Animation Database for real-time motion reconstruction

Bachelor project, EPFL

Simon Richert

Supervisors : Sofien Bouaziz, Thibaut Weise

June, 2012

Abstract

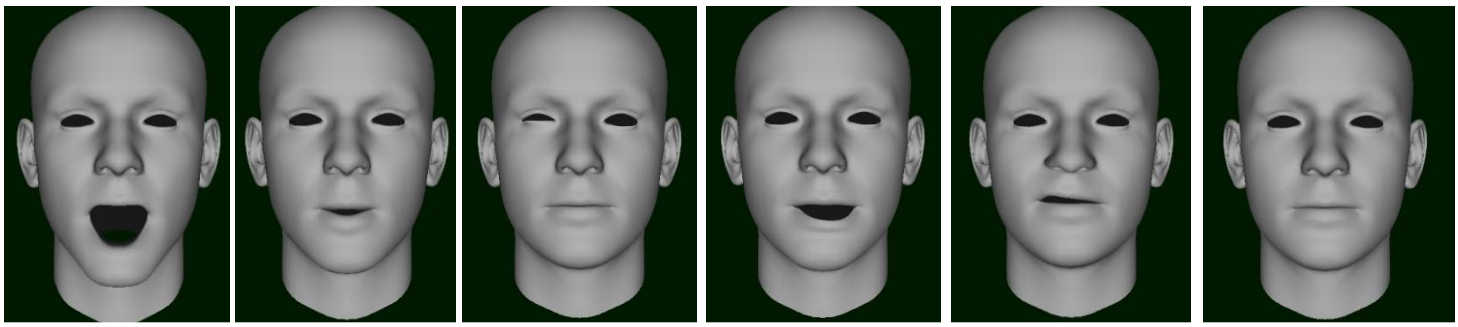
In computer animation, the use of pre-recorded motion capture databases has become an important field of research for both motion analysis and synthesis. This task requires efficient algorithms for fast searching of matching subsequences withing huge datasets. In this paper we present a real-time method to reconstruct facial animation given a sensor input stream, and discuss about its applications. It can be used for denoising motion capture data, reconstruct full dimensional facial animation using partial input and efficient search of disrupted motions contained in a knowledge base. The implementation relies on a high dimensional kd tree indexing a large set of pre-recorded animations for efficient pose searching, and allowing easy scaling to even larger data sets. We then incrementally build an online lazy neighborhood graph - suitable for real-time applications - which is used to retrieve similar sequences in the database. Then a motion reconstruction method is applied, interpolating the frames to produce visually plausible results, even if the input is only partially matching a motion in the knowledge base.

Methodology

1. Efficient similar motions search

1.1 Input representation and motion database

A motion is made up of several frames, each one corresponding to a facial pose. To represent a pose we use control points instead of cartesian coordinates as they are suitable and convenient to describe a facial expression. In the number of 39, control points correspond to an amount of predefined specific pose - between 0 and 1 - such as a smile, a right eye blink etc… (**Figure 1**) By combining them using simple linear combination, we are able to synthesize a wide range of expressions. Thus with just a few points instead of a fully described face it indeed greatly enhances memory cost and computing performance while having good representation and results. The database of animations used for this project consists of different short motion sequences, for a total of 16623 frames or about 10 min. The representation of these frames is as described above, using the 39 control points.



Open mouth *Whistle* *Partial right blink* *Mouth down left* *"Disgust" right* *Neutral*

Figure 1. Various poses corresponding to different isolate control points

If the input consists of different various markers positions in cartesian coordinates it is possible to convert the database in a similar data representation -for searching capability- by matching the corresponding markers after frame analysis. (We can identify a marker position for a given pose using the polyogon indices which are consistent through the blendshapes) Then we can use the original database to get actual fully described poses that correspond.

For fast searching of the k poses most similar to a given input within the database, we use a kd tree, which is notably well suited for nearest neighbors search. Although with high dimensional vectors like we have they tend to become less efficient, we still have satisfying processing time even with a very large number of points (retrieving neighbors among millions of frames is still conceivable for

real-time applications). Thus we first index the knowledge base using a kd tree as a preprocessing step. Currently, the k-nearest neighbors retrieval is exact, and if some computation time problem occurred we could improve it a lot by considering an approximate search.

1.2 Online lazy neighborhood graph

We compare the query input, for which we can add additional noise to simulate motion capture imprecision, to the frames currently in database. The retrieved neighbors can belong to very different motion subsequences although they are similar to the input. Also, when using lower dimensional input (either truncated input or simulated markers) and when the signal to noise ratio becomes low, false positive matches often occur. To cope with this issue and to retrieve similar subsequences within the knowledge base, we use an online lazy neighborhood graph OLNG similar to the one described in [2] and [1]. This data structure is well suited for this task, as it meets the project requirement. First it allows a certain temporal tolerance for retrieved motions, because similar sequences can be achieved at different speeds, while keeping temporal coherence. We only consider strictly monotonous indices for the database frames. (No backward search) While it is possible, the resulting motions clearly lack a “natural” evolution. The OLNG can also be built efficiently and incrementally, with no latency, allowing real-time processing.

