Motion Fields for Interactive Motion Synthesis

DUARTE DAVID, University of Saarland

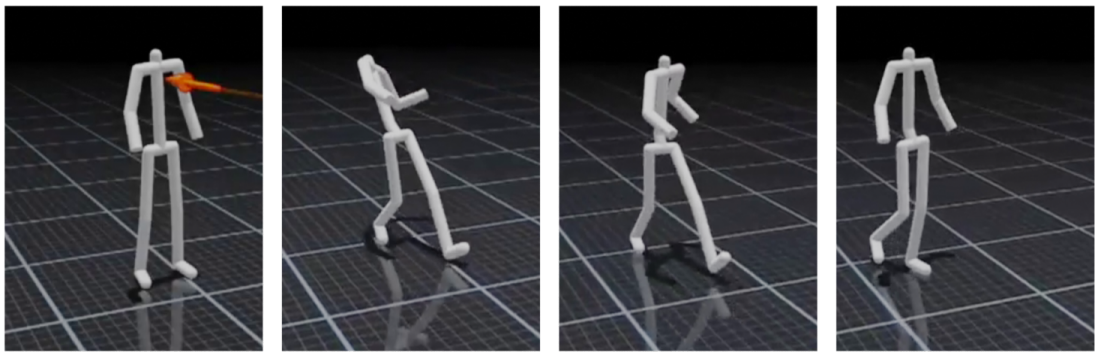

Fig. 1. The Motion Fields algorithm allows dynamic perturbation to affect the animation, with a realistic looking recovery animation being generated from motion capture data of a push, as well as from the "interpolation" done by the algorithm. See video at https://grail.cs.washington.edu/projects/motion-fields/ [Lee et al. 2010]

We present the motion synthesis algorithm based on **motion fields**. We compare it with other landmark interactive data-driven motion synthesis algorithms (**Motion Graphs**, **Motion Matching**, and **Phase-functioned Neural Networks**) by presenting the key concepts behind these algorithms, and by analyzing their effectiveness at solving the problem at hand, computational cost, and ease of implementation and use, as well as their impact on past and current research. We conclude on the importance of **motion fields** and briefly look at the future of interactive data-driven motion synthesis.

Additional Key Words and Phrases: Motion Synthesis, Motion Fields, Motion Matching,Motion Graphs, Phase-functioned Neural Networks, Markov Decision Model, Reinforcement Learning

1 INTRODUCTION

The purpose of the this report is to complement the presentation given on *Motion Fields for Interactive Character Animation* [Lee et al. 2010] for the seminar Motion Synthesis for Virtual Characters. With this report, we aim to frame it against other interactive, data-driven, motion synthesis algorithms, some of which preceded it, and some others which succeeded it. In doing so, we aim towards forming a global picture in the history of algorithms that attempt to solve this task, and evaluate the importance of motion fields in current and future research on the topic of interactive, data-driven, motion synthesis.

One question which should be addressed is: why do we care about interactive motion synthesis? One of the most obvious use cases comes from the video game industry, where often a user controlled avatar must move realistically based on input from the user, and should react to the surrounding environment. This makes motion synthesis harder, as we have two more constraints: each frame should be computed at interactive rates and the motion path (or objective function) isn't known for the whole animation.

We start by presenting the algorithm for synthesizing motion with *motion fields* [Lee et al. 2010] in **Section 2**. In **Section 3**, we discuss related work and briefly summarize three other landmark algorithms: *Motion Graphs* [Kovar et al. 2002], *Motion Matching* [Büttner 2015; Clavet 2016], and *Phase-functioned Neural Networks (PFNN's)* [Holden et al. 2017]. In **Section 4**, we discuss the advantages and shortcomings of *motion fields*, and how it improves upon previous work and how more recent work builds on it, using the algorithms cited above as a comparison. **Section 5** concludes by summarizing the key concepts and ideas from the previous discussion.

2 MOTION FIELDS

2.1 Motion Space

One of the key concepts of this algorithm is that of a **motion space**. The **motion space** is a concept identical to that of *state space* and *phase space* in physics. A point in the *motion space* is a tuple with the pose (root position and bone orientations) and velocities (variation of root position and variation of bone orientations) of the character. One can define addition (+) and subtraction (−) as a component-wise addition and subtraction. These operations are defined element-wise, in such a way that the addition of two elements $(e + e')$ corresponds to performing both transforms in succession $(e'e)$ and subtraction $(e - e')$ corresponds to doing the inverse transform of the second element after the first one $(e'^{-1}e)$.

