM parameters and original poses. Unlike the approaches based on RBFs, the method is capable of extrapolating poses. Moreover, each SGPLVM can represent a particular motion style, and two different motion styles may be combined by linearly interpolating between their $L_{IK}$ functions. Grochow et al. have applied their approach to motion editing and authoring problems such as interactive character posing and reconstruction of incomplete motion capture data; however, the SGPLVM method is entirely applicable to motion synthesis problems in interactive applications, from IK problems (e.g. constraint enforcement) to parametric motion synthesis.

\subsection{Motion Blending}

Before example motions are blended, they must be time-warped to ensure that correspondent events in the example motions occur synchronously during blending. Similarly, they must be aligned in the 2D floor plane, otherwise root position and orientation will not be blended correctly. Finally, physical constraints specified on the example motions must be enforced where appropriate. These issues were recognized quite early by researchers: in the work of Ken Perlin [Per95] temporal synchronization and spatial constraints are applied to ensure correct blending of procedurally generated animations. The most complete solution to these issues, for example based parametric motion are registration curves of Kovar and Gleicher [KG03]. These automatically generated data structures encapsulate complete timing,
spatial alignment and constraint matching information and are nowadays employed by most implementations of parametric motion.

4.3.1. Timewarping

The issue of temporal synchronization of example motions can be resolved by timewarping these motions so that correspondent events occur at the same points in time. In earlier work [RCB98, PSS02] this was a semi-manual process—given \( N \) example motions, the user would manually mark sparse key times that correspond to significant events in the input motions (e.g. heel strokes in locomotion) and a B-spline timewarp curve \( S(u) \) would then be fitted through the frame correspondences. The timewarp curve returns a set of corresponding frame indexes \((f_1, f_2, \ldots, f_N)\) for a given generic time \( u_i \).

Bruderlin et al. [BW95] proposed an automatic algorithm to compare two motions and use dynamic timewarping to find a minimal-cost path through the frame distance grid, which serves as the timewarp curve. Kovar and Gleicher [KG03] generalize their method to an arbitrary number of example motions. They first pick a reference motion \( M_{\text{ref}} \) and create a 2D timewarp curve for each \((M_{\text{ref}}, M_i)\) pair by computing a minimal-cost path through the frame distance grid and fitting a 2D uniform quadratic B-spline through the frame correspondences. These 2D curves are then combined into the final, \( N \)-dimensional timewarp curve \( S(u) \). Kovar and Gleicher also ensure that the timewarp curve is strictly increasing (i.e. there is guaranteed one-on-one frame correspondence for blended motions) (Fig. 4).

4.3.2. Root Alignment

The issue of root alignment can be solved in an analogous manner. Kovar and Gleicher [KG03] compute aligning 2D transformations for all frame correspondences in the 2D timewarp curves and fit 3D uniform quadratic B-splines through these transformations, which yields a set of alignment curves. These are then combined into the final alignment curve \( A(u) \) for the entire motion set and used to align blended frames at run-time.

On the other hand, Park et al. [PSS02] do not precompute any alignment data. They avoid the alignment issue by not blending root transformations at all—instead, they fit blended motions to a user-specified path using a method similar to Gleicher’s motion path editing [Gle01].

4.3.3. Constraint Handling

The blending method must be able to handle conflicting constraints in example motions. This is not a serious problem for transitions, where blend weight changes monotonously from 1 to 0. The simple constraint handling technique employed in [KG02, AW01] enforces only constraints of that motion which has the greater blend weight. In parametric motions, however, blend weights can change in an unpredictable manner, often remaining equal or nearly equal for the duration of the blend, and utilizing the simple constraint handling scheme could result in sharp changes in constraints even for small changes in blend weights. Kovar and Gleicher [KG03] propose a more complex scheme that identifies a set of constraint matches \( C(u) \) by projecting all constraints to a common time frame, merging those that overlap and discarding those that are not part of any matches (see Fig. 5). A similar method was employed beforehand by Rose et al. in [RCB98], but they identified constraint matches manually.
4.3.4. Creating a Frame of Blended Motion

Kovar and Gleicher use registration curves \([\text{KG03}]\) to perform correct run-time blending. A registration curve is composed of a timewarp curve \(S(u)\), alignment curve \(A(u)\) and constraint match information \(C(u)\). Creating a single frame of the blend \(B(t_i)\) using the registration curve is a four-step process:

1. Determine the current position \(S(u_i)\) on the timewarp curve for current time \(t_i\).
2. Position and orient the frames at \(S(u_i)\) using the 2D transformations at \(A_j(u_i)\).
3. Blend the timewarped and aligned frames using the blending equations 1, 4, 5 and 6.
4. Query \(C(u)\) to determine and enforce active constraints. Because a constraint match is active at different times for different example motions, weighted averaging is used to determine the interval over which the constraint is active.