The idea of the OLNG is to keep the $K \in \mathbb{N}$ best similar motions at every time step, by considering the last $T \in \mathbb{N}$ inputs. In our case T=32 or about 1s, sufficient for facial animations which are composed of very short motions, like switching from one expression to another (Open to closed mouth for instance). The OLNG is represented using a $K \times T$ array for the nodes, and another $K \times T$ array for the edges. The t-ieth column of the array contains the set of the k nearest neighbors K_t indices for the input i_t at time $t \in \mathbb{N}$. Each path in the OLNG corresponds to a sequence of frames, i.e. a matching sub motion, thus every edges leaving from a column arrives in a column at a previous time. (Temporal coherence) They are assigned a cost depending on the similarity of its components regarding the input as well as the duration. (Longest paths preferred) We update the OLNG for each incoming frame by adding edges from K_t to the previous sets, in such way that the edges meet different criteria while keeping paths of minimal cost. These criteria are related to the previously mentioned temporal tolerance and coherence and applied using index bounds between connected nodes. Figure 2 presents a general overview of the different steps.

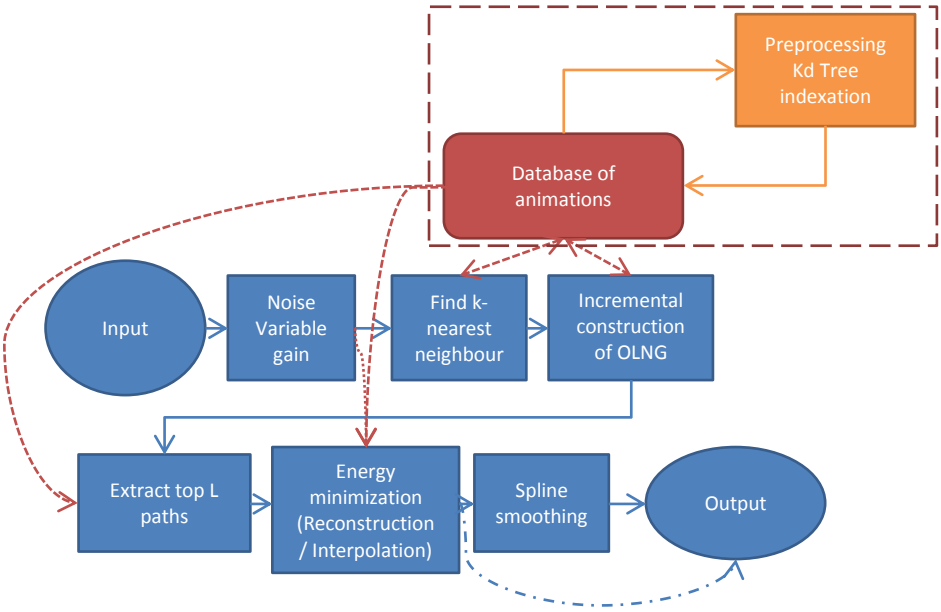


Figure 1.2. Diagram overview of the processing steps

Let's see more in detail how the OLNG construction is achieved.

Preprocessing

As described in Figure 2, we first index the database in a kd tree, which requires O(n.log n) computations. If different input data representation is used, we first adapt the database frames to match the structure, but still keep the original database in memory for later steps. This can be done easily in linear time.

Update loop

1. Search. Given an input pose q_t, we search for the k nearest neighbors in the database that compose the K_t set. The k parameter is user definable. In most scenarios, we have set k=256. The knn retrieval can be achieved in O(k.log n).

2. Graph update

For each neighbor $p_{t,i}$ in K_t, we look for the set of its potential predecessor in K_{t-1}, referred to as $V_{t,i}$. These predecessors fulfill temporal condition for similar motion, i.e. each element of $V_{t,I}$ must be in the range $[p_{t,i}-a, p_{t,i}+b]$. a=0 means we consider index monotonicity as a valid case of continuation, while a=1 enforces strict monotonicity. As stated above, the latter gives better result, but we will see some case when simple monotonicity can compensate some error. b is the maximum number of frames it is possible to look forward in time for a transition to be considered valid. If b=2, it means we can match animations which are up to half the speed of the input sequence. (The index bounds allow to simulate dynamic time wrap.)

If $V_{t,i}$ is not empty a directed edge is added by connect ing $p_{t,l}$ and the element of $V_{t,l}$ which minimizes the cost of the resulting path.

Its cost is computed as follow

$Cost(path_i^t) = \frac{\sum_{i=1}^{l} c_{t-l+i}(s_i)}{l^2}$, $path_i^t$={s_1, s_2,..., s_{l-1}, s_l} . $path_i^t$ is a set of indices corresponding to the path which ends at the i^{th} nearest neighbor of q_t.
$c_t(i)$ is the cost of the nearest neighbor of q_t at position i

The cost of the added edge can be modulated according to the current path at t – 1 so that the algorithm is forced to stay with a motion a bit longer, even if less similar.

By dividing by l^2 we enforce creation of longer paths. Due to the topological order of nodes in the graph and given the fact that we only update paths of previously minimal cost, the single-source shortest path problem is considerably simplified. The edge construction step takes $O(K^2T)$ operations in the worst case. Indeed path length can't exceed the window size t. See Figure 1.3 for an illustration of the edge construction step.

3. Path retrieval.

We just apply a partial sort on the set of path to extract a predefined number of best paths.