The distance in the *motion space* is defined as:

$$d(m, m') = \sqrt{\begin{aligned} &\beta_{root}||v_{root} - v'_{root}||^2 + \\ &\beta_0||q_0(\hat{u}) - q'_0(\hat{u})||^2 + \\ &\sum_{i=1}^{N} \beta_i||p_i(\hat{u}) - p'_i(\hat{u})|| + \\ &\sum_{i=1}^{N} \beta_i||(q_i p_i)(\hat{u}) - (q'_i p'_i)(\hat{u})|| \end{aligned}}$$

In this equation, m and m' are the states in the *motion space*, v_{root} is the velocity of the root bone, p_i is the local rotation for bone i and 1_i is the local rotation delta for bone i. The weights β are parameters. In the original paper, $\beta_{root} = 0.5$,$\beta_0 = 0.5$ and β_i is the length of bone i. $\hat{u}$ is an arbitrary unit vector.

2.2 Motion Synthesis

For any given state, there are a set of actions which can be applied to it in order to proceed to the next state. This set of actions, $A(m)$, is called the **motion field** evaluated at a state m, which is a convex combination of k actions a_i, each based on one of k nearest neighbours, m_i.

The only thing necessary for motion synthesis is a function which, when applied to the current state (i.e.: the current frame of animation), returns the next state. In order to do so, a variable y, which is the velocity evaluated at the next frame in the original animation data, is stored for every state m.

This function is called the *integration function* and defined as:

$$I(m,a) = m' = (x + v', y')$$
$$v' = (1-\delta)\sum_{i=1}^{k}(a_i v_i) + \delta((\bar{x} + \bar{v}) - x)$$
$$y' = (1-\delta)\sum_{i=1}^{k}(a_i y_i) + \delta\bar{y}$$

In the above equation, $\bar{m} = (\bar{x}, \bar{v})$ is the nearest neighbor of m. The parameter $\delta \in [0,1]$ is a smoothing parameter that "pulls" the new state towards the nearest neighbor.

2.3 Choosing an action

To simplify things, the set of possible actions is restricted to k actions, one for each nearest neighbor, where the *n-th* action is defined by:

$$a^n = \frac{(w_1, ..., w_{n-1}, 1, w_{n+1}, ..., w_k)}{(\sum_{i=1, i\neq n}^{k} w_i) + 1}$$

where $\mathbf{w}$ are the so called *similarity weights*, defined as $w_i = \frac{1}{\eta \cdot d(m, m_i)}$, and η is the normalization factor such that their sum is one.

To choose the action, the problem is modelled as a *Markov Decision Process* and solved using *Reinforcement Learning*.For that purpose,a new state space is defined, where each state $s = (m, \theta_t)$ (θ_t is the vector of *task parameters*, which gives information on how well a task is being performed).

A new transition function is also defined as:

$$s_{t+1} = I_s(s_t, a_t)$$
$$I_s(s,a) = (m', \theta_t') = (I(m,a), \theta(m'))$$

A reward function, $R(s,a)$, that defines how well the task is being performed is also necessary, but its definition depends on the problem: for following a direction, one can simply return the absolute value of the difference between the intended angle and the angle the character is facing, scaled.

The objective is to find a deterministic policy (i.e.: a mapping from states to actions) that maximizes a value function $V(s_0) = \max_{a_t} \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$, where $\gamma \in [0,1)$ is the so called *discount factor*.

Maximizing an infinite sum may appear complicated at first, but the following iterative algorithm can be used (*Value Iteration*):

Do:

- $\pi(s) = argmax_{a \in A(m)}[R(s,a) + \gamma V(I_s(s,a))]$
- $V(s) = R(s, \pi(s)) + \gamma V(I_s(s, \pi(s)))$

while V **changes.**

The value of V is stored only for a finite amount of points (obtained from the cartesian product of motion space states in our data with a regular sampling of the space of θ_t). When its value is needed for other points, we compute it by interpolation (using the *similarity weights* in the motion space dimensions, and using multilinear interpolation in the dimensions of θ_t).