Constraint enforcement can be done using an online re-targetting method, such as those described in [SLSG01] or [KSG02]. The former method may produce popping artefacts when a limb is near full extension, which the latter avoids by allowing small changes in limb length in addition to rotation.

4.4. Parametric Transitions

Creating transitions between discrete motions is fairly straightforward—all we must do is to identify pairs of similar frames in the example motions and create transitions centered around these frames. However, when the motions in question are parametric, the number of possible transition points is infinite, so a different transition scheme is needed. Park et al. use a straightforward approach where they manually group similar motions into motion spaces and create transitions between them [PSS02, PSKS04], but that is not applicable to automatically created, densely sampled motion spaces. So far two schemes for online parametric transitions have been proposed—stitching motions together using techniques of Snap-Together Motion (STM) \([\text{GSKJ03, KS05, SO06}]\) and finding transitions between sample motions [HG07]. A more powerful technique is A* search of interpolated motion graphs \([\text{SH07}]\), but it can only be used for off-line motion synthesis.

4.4.1. Stitching Together Parametric Motions

The first step towards automatically generated parametric transitions was made by Gleicher et al. \([\text{GSKJ03}]\), who developed the concept of STM. They propose organizing motion into a semi-automatically constructed motion graph in which each node represents a pose that occurs often in the example motions. These nodes serve as familiar start- and end-points of all transitions, facilitating a higher degree of control over generated motion. Though their STM graph still uses discrete motions, motions embedded in edges lend themselves well to parametrization. This idea is expanded upon in [SO06] and [KS05]. Kwon et al. employ a similar approach to stitch together parametrized locomotion, while Shin et al. group STM edges that represent similar motion segments into parametric motions and dub the resulting graph structure a fat graph. Fat graphs can be used to generate continuous streams of parametric motion, where parametric transitions are executed in the same way as parametric motion in \([\text{KG03}]\).

4.4.2. Transitions between Sampled Motions

Fat graphs have several disadvantages: they decrease the amount of structure in parametric motion by separating motions that represent the same logical action (such as walking or punching) into different fat edges, and they constrain motion to a discrete set of common poses. In an attempt to alleviate these issues, Heck and Gleicher \([\text{HG07}]\) first build parametric motions using the automated methods of Kovar and Gleicher \([\text{KG04, KG03}]\) and then organize these motions into a parametric motion graph. Nodes of the graph correspond to parametric motions, while edges represent transitions between them (Fig. 6).

Identifying and constructing the parametric transitions necessitates locating transition points between parametric motions. Because parametric motions are continuous, transition points are continuous as well and in fact represent regions of parametric spaces. A parametric transition is generated by sampling the source and target parametrized motion spaces and then, for each source sample, computing a region (sample subset) of the target motion space to which it is possible to make a good transition Fig. 7. The number of samples can be kept reasonably low by exploiting the fact that the motion
Figure 7: Mapping samples from the motion space of source node $N_s$ to regions of the motion space of target node $N_t$. Regions are approximated by axis-aligned bounding boxes. From [HG07].

spaces are smooth, that is nearby samples represent similar motions. Source samples and target regions are encoded in the parametric edge; the latter are approximated with axis-aligned bounding boxes. Run-time parametric transitions can be executed by using the $k$-nearest-neighbours method to find the region of the target motion space that can be transitioned to from the end of the current source motion, as well as the correct target time.

AI- or user control over the character can now be achieved by using local search algorithms such as that of Kovar et al. [KGP02], though with a few additional drawbacks. An edge between two parametric motions can be generated only when every sample motion of the source motion space can transition to at least one subspace of the target motion space. Also, transitions are only possible from the end of the source motion (a necessary sacrifice to reduce the number of source samples), which can be a problem when parametric motions are long. These restrictions somewhat limit the usefulness of parametric motion graphs for character control, though they are still more structured and intuitive than fat graphs.

5. Motion Planning

Motion graphs are a powerful and flexible tool for real-time motion synthesis; they enable better motion reuse, automatic generation of natural-looking transitions and efficient synthesis of motion sequences that satisfy specific criteria. One of their principal disadvantages—constraining character motion to a limited number of valid discrete motions with respect to the current state—can be overcome by incorporating finely controllable parametric motion.