Once the graph is updated we have a set of optimal motions ending at some of the neighbors in K_t. Some elements may not have predecessor, so K is only an upper bound limit for the number of paths diffrent. Usually the algorithm rapidly converges to the most similar motion, within a few steps. At this stage, we could consider the last frame of the best path (minimal cost) as the output but this leads to discontinuities in the retrieved motions. When the submotions are similar to each other, the algorithm can't decide and the output may jump between these causing visual discontinuities. Also the error can occur between sequentially matched sequences, when the algorithm doesn't converge to a particular motion.

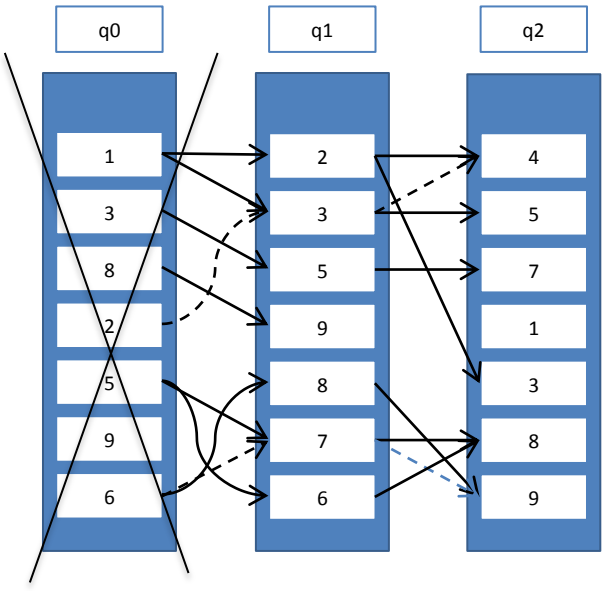


Figure 1.3 Example of an edge construction step. Nearest neighbors are sorted according to the distance to the input. (Dashed edges are not added because not optimal while correct). Because of the finite window size the oldest set of nearest neighbors is discarded.

To provide a visual feedback and representation of the abstract structure and paths, a visual debugging tool was developed for this project. (See Fig. 4) The green rectangle highlights correspond to the different components of the best path currently retrieved by OLNG. The yellow line is to better visualize this path. Each rectangle correspond to a nearest at time t, which is column t % T.