At runtime, the action that maximizes $R(s, a^i) + \gamma V(I_s(s, a^i))$ is chosen.

2.4 Other considerations

The problem with the previous scheme is that the amount of discrete points that need to be considered when doing *value iteration* and values of the *value function* that need to be stored grows exponentially with the dimensions of the task parameters, increasing the offline computation load and the runtime memory requirements. To ease this problem, it was noticed that the value function varies smoothly in time, and therefore computing it only for every $k-th$ frame (yielding a compression ratio of $1/k$) and performing temporal interpolation for the other frames yields decent results, for small k. The problem is that the "reaction time" of the motion synthesis (i.e.: the time it takes for the reward function to get closer to zero) increases. Our character loses flexibility.

One of the advantages of this algorithm, as we'll see, is that the starting point doesn't have to be part of the data.

One thing we did not consider here are small corrections, such as using *inverse kinematics* to prevent foot-skating. We refer to the original paper for such considerations.

3 RELATED WORK

3.1 Prior Work

3.1.1 Motion Graphs. A *Motion Graph* is a graph where each edge represents a clip of animation. Given a task, a graph search is done in such a way that it maximizes the reward set by that task, thus playing the appropriate animation made up of various clips of animation. The main contribution of this paper are the heuristics on how to split and create edges in such a way that the proper motion can be generated. To do so, we assign a mesh to the animated skeleton. For each clip, we create a point cloud, where each point is a vertex of the mesh at each k-frames before the end of the second clip/after the start of the first, where k is the length of the transition edge. The square root of the sum of squared differences of each corresponding point in the point cloud is used as a metric for compatibility between two clips in order to create an edge from the beginning of one to the end of the other.

3.2 Later Work

3.2.1 Motion Matching. Motion Matching heavily builds on Motion Fields, but discards most of the complicated ideas in favour of simpler ones. The key point is that the value function computation is too costly (both the offline computation cost, which doesn't allow for quick iteration of fine-tuning the choice of data given, as well

as the runtime memory costs), and as such it should be avoided at all costs. Instead of saving a dense motion state, only the endpoints and/or velocities of some key bones are stored (without the root transform), as well as future trajectory information (a sparse point cloud of sorts, similar to what was used in *Motion Graphs*, but sparser and potentially with some velocity information as well). Then, the cost function is defined as a p-norm of a vector between the current point (current pose/velocity and desired trajectory) and another point (pose/velocity/trajectory) in the data.

The next frame of animation is then simply the frame in the data that minimizes this cost.

Since each of these dimensions can be arbitrarily scaled (to give more importance to some things, like specific bones, smoothness of animation or responsiveness (decrease/increase the scale of the trajectory dimensions, respectively), and one is free to choose which features to use, there is a lot of flexibility in the cost function. However, since our cost function is a p-norm, we can use a *kd-tree* structure to efficiently choose the nearest neighbor.

3.2.2 Phase-functioned Neural Networks. One of many Neural Network based approaches to motion synthesis, this was the first to be able to perform interactive motion synthesis well, and on a variety of terrains. Previously, methods based on *Convolutional Neural Networks* [Holden et al. 2016] and *Recurrent Neural Networks* [Fragkiadaki et al. 2015] had been tried. The problem with *CNN* based approaches is that they require the whole input sequence to be provided at once, making them not very useful for interactive motion synthesis. *RNN's*, on the other hand, accumulate error over time, which makes the motion they produce unusable unless some sort of smoothing (such as interpolation with the nearest neighbor) is used. Doing this, however, requires a nearest neighbours search, which makes the algorithm perform worse.

The big contribution of *Phase-functioned neural networks* is the observation that human locomotion tends to be cyclical, and, as such, this information can be provided to the neural network to help predict the result of the next step. This algorithm also provides the surrounding terrain as input to the neural network, which makes the animation predicted by the neural network react to its surrounding.

4 DISCUSSION

4.1 Results

The main problem with the results obtained with *Motion Graphs* is their responsiveness (or lack thereof). Because we can only transition at the end of an edge, each clip has a predefined length, which means that everytime the objective changes, it takes time for the current clip to finish until we start walking on an edge which matches (at least partially) our goal. However, having shorter edges means that the graph is more connected, which increases the search time. On the other hand, arbitrarily complicated goals can be defined (as long as their computation is not too costly), which makes this algorithm better suited for more complicated actions or sequences of actions.