Usefulness of motion graphs and parametric motion in practical applications depends on how effectively they can be utilized by high-level application modules (AI, input system) for synthesis of motion sequences that accomplish various goals at run-time. Translating these high-level goals into synthesis of low-level motion sequences is the principal function of the character controller. Controller tasks include:

- **Locomotion.** AI- or user-directed movement of the character to a target location. Also entails avoidance of static and dynamic obstacles along the way.
- **Environment interaction.** Grasping and manipulation of objects in the environment (e.g. opening doors or picking up items).
- **Character interaction.** Interaction of two or more characters (e.g. hand-to-hand combat).

These tasks constitute the most basic mechanics of interactive applications, yet their growing complexity poses an increasing challenge for online motion synthesis techniques. In most practical implementations, example-based motion synthesis is still done by performing search of the character’s motion corpus with local information. Such methods do not at all guarantee achievement of objectives in an optimal manner and may fail altogether in more complex scenarios. To make matters worse, sometimes motion synthesis can fail not due to shortcomings of local search, but because the motion corpus itself does not contain motion clips needed to generate the required motion sequence (this is more likely to occur when using only discrete motions), thus effectively rendering a specific objective unachievable. These problems are illustrated in Figure 8, which depicts several characters failing to walk through an open gate.

In this section, we give an overview of motion planning, a class of techniques for synthesis of optimal motion sequences that achieve complex high-level goals. Motion planning techniques originate from the field of robotics and have been applied to a variety of motion tasks—locomotion (walking, running, climbing, crawling) of one or more characters [CLS03, KVD01, KNK+01, LK05, LK06, PLS03, SKG05], grasping and manipulation of environment objects [YKH04,
KKKL94], obstacle dodging [PLS03], etc. Unlike techniques that search for appropriate motion clips while using only local information, motion planning methods consider the entire relevant state space and generate motion sequences that are close to being globally optimal, that is, they are near-guaranteed to achieve objectives in the best, most expedient manner possible. As performing global motion search at runtime is unfeasible, it is necessary to perform as much of the planning computations as possible in a preprocessing step and then employ the precomputed data at runtime to make near-optimal local decisions. A well-studied approach is to unroll motion into the environment to generate a discretized representation of the state space and then compute paths through the state space at runtime (an example of this are probabilistic roadmaps or PRMs). More recently efficient and near-optimal planners have been presented which employ control policies estimated through the iterative process of reinforcement learning.

Unlike the techniques discussed in Sections 3 and 4, motion planning has only started to come to prominence and is still seldom used in practical applications. Modern video and computer games in particular are characterized by complex scenarios involving navigation of many characters through complex and dynamic environments, yet only two games are currently known to utilize a variant of planning for such tasks – BioShock and Full Spectrum Warrior.

5.1. Motion Capabilities in Complex Environments

In the introductory part of this section we considered a scenario where characters fail to navigate through an open gate and noted that such failures can sometimes occur not due to limitations of motion synthesis techniques, but because certain sections of space are inaccessible due to lack of necessary motion clips in the motion corpus. Clearly, having a large body of motion-captured actions does not guarantee that our character will be able to successfully navigate and interact in a complex environment with an arbitrary layout, which is a direct consequence of the fact that motion is seldom captured in the environment in which it is to be deployed. Preparing a motion planning controller for use in such an environment would be pointless, as it would be restricted only to the accessible portion of the environment.

These issues have motivated researchers to develop methods for verifying the navigational and interactional capabilities of the available motion corpus and for aiding designers in building a well-covered environment by annotating its parts with performable actions. During the motion synthesis process, environment annotations serve as constraints, for example when the user chooses to open a door, a motion sequence must be generated where the character walks up to the door (only walking and running motions may be used) and opens it (walking or running must be followed up by open action).

Figure 9: Evaluation of a motion graph embedded in a simple environment. Left: Coverage, indicated by brightness (brighter grey tiles are better covered; red tiles are obstacles). Right: Sample path through the environment (green), juxtaposed with the ideal path (red). From [RP07].