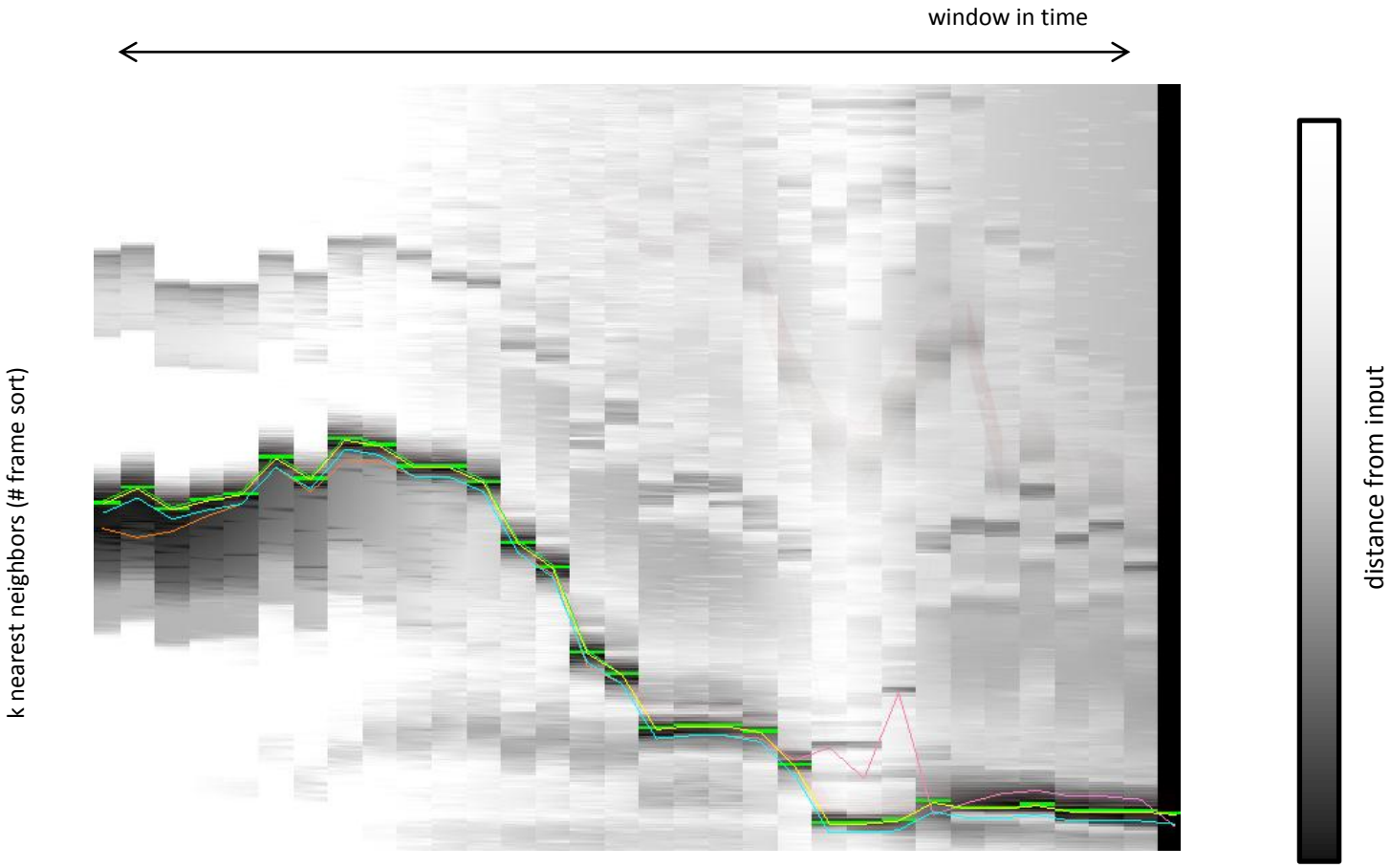


Figure 1.4

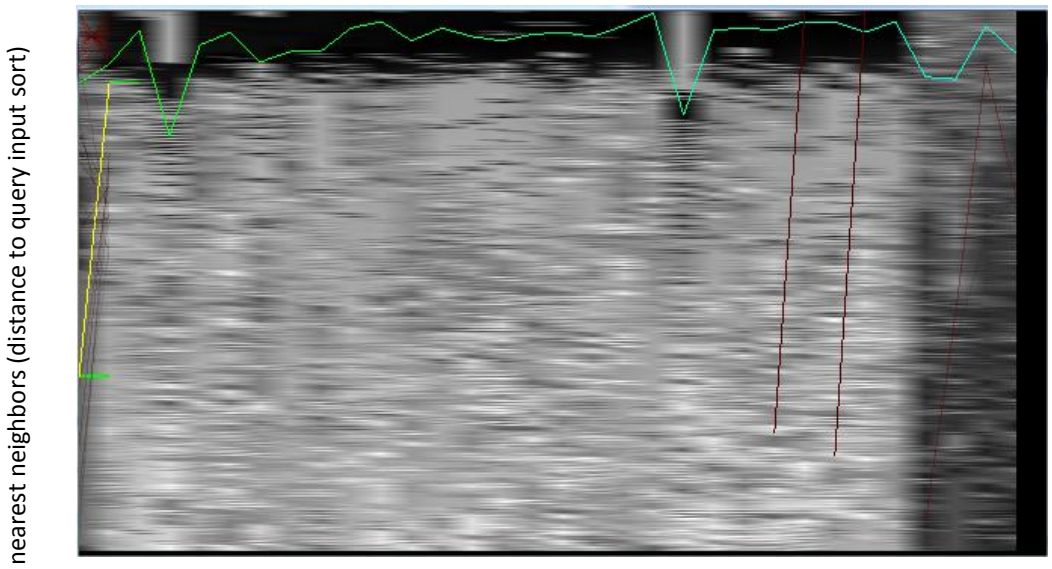


Figure 1.5 Same as above but this time neighbors are sorted by their distance to actual input.

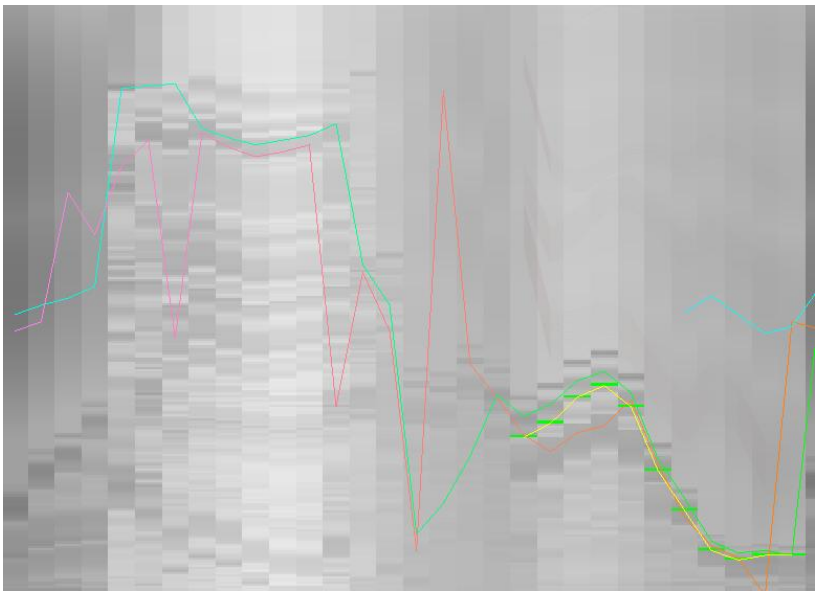


Figure 1.6 Two sequential submotions matched within a short time. In red/pink, the motion reconstructed by the motion synthesis. (Position next to its closest frame)