Motion Fields suffer less from this problem (as they are, in fact, quite responsive). The generalization of space of transitions (i.e.: the motion fields) means that the algorithm can start from any pose/velocity. However, obviously, starting states that are well outside the given data will not produce good results, as is the case for all data-driven algorithms.

Due to the nature of the algorithm, we can at any point blend to a different motion synthesis algorithm and then blend back (for example, simple dynamics simulation, to simulate a fall, and then we blend back with a realistic recovery animation, if there is such data - and reasonable results even if there is not). The responsiveness of characters animated with this algorithm is what made it so important in the development of subsequent methods.

Motion Matching does not produce frames outside of the data. Moreover, since the similarity cost function only considers some attributes (and not the entire skeleton), it is possible that we get two consecutive generated frames have entirely different poses. Nevertheless, this is not necessarily a problem. The first problem can be advantageous: if the data is expressive enough, it contains a frame similar enough to the one we want to transition to, and we never risk having unrealistic poses. The second problem is heavily mitigated by the flexibility of the model, as one can choose attributes in such a way that the problem never happens in the data (for typical human motion, just a few points are enough to distinguish uniquely between the very different poses of animation). Moreover, if "fed" the right data, it produces better results [Zadziuk 2016].It can also be as responsive as one wants it to be (as the animation can "jump" to any other frame, depending on the cost function), unlike *Motion Fields*, which can only approach the closest frames in the motion space.

PFNN's produce the highest quality results, both in terms of responsiveness and believability. Moreover, their design allows them to be suitable for various environments, which is impossible to do in *Motion Graphs* (as there is no fine-grained control over the animations) and very hard with *Motion Fields*, as it would be very difficult to find the proper reward function without increasing the number of task parameters. Due to their flexibility, it might be possible to incorporate some simple terrain information into *Motion Matching*, but even then, since we are only allowed to use frames from the data, even if our terrain was very simple we would still need to do some post-processing (for instance, setting the foot positions with *inverse kinematics*).

4.2 Computational Cost and Scalability

One of worst problems of *Motion Fields* are the associated computational costs. Computing the *value function* takes a long time and it might take a lot of space. This means that choosing which data to feed the algorithm cannot be done easily (as we have to compute the value function each time), and we have runtime costs associated with the size of the *value function*. *Motion Graphs* have similar computational costs when building the motion graph, but the size of the graph itself is small enough that it allows for better runtime performance. *Motion Matching*, on the other hand, has minimal offline costs, as it only requires an acceleration structure to be built for the nearest-neighbor search (usually a *kd-tree*), which can be done quite efficiently. This allows for quick iterations with different kinds of data to get the best results. *PFFN's*, on the other hand, have the worse computational costs in the precomputation step, as it needs

to not only train a neural network, but also fit terrain data to the animation data.

At runtime, however, the situation changes. *Motion Graphs* and *PFNN's* are very fast and as such can be used to animate multiple characters. Both *motion graphs* and *motion matching*, however, have the bottleneck of having to compute nearest neighbors, which makes them quite inefficient in comparison. Moreover, this computation scales very poorly with the number of dimensions, which means that we have restrictions on how well we represent each state. This is particularly important for *Motion Matching*, given that, for each state, it needs to represent trajectory information.

4.3 Ease of use and implementation

Methods for editing graphs, both automated and manual, are well known. This makes *motion graphs* particularly appealing, as they can be fine-tuned to yield the best results for a particular problem. However, with all the other methods, one can simply "feed" them the data and observe the results, while with *motion graphs*, it is necessary to at least select a threshold for creating edges. Nevertheless, it should be noted that the high space dimensional space in which the other methods rely can be somewhat opaque, particularly with *PFNN's*

One of the biggest advantages of *Motion Matching* is its ease of implementation and its flexibility, in particular when compared to *Motion Fields*. The simplification of the algorithm means that all that is needed is an efficient nearest-neighbor search and correct trajectory prediction. It has a lot of parameters to fine-tune (which data to use, what joint positions to consider, what cost function to use, among others) but the fast precomputation time allows the user to quickly iterate between different variables. *Motion Fields*, on the other hand, requires more concepts to be implemented, and it doesn't have as much flexibility when being used. *PFNN's* are the more sophisticated algorithm and thus the hardest to implement and use (as the training and terrain fitting takes a long time). Still, the results are well worth it, as evidenced, for instance, by their growing popularity in the video game industry.