5.1.1. Evaluating Navigational Capabilities of Motion Graphs

Reitsma and Pollard [RP07, RP04] were the first to propose methods of systematic evaluation of motion graphs with respect to their ability to synthesize motion sequences in a specific environment. They discretize a static environment into tiles and unroll the motion graph into the environment, yielding an embedded motion graph. Node $i$ of the embedded motion graph is defined as a tuple $(x_i, z_i, \phi_i, M_i)$, where $x_i$ and $z_i$ specify the tile coordinates in the ground plane, $\phi_i$ is the orientation in the ground plane (also discretized) and $M_i$ specifies the clip which brought the character into the given ground-plane position and orientation. A transition between two nodes $i$ and $j$ exists only if there is valid transition from $M_i$ to $M_j$ which changes $(x_i, z_i, \phi_i)$ to $(x_j, z_j, \phi_j)$, and there are no obstacles along the motion path. Evaluation is then performed on the embedded motion graph, by measuring specific properties of the graph. Reitsma and Pollard define metrics for measuring the following properties:

1. Environment coverage. How many clips pass through each tile (see Fig. 9 for an example).
2. Path efficiency. Length of an actual path through the environment relative to minimal or ideal path (Fig. 9).
3. Action efficiency. Path efficiency when a specific action must be performed at the target location.
4. Local maneuverability. Responsiveness to interactive control.

Several observations can be made from the experiments of Reitsma and Pollard, principally that navigational capability degrades rapidly with environment complexity. Capability can be substantially improved by using more motion, editing existing motion or using shorter graph edges, but only to an extent—adding new motion into the dataset eventually hits a point of diminishing returns, while excessive editing degrades motion quality. It must also be noted that even though evaluation was done on discrete motions, it would be fairly simple to implement support for parametric motions.
as well, and it is reasonable to assume that these would considerably improve navigational capability.

5.1.2. Annotating the Environment with Available Actions

Given the results obtained by Reitsma and Pollard, it has been proposed that the environment should be partitioned into primitive segments before embedding. Not only would this reduce the computational and memory requirements of embedding, but it would enable the designers to build the environment from geometrically simple subregions annotated with actions that can be performed well within them. Such an approach has been adopted by Lee et al. [LCL06]. They embed motion data into motion patches, small regions derived by observing geometric regularities in the motion and target environment, each composed of one or more simple unit objects (ground panel, sloped ground panel, etc.) A separate motion graph is created for each patch, containing all actions that can be performed within the patch with good coverage. The patches then serve as building blocks for the navigable environment—they are fitted to the target environment until all reachable parts of the environment are covered, enabling the character to move about the environment at run-time.

Locomotion is followed-up by interaction with environment objects, for example walking up to a door and opening it. Object interactions are incorporated into the motion synthesis model by annotating objects with possible actions. An example of this are smart objects, seen in [KT98] and The Sims computer game [FW01]. Smart objects provide the character with an interaction plan—essentially a sequence of actions that must be performed when interacting with the object. An evolution of this approach is the concept of spatial situations, composable, reusable sets of typical actions associated with a specific location (e.g. “crosswalk” or “ticket booth”) [SGC04].

5.2. Planning with Probabilistic Roadmaps

Probabilistic roadmaps (PRMs) are a type of low-dimensional state space discretization that has often been used for motion planning. They are generated by sampling the environment for valid character figure configurations (usually represented in a low-dimensional way, e.g. with footprints or bounding cylinders), which are then connected using motion clips in the motion corpus, resulting in a traversable graph of character figure configurations covering the accessible environment. Motion synthesis is done by computing a minimum-cost path (motion sequence) through the roadmap that reaches the target location.

Motion planners which use PRMs are presented in [CLS03, SKGO5, PLS03, KVD01]. Choi et al. [CLS03] construct a probabilistic roadmap by randomly sampling the state space for footprints. They use Dijkstra’s algorithm to determine a minimum-cost path (i.e. motion sequence) to the target, while deviations from foot positions are corrected using online motion retargetting. Sung et al. [SKG05] adopt this method to crowd animation, with some modifications. They query the PRM to obtain a sequence of approximate motions which are then refined using a fast randomized search algorithm. Finally, Petté et al. [PLS03] are more focused on collision avoidance—their PRM uses lower-body bounding cylinders instead of footprints and they also incorporate a motion warping module that controls the upper body and dodges 3D obstacles such as branches and door frames.

5.3. Planning with Precomputed Search Trees

The system of Lau and Kuffner [LK06] uses motion planning with precomputed search trees for synthesis of long motion sequences even in dynamic environments containing multiple characters. They represent possible character behaviours and their transitions as a manually modelled FSM, each state mapping to specific motion [LK05]. In the precomputation phase, they build a search tree of states in the FSM and precompute a gridmap representation of the environment which they superimpose over the search tree to facilitate efficient access to tree nodes and paths (similar to the embedded motion graph of Reitsma and Pollard [RP07]). At run-time, a bitmap planner is used to search for a coarse path and its sub-paths are iteratively refined until a full path to the target has been constructed. The resulting path corresponds to a sequence of FSM states chosen from the FSM and can be converted into motion that allows the character to move to the target (Fig. 10).