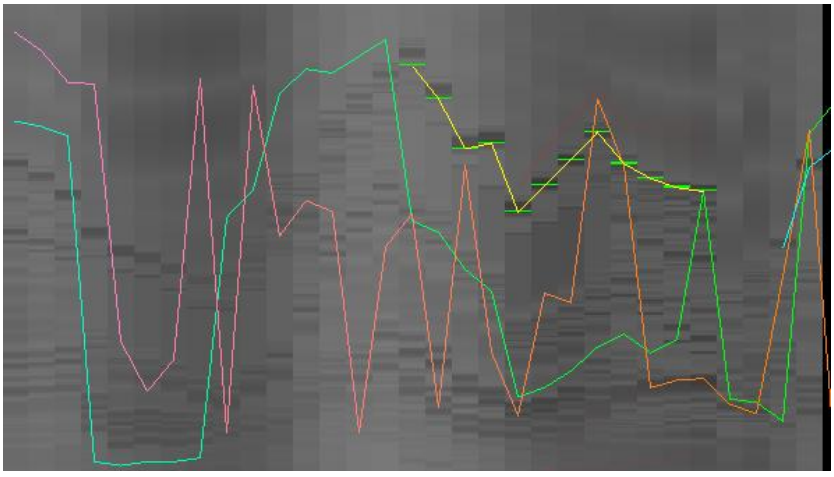


Figure 1.7 Some artifacts caused by switching between motions.

2. Motion synthesis

2.1 Energy minimization

To compensate the error and have smoother, more realistic output, we need to synthesize the motion to reconstruct missing or erroneous poses. Instead of choosing the minimal cost sequence, we use the top I path to reconstruct the motion. (See Figure 2.1) In the case of a motion input not contained in the knowledge base, these paths tend to have a very high cost while being short in time, about 300 ms on average. The method used is similar to [1], and is based on an energy minimization formulation of the problem.

Let $C^t=\{c^t_1, c^t_2, \dots, c^t_I\}$ be the set of path costs computed while updating the OLNG. Given the set of I paths at time t, each one is associated a weight that represents its contribution importance in the final synthesized pose.

$$w_i^t = \frac{\max(C^t) - c_i^t}{\sum_{j=1}^{I} \max(C^t) - c_j^t}$$

Given a pose q, its energy can be computed regarding the last frames of the paths as well as the weight coefficients.

$$E(q) = \sum_{i=1}^{I} w_i^t \times |q_i^t - q|_2$$

The optimal pose likelihood is achieved when the energy is minimized. This step is done using a gradient based approach to approximate the result, as this can becomes computationally intensive and impracticable without approximatin.

The l^{th} coordinate of the gradient is given by

$$\Delta^l E(q) = \sum_{i=1}^{I} w_i^t \times \frac{q_i^{t\,l} - q^l}{|q_i^t - q|_2}$$

We use a specific library (LBFGS) implementing the Broyden-Fletcher-Goldfarb-Shanno method, which can optimize non-linear problems like that. This step requires more time than the previous ones (10 factor) and is directly linked to the number of paths chosen for reconstruction, and to a lesser extent the dimension. If the optimization succeeds, the synthesized pose is the one optimally satisfying the equations.

While the results are visually more plausible than when using solely the OLNG, they still have some artefacts and there is an obvious loss of some high-frequency details. This will be discussed in the results section below. Let's now have a look at the implementation of the algorithm.

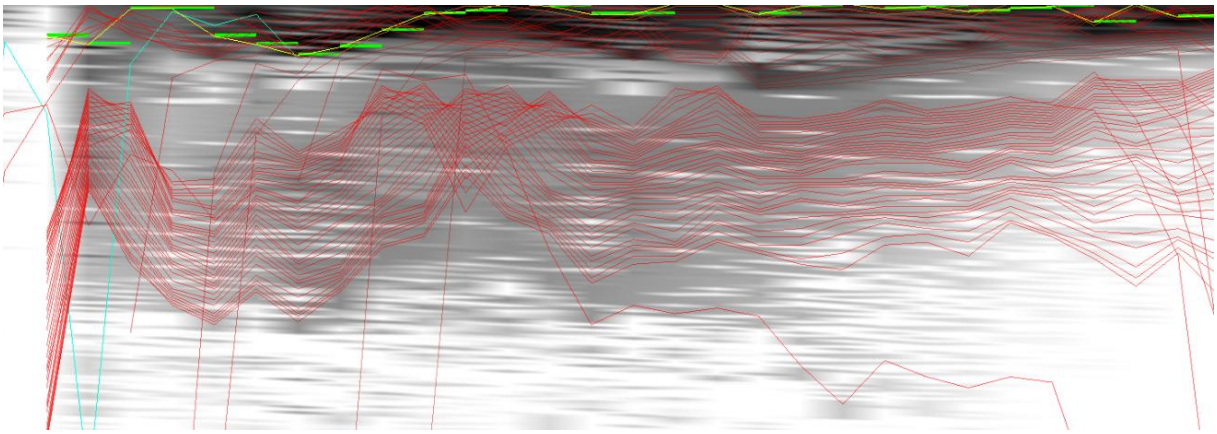


Figure 2. The 50 best paths chosen for reconstruction

Figure 2.2 Some reconstruction step for a noisy input not included in database. From left to right, actual query input, noisy input (what the algorithm knows), the OLNG last frame pick of best path at current time, motion reconstruction.

3. Implementation

The project was entirely written in c++ using Visual Studio. The rendering is done using OpenGL and blendshapes corresponding to the various control points. (Figure 3.1) Focusing on performance was one priority throughout the project. While the database used here is relatively small, the program should be capable of handling much larger sets. The indexing and retrieval of nearest neighbors was done using ANN, a KD Tree implementation library. LBFGS is another library which was used for the optimization of the energy problem. For further understanding of implementation, the best way is to have a look at the code. It would not be convenient to describe implementation decisions while not putting code, which is too long to be displayed here.