4.4 Future Work

None of the algorithms mentioned in this report have taken advantage of the fact that the motion we are trying to synthesize often obeys physical constraints. Simulation based approaches already exist, but so far the results have either been limited to 2D locomotion [Peng et al. 2016], or to learning small clips of animation (in order to do motion retargeting) [Peng et al. 2018]. Moreover, we only considered algorithms which require a good amount of animation data (*Motion Matching* and *PFNN's* used over one hour of animation data), which needs to be motion captured (as it is very costly to hand animate such data, and, one could argue, somewhat disheartening for the animators who have to animate such mundane actions). This means that we cannot synthesize stylized/non-realistic motion. One of the advantages of [Peng et al. 2018] is that it requires only a small amount of data, which can be hand animated.

Moreover, algorithms which learn locomotion behaviours from the ground up, like [Heess et al. 2017] were also not considered in this report.

Future work would involve exploring these algorithms and see how they can be applied in the context of interactive motion synthesis, in order to either improve current methods or apply them with less amount of data (in order to be able to synthesize motion for hand animated characters). One possible way to do the latter is try to transfer "style" to motion generated with one of the methods discussed here, similar to [Holden et al. 2017].

5 CONCLUSIONS

As we have seen, all of these algorithms build on top of one another, and each present landmark findings that make them essential to the understanding of the field of Motion Synthesis. *Motion Fields* present an interesting way to formalize the problem, even if there are simpler methods, like *Motion Matching*, or more scalable methods with better results, like *PFNN's*. However, the concepts introduced by *Motion Fields* are still important to the understanding of the field of motion synthesis, as they can provide some insight into the problem. Further work still needs to be done to improve interactive motion synthesis, in particular for hand-animated characters.

REFERENCES

Michael Büttner. 2015. Motion Matching - The Road to Next Gen Animation. nucl.ai 2015. (2015).

Simon Clavet. 2016. Motion Matching and The Road to Next-Gen Animation. GDC 2016, https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road. (2016).

K. Fragkiadaki, S. Levine, P. Felsen, and J. Malik. 2015. Recurrent Network Models for Human Dynamics. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 4346–4354. https://doi.org/10.1109/ICCV.2015.494

Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. 2017. Emergence of Locomotion Behaviours in Rich Environments. *CoRR* abs/1707.02286 (2017). arXiv:1707.02286 http://arxiv.org/abs/1707.02286

D. Holden, I. Habibie, I. Kusajima, and T. Komura. 2017. Fast Neural Style Transfer for Motion Data. *IEEE Computer Graphics and Applications* 37, 4 (2017), 42–49. https://doi.org/10.1109/MCG.2017.3271464

Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned Neural Networks for Character Control. *ACM Trans. Graph.* 36, 4, Article 42 (July 2017), 13 pages. https://doi.org/10.1145/3072959.3073663

Daniel Holden, Jun Saito, and Taku Komura. 2016. A Deep Learning Framework for Character Motion Synthesis and Editing. *ACM Trans. Graph.* 35, 4, Article 138 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925975

Lucas Kovar, Michael Gleicher, and Frédéric Pighin. 2002. Motion Graphs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*. ACM, New York, NY, USA, 473–482. https://doi.org/10.1145/566570.566605

Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. 2010. Motion Fields for Interactive Character Locomotion. *ACM Trans. Graph.* 29, 6, Article 138 (Dec. 2010), 8 pages. https://doi.org/10.1145/1882261.1866160

Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. 2018. DeepMimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills. *ACM Trans. Graph.* 37, 4, Article 143 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201311

Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2016. Terrain-adaptive Locomotion Skills Using Deep Reinforcement Learning. *ACM Trans. Graph.* 35, 4, Article 81 (July 2016), 12 pages. https://doi.org/10.1145/2897824.2925881

Kristjan Zadziuk. 2016. Animation Bootcamp: Motion Matching: The Future of Games Animation...Today. GDC 2016, https://www.gdcvault.com/play/1023478/Animation-Bootcamp-Motion-Matching-The. (2016).