5.4. Planning with Reinforcement Learning

In recent years researchers have embraced reinforcement learning as an effective technique for precomputing planning character controllers for a variety of tasks—navigation [LL04, IAF05, TLP07, LZ08, SMSH04], obstacle avoidance [IAF05, TLP07, LZ08], object grasping and manipulation [TLP07, LZ08], multiple-character interaction [SKSY08, SKYO8, LL04, GHG04], interactive character control [MP07], etc. A good introduction to reinforcement learning is given by Tadepalli and Pinto [TP07].
learning techniques and their applicability to robotics and motion planning can be found in [LaV06].

The idea of reinforcement learning is to formulate objectives that the character must accomplish as reward functions $R$, which compute rewards for being in a particular state and performing specific actions (motion clips) depending on how much these actions contribute to accomplishment of the objective (in some papers, costs are used instead of rewards—both formulations are valid, though inversely proportionate). A control policy is precomputed and used at run-time to pick actions which maximize the long-term reward, thus ensuring synthesis of an optimal motion sequence.

5.4.1. State and Transition Rewards

Character state representation is task-specific—it typically includes the identifier of the currently playing action $A$ and position of the character $(x, z, \theta)$, though it may also include the position of the nearest obstacle $(u, v)$ and any other information relevant to the task. For example:

$$s = (A, x, z, \theta, u, v).$$  

As the current action transitions to the next action $A'$, state is updated using the state transition function $f(s, A')$:

$$f(s, A') = s' = \begin{pmatrix} A' \\ x + \cos(\theta)\Delta x - \sin(\theta)\Delta z \\ z + \sin(\theta)\Delta x + \cos(\theta)\Delta z \\ \theta + \Delta \theta \\ u - \cos(\theta)\Delta x + \sin(\theta)\Delta z \\ v - \sin(\theta)\Delta x - \cos(\theta)\Delta z \end{pmatrix}$$

The reward $R(s, A')$ generally has two components—state reward $R_s$ (reward received for being in a particular state $s$) and state transition reward $R_t(s, s')$ (reward received for transitioning from state $s$ to state $s'$ using action $A'$). State reward $R_s$ is higher for states that are closer to the target state, for example it increases with proximity of the character to the target location and decreases with proximity to obstacles. Transition reward $R_t$ is higher for state transitions that bring the character closer to the objective, for example if a transition causes the character move in the direction of the target, its reward is higher.

5.4.2. Control Policy Estimation

As stated before, it is not feasible to compute the entire motion sequence at run-time. Instead, control policies $\Pi(s)$ are used to choose the next action $A'$ based on the current state $s$. A greedy control policy is one that chooses $A'$ based only on the local state and transition reward ($R_s$ and $R_t$, respectively):

$$\Pi_s(s) = \max_R(R_s(s, s') + R_t(s')).$$

The objective of reinforcement learning is to precompute a near-optimal control policy $\Pi$, that will always choose the action that yields the best possible long-term reward. This control policy has the following form:

$$\Pi_s(s) = \max_R(R_s(s, s') + \alpha V(s'))$$

where $V$ is the value function which computes an estimate of the long-term reward for transition $s \rightarrow s'$. Reinforcement learning is employed to precompute an approximation of optimal $V$. Figure 11 shows a comparison of greedy and optimal control policies, illustrating the clear superiority of the latter in maximizing the long-term reward.

Different algorithms have been proposed for estimation of $V$. When the state space is discrete, value iteration can be employed—each state $s$ is assigned an initial value $v$, which is then iteratively refined until near-optimal values are achieved. Fitted iteration is suitable for continuous state spaces—it creates a training set of sample states and iteratively improves their corresponding values. It follows that different formulations of the value function are possible:

- **State-value tables.** When the state space is small and discrete, it is feasible to use value iteration to compute the optimal value for every state and tabulate it for rapid retrieval. For example, Lee and Lee [LL04] use this approach for their boxing controller.
- **Scattered data interpolation.** Ikemoto et al. [IAF05] assume that the state space is smooth, so they only store state-value pairs $(s, v)$ for sample states with the best values. At run-time scattered data interpolation of sampled values is performed to compute the estimated optimal value.
- **Linear combination of basis functions.** Treuille et al. [TLP07] formulate the value function as a linear combination of $n$ basis functions and compute it using linear regression [dFVR03]. This representation is much
more compact than others, but requires manual choosing of appropriate basis functions and can have difficulty approximating control policies for more complex tasks.