One word about rendering :

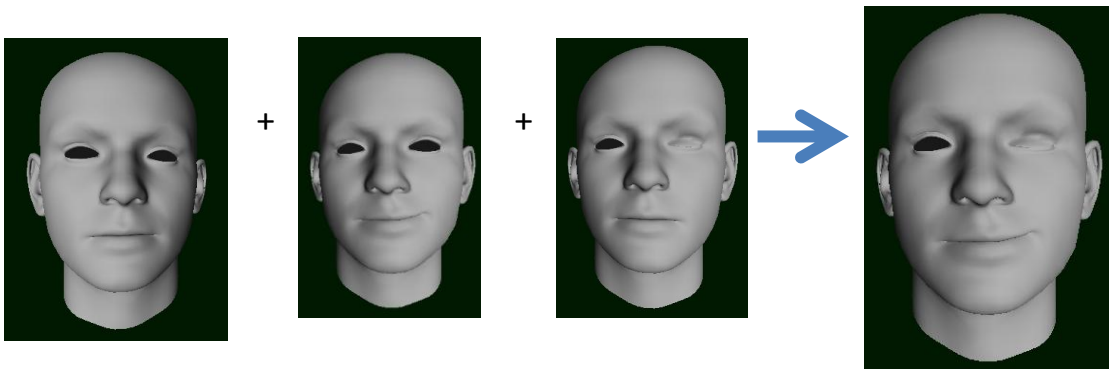


Figure 3.1 Example of blendshapes composition.

The pose is simply synthesized by adding a linear combination of blendshapes's vertices related to control points.

4. Results

The motions produced by the method presented in this paper highly depend on the quality and size of the database itself. Also the results appear to be varied according to the different query input scenarios tested.

While this algorithm was shown to be suitable for full body reconstruction using joint angles as in [1], we have higher dimensional data and the facial animation is somewhat different to the first one in many aspects. Facial movements are much quicker, subtle and precise. For instance, in a full body representation it doesn't really matter if the elbow is 5% or 10% different than the real pose provided the movement globally match the motion query. As they also last usually longer, there are more chances that the algorithm converges to a good solution. But in the case of facial animation, with a lot of short sub motions, this can give erroneous expressions which are visually inconsistent. (Disgust instead of anger etc..) One solution is to focus more on local variations as we did in this project. But we have the discontinuous motion problem as with the OLNG back again. It is also not really conceivable to predict further frames by using forward kinematics, such as integrating control points "velocity" as the effect of momentum is basically insignificant when it comes to movements of the face. It would however be possible to lower the importance of some of the control points, to more closely match the distance evaluation of the algorithm with a human one. Indeed, a person does not associate the same amount of information for the different parts of the face. While slightly improving the result in term of visual plausibility, in our tests it lowers the range of expressions that can be output. Further study and visual assessment would be needed to estimate the influence of the different control points in the human perception of a pose.

If the motion to be analyzed is however composed of sequences which are in the database, and even with high noise added, making it unrecognizable to the eye, the error is usually remarkably limited, with the motion being almost fully recovered by the OLNG. The remaining error is generally around motion disruptions (between the end of a motion and the beginning of another one)

4.1 Influence of OLNG/motion synthesis parameters on results

Window size T

While reducing the window size significantly animation quality, using T > 32 has limited benefits. Indeed it is only a limiter for the lengths of the matched motions in the database, that, in practice, are quite short. The only situation when it becomes useful is when considering an input animation which is contained in the database. In that case, it allows to prevent switching between different motion by enforcing the choose of longer path, which are often already correct.

Nearest neighbors K

The number of nearest also has limited effect on the output. Worse, when considering the previously mentioned situation, we may have unrelated sequences matched, but with a long length, so they are considered in the final reconstruction and disturb it.

Number of paths used to reconstruct motion I

Increasing the number of paths used in the synthesis step give smoother animation while in some cases reducing the amount of output details. Certain motions may cancel off each other for instance. In our tests, I=50 give the best compromise between reconstruction and error consistency through different scenarios.

Index bounds

While imposing strict monotonicity usually gives better results, in the case of a motion not recorded in the database, it can be useful to remove the strict condition. Indeed some of the motions may be similar to one in the database, but performed much slower. Due to the discrete sampling frequency, it is only feasible to retrieve correctly a motion realized at same speed or faster than the one in the database (when tweaking the upper index bound). By removing this condition, we allow the algorithm to "wait" for the rest of the input to come and then we avoid to discard a potentially satisfying sequence, which would inevitably occur if the input motion is too slow. (Frames are more and more delayed at each time step) It is however not the ideal solution, as it tends to make the animation "lag". Run time interpolation using spline didn't show much improvement. But "slowing down" the entire animation database using interpolation in the pre processing part while using larger upper index bound might reduce the reconstruction error for this specifc problem.

KxT	256x32	256x48	512x32	512x48
256x32	1	-	-	-
256x48	0.96	1	-	-
512x32	0.88	0.85	1	-
512x48	0.80	0.81	0.91	1

Figure 4.1 Error correlation for different parameters

4.2 Comparison of error for different scenarios and methods

4.2.1 Relative error to query

We consider different scenarios regarding the input data, whether they consist of motions included in the database or not, and measure the average error. Error here is the distance to the actual query input.

- #1 : motion in database + noise
- #2 : series of motions in database + noise
- #3 : input not in db
- #4 : noised input not in db

Scenario\Error	Average Noise	Nearest Neighbor	OLNG	Motion synth	Avg best path length
#1 medium noise	0.186	0.009	0.016	0.149	29
#2 high noise	0.562	0.079	0.093	0.204	28
#3 no noise		0.36	0.503	0.463	12
#4 low noise	0.106747131	0.53	0.556	0.486	11

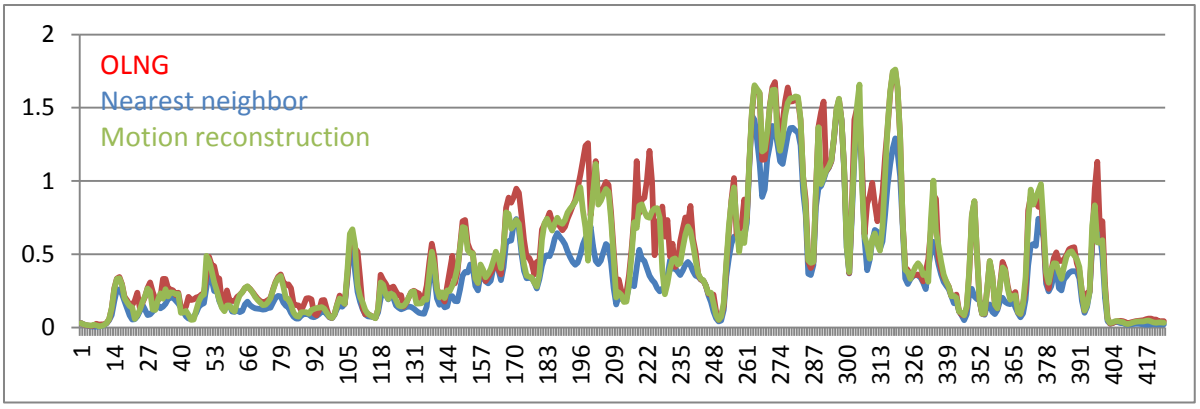
	Nearest Neighbor	OLNG	Total		
correct hits #1	163 (67%)	196 (81 %)	242		
correct hits #2	362 (44%)	439 (54%)	814		

(Correct hit is when the output frame matches exactly the input one.)

As we could expect, in the first two scenarios, motion synthesis just adds error to the motion retrieved. Considering solely the best frame of the OLNG as output gives a better matching. (The synthesized result is not contained in the database)

While choosing closest nearest neighbor seems to give low error in every situation, the results are not visually satisfying, due to inconsistent motion frames. We must keep in mind that this error measure can't be considered as the ideal similarity measure between motions. Visual assessment remains necessary when comparing quality of reconstruction.

Motion synthesis seems to enforce robustness of the algorithm to noise, especially when considering scenarios like #3 and #4, as well as improving close matching of the input and smoothness. (Thus plausibility) However in those cases, there is still too much discontinuity at some times.



Error reconstruction for a noised animation not contained in the database

4.3 Performance

The following tests represent the average computation time on a laptop with an i7 620 M (2.7 GHz), with optimization for speed enable in the Visual Studio compiler.

Computation mean time (ms) \ KxT	128x32	256x32	512x32	1024x32
Kd Tree Search	0.85	0.90	1.04	4.6
OLNG update	0.09	0.29	1.13	4.41
Motion Reconstruction	4.11	4.10	4.22	4.25
Total	5.05	5.29	6.37	13.26

The computation time doesn't depend on the window size, which can be arbitrarily long but should be left less than the minimum length of motions contained in the database. The last two cases with K=512, 1024 are here just to show the impact on performance, as in practice the result becomes of lesser quality with such many neighbors.

We tried different database sizes with KxT : 256x32.

Database size (random data)	**Average processing Time**
5 mo	5 ms
30 mo	8 ms
100 mo	22 ms

We can see the performance is quite good and scales well to larger databases. They have been generated randomly to approximate the real kd tree performance.

Application : reconstruct full facial representation data using lower dimensional input

The algorithm presented is flexible and can be easily adapted to different data representation. One application is to use less control points but produce a fully dimensional output. As control points define pose and do not consist in cartesian coordinates markers, it is not possible to remove arbitrary control points. Indeed some important details will be lost because each one represent a pose. Still some of them are relatively correlated, and we can carefully remove a number of them.