- **Regression trees.** Lo et al. [LZ08] propose a formulation that is less compact than that of Treuille et al., but provides a better approximation for most tasks. They perform extra-trees regression [EGW05] to compute a regression tree, the leaves of which hold subsets of optimal state-value pairs.

A problem encountered in all motion planning implementations is time and memory requirements for the value function which increase exponentially with state dimensionality. Researchers attempt to reduce these requirements by exploiting the fact that only small subsets of the state space are relevant to the controller task. For example, Treuille et al. [TLPO7] take advantage of the fact that some state variables remain unchanged during transitions (e.g. distance between the character’s path and a static obstacle) and compute the value function separately along these dimensions. They are able to switch between value functions at any time and blend them along separable dimensions (e.g. value functions for separate static obstacle positions can be linearly blended to produce values for any obstacle position).

### 5.4.3. Motion Synthesis

Once an optimal control policy \( \Pi \), has been estimated, motion synthesis is straightforward. As the current action finishes, the control policy is used to pick the next action \( A' \) based on the current state \( s \). The next state \( s' \) is computed using the state transition function \( f(s, A') \). Motion blending and transitions can be implemented in different ways:

- **Motion graphs.** In [LL04] and [IAF05] a discrete motion graph is employed. This approach is not suitable when responsiveness is critical (e.g. interactive character control), as the character’s movement is constrained by the graph. McCann et al. [MP07] attempt to make the character more responsive by using an unpruned motion graph that contains only short motion segments, and by allowing even low-quality transitions when better ones are unavailable.

- **Motion fragments.** Treuille et al. [TLPO7] use a step-based approach with implicit transitions. They divide example motions into fragments, each holding a sequence of frames that represents a single walk cycle (Fig. 13). Transitioning between motion fragments is done without a motion graph or constraint enforcement—the blending algorithm simply aligns the fragments in the middle of their ground contact phases and blends with the ground contact foot being treated as the root of the skeleton, which guarantees that there will be no footskates except in rare cases of motions where both feet are in contact with the ground.

- **Parametrized motion fragments.** Lo et al. [LZ08] extend the motion planning framework with support for parametric motions. They group motion fragments with similar rewards together into clusters based on the criteria of numerical similarity (according to the metric of Kovar and Gleicher [KG04]) and similarity in values of state parameters. Registration curves [KG03] are then constructed for blending. Transition rewards \( R_t \) between clusters are computed as averages of transition rewards for each pair of clips in the clusters. The transition function is reformulated to accommodate parametric motions and includes blend weights; on each transition, the optimal blend weights are computed by uniformly sampling the blend weight space and choosing the sample with the best reward value.

Planning character controllers based on reinforcement learning have been demonstrated to synthesize higher-quality motion sequences than controllers using greedy search. Moreover, they are usually able to generate a good motion sequence even in scenarios where greedy controllers fail altogether. Planning controllers that use discrete motions can still have problems with tasks that require great control precision, such as navigation between densely placed obstacles, because they must choose from a limited selection of discrete motions even when the optimal motion would lie somewhere in between. Lo et al. [LZ08] attempt to address this issue by adding support for parametric motions. Figure 12 illustrates performance of different classes of character controllers for a simple navigation and grasping task.

### 5.5. Multiple-Character Interaction

Simulating interactions between two or more characters is currently a rather active research topic in character animation. In the last 5 years, several groups of researchers have presented their work on the subject, from which three notable trends can be observed. Firstly, most work appears to focus on competitive interactions (specifically hand-to-hand and melee combat)—no doubt due to great mainstream popularity of action games—though some interesting non-competitive interactions have been explored as well (for example, Shum et al. [SKSY08] simulate two characters carrying a heavy box together). Secondly, presented systems predominantly use example-based motion synthesis with motion planning to generate interactions. Finally, here too is motion planning with reinforcement learning proving itself to be the technique that offers the best combination of motion quality and computational efficiency. Though researchers have proposed hybrid approaches that combine motion-captured actions with physics-based optimization with spacetime constraints to synthesize two-character interactions [LHP06], these methods are computationally demanding and not suitable for online use.

© 2010 The Authors
5.5.1. Reinforcement Learning for Planning Two-Character Interactions

Lee and Lee [LL04] used reinforcement learning to train a boxing controller. Their character is able to approach the opponent and attack, but as the controller does not account for the opponent’s actions, it is not able to react defensively. Other researchers [GHG04, SKY08, SKY07] work with an expanded state space which accounts for both characters’ actions in conjunction. Interactions must first be captured either by recording a single individual performing all interactions (as done by Shum et al.) or by capturing two interacting individuals simultaneously (more difficult due to limitations of motion capture equipment). Individual actions must be extracted from the resulting motions streams and classified into actions, which can be done manually or automatically [SKY07, KCPS08]. Finally, a motion graph is constructed from the classified actions.

Choosing a suitable state representation that incorporates actions of both characters is important for planning interactions. For example, Shum et al. use the representation $s_t = (r, \phi_1, \phi_2, Next(A_1), A_2)$, where $s_t$ is the state of character 1, $r$ distance between characters, $\phi_1$ and $\phi_2$ relative facing angles of the characters, $A_1$ and $A_2$ current actions of the characters and $Next(A_1)$ the set of actions that can follow $A_1$. When taking state space samples, two distinct approaches can be adopted. “On-policy” method takes samples selectively, giving preference to states that yield better reward values. This approach is employed by Graepel et al. [GHG04] and for most locomotion planners (Section 5.4).

The less-common “off-policy” method is used in [SKY08, SKY07]. It is independent of the reward function and explores the state space with only two constraints—high state connectivity and high interaction density—which means the reward function can be modified without having to redo the whole training process. For example, the martial arts controller of Shum et al. [SKY08] can switch between fighting styles at run-time by adjusting the current reward function. Moreover, Shum et al. optimize the state space search tree by grouping similar states together, yielding an interaction graph (Fig. 14).

Reward values for search tree or interaction graph edges can be precomputed using dynamic programming, but that approach is not so well suited for competitive interactions, as the goal is not only to maximize the character’s reward, but also minimize the opponent’s reward. Shum et al. therefore use min–max search for their martial arts controller [SKY08].

When changing the fighting style at run-time, min–max search must be repeated using the new reward function—in that case, on-line performance is preserved by performing the search as a background process and by using reasonable search depth (3–5).

5.5.2. Statistical Modelling of Two-Character Interactions

An alternative approach that achieves interactive performance is proposed in [KCPS08] and uses Bayesian networks for modelling competitive two-character interactions. Once motions have been recorded, segmented and classified
(multiclass support vector machine classifiers are employed),
a coupled motion graph is constructed, where edges are either
transitions between actions on the same character or “cross-
edges,” which denote correlation between actions on the two
characters. For example, if action $A_{i+1}^1$ on character 1 occurs
immediately before action $A_j^2$ on character 2 in the captured
motion stream, a cross-edge is constructed between the two.
Obviously, this suggests that motion capture should be done
simultaneously on two actors performing the interactions.

It follows from the above that each action $A_{i+1}^1$ on char-
acter 1 is dependent on two actions—$A_j^1$ (i.e. the preceding
action on the same character) and $A_{i+1}^2$ (the preceding action
on the other character). These simple causal relationships
serve as building blocks for a Bayesian network, which re-
resents interactions between the two characters. Conditional
probabilities for nodes in the Bayesian network are estimated
by using the captured example motion stream as a training
set. Moreover, user control over a character can also be im-
plemented, by incorporating user control signals $c_j$ into the
network.

5.5.3. Response Motions

When high-impulse contact occurs during character interac-
tion (e.g. a character gets hit during combat), it is expected
that the character reacts appropriately to the impact. This
is achieved by executing a response motion that depicts the
character recoiling from the impact and possibly falling down
or struggling to maintain balance.

Response motions are typically implemented using
physics-based methods or combinations of example motions
and physics-based approaches. One of the most popular tech-
niques applies physical collision dynamics to simulate re-
action of the character to impact, upon which a balancing
controller based on proportional derivative (PD) servos is
employed to restore the character to balance and blend into
an appropriate example motion [ZH02, ZMCF05]. Arikan
et al. [AFO05] use motion-captured response motions which
are interactively deformed to make them appropriate to im-
 pact force and direction. Machine learning is employed to
ensure visual quality of synthesized motions. Zordan et al.
[ZMM’07] use support vector machines to select appropriate
response motions at run-time. Abe et al. [AdSP07] perform
frame-based local optimization that requires solving of a
quadratic program (QP). Finally, Liu et al. [LHP06] pose
synthesis of response motion as an optimization problem
with spacetime constraints, though their approach is still not
suitable for real-time applications.

5.5.4. Interaction Patches

We have so far discussed methods for synthesizing interac-
tions between two characters, but sometimes it is necessary
to model large-scale interactions involving many characters
in a complex scene, such as massive battles often seen in film.
Shum et al. [SKSY08] introduce the concept of interaction
patches which serve as building blocks for such complex
interactions.

Before interaction patches are constructed, the user first
defines interaction patterns for a pair of characters which
specify short interaction sequences that should occur in the
final scene (e.g. character 1 : kick → character 2 : dodge).
State space is then explored like in [SKY07], except now
interaction patterns are used as constraints to eliminate un-
desired state sequences. Interaction sequences encoded in
the search tree are evaluated with a cost function that incor-
porates criteria such as relative distance and facing of the
characters, whether there is contact between the characters
and how well their actions are synchronized. Finally, the best
sequences are saved as interaction patches.

For each pair of interaction patches it is verified if they can
be concatenated temporally (i.e. sequenced) or spatially (i.e.
if more than two characters can interact concurrently). User
can now manually author a scene involving many charac-
ters by selecting and concatenating interaction patches (see
Fig. 15 for an example). Since the method achieves inter-
active performance, it could conceivably be employed for
procedural generation of such scenes in an interactive ap-
lication, for example large-scale combat scenes in action
games.

6. Discussion

Example-based motion synthesis methods are currently by
far the dominant means of generating character motion in in-
teractive applications such as video games and they are likely
to retain that dominance in the years to come. The naturalness
and expressiveness of motion-capture or hand-crafted animation is in most cases vastly superior to that generated by procedural and physics-based techniques. Automated methods for organizing motion into graph structures and parametrized motion spaces reduce the amount of manual labour needed to achieve synthesis of quality motion. However, some issues have still not been fully addressed and there is substantial room for further development in various respects.

For example, most existing character animation systems are reliant on greedy search of motion data to synthesize motion sequences that accomplish high-level objectives, which are often not reliable enough for tasks such as navigation, interaction and interactive character control. Because character behaviour is animation-driven, limitations of motion synthesis methods effectively impair the capabilities of the AI, which has been repeatedly observed in practical implementations [Cha09]. These problems can be exacerbated by the fact that motion graphs constrain motion synthesis only to transitions deemed valid by the construction algorithm, even though in speed-critical scenarios—such as interactive character control in action games—lower-quality transitions would be an acceptable trade-off for better character responsiveness.

Motion planning with reinforcement learning has been proven to yield better motion sequences than search-based methods and appears to be an effective solution to the above issues. Its main drawback used to be that it supported only discrete motions, but recently an effective parametric motion planning controller was presented in [LZ08]. Moreover, recent planning implementations have shown that better motion control can be achieved by using short, implicitly transitionable motion segments [TLP07] instead of a regular motion graph, or by including a model of transition quality control and using it to make acceptable quality trade-offs when necessary [MP07]. Unfortunately, time and memory requirements of planning controllers remain quite high due to character state dimensionality that increases with complexity of tasks and environments. It can be assumed that a central issue in future research will be development of new, more efficient control policy approximations which will exploit the fact that only a subset of the state space includes meaningful actions.

One important limitation of example-based motion synthesis is that quality and controllability of synthesized motion is heavily dependent on the size and quality of the example motion corpus. Short of repeating motion capture for missing or poor motion, the only way to improve an existing motion dataset is to apply motion editing techniques to example clips, though only a limited amount of editing can be done before noticeable degradation of quality occurs [RP07]. The solution may be to couple the animation production process (i.e. motion capture sessions) with generation of data structures for motion synthesis to ensure an optimal initial motion dataset. For example, the system proposed in [CHP07] integrates motion capture with character controller learning—it automatically detects inputs that the controller cannot handle well and prompts the actor to perform the corresponding motions. Not only does this method result in a more capable controller, but it also substantially reduces the number of samples needed to achieve good motion synthesis. Given the high cost of motion capture and limitations of motion editing, such systems are certain to remain an important research topic.

References


[GR96] Guo S., RobertJ.: A high-level control mechanism for human locomotion based on parametric frame space


Computer Animation (Aire-la-Ville, Switzerland, Switzerland, 2004), Eurographics Association, pp. 89–98.


© 2010 The Authors
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.