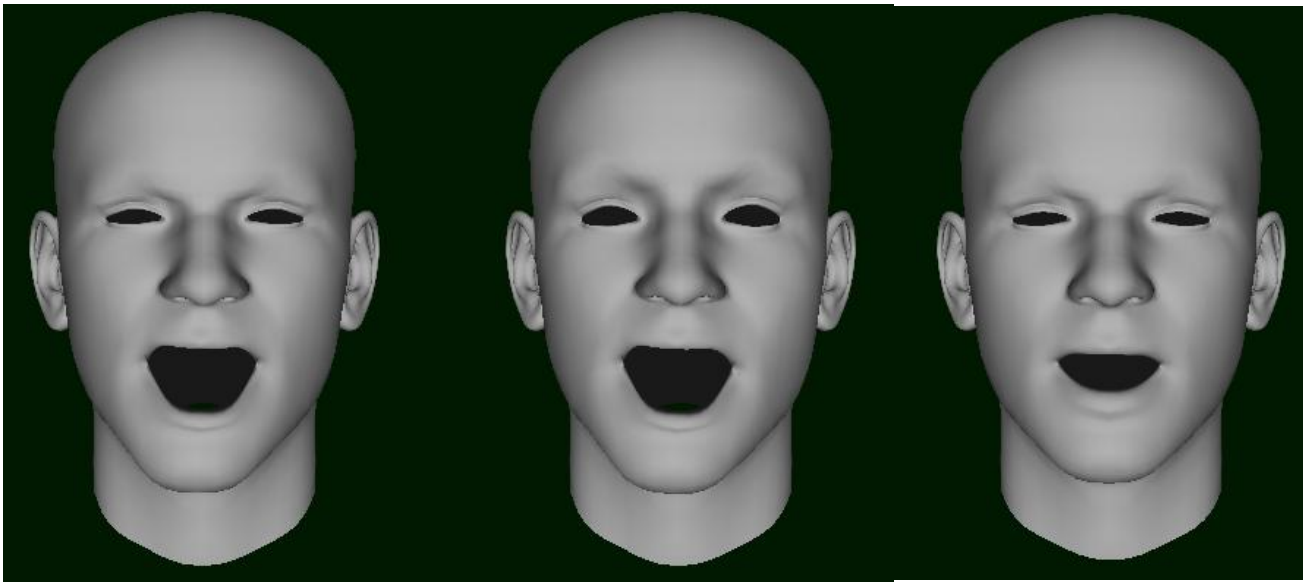
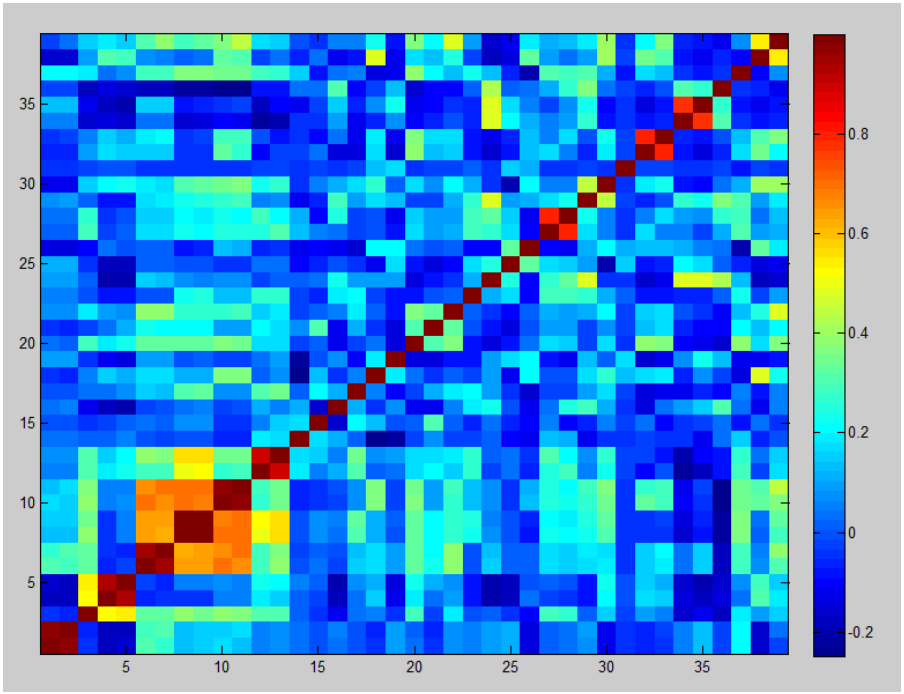


Figure. The first pose is the original input (not in database). The second one, which is the input seen by the algorithm, is the same as the original but all 10 control points concerning eyes are removed. The third one is the output of the OLNG algorithm. As mouth and eye motion are naturally correlated to some extent, we can see that the eye reconstruction is clearly accurate in that case. This is just an example however, while it works there, this is obviously a bad choice of removing these points as the eyes can move independently from the rest of the face.

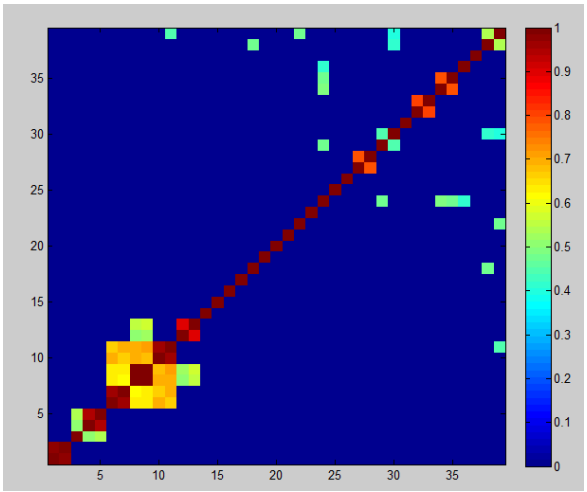
Correlation of different control points

To simplify the process of choosing which control point can be removed, we measured correlation between every two-pair of control points for the entire database and the resulting matrix is shown here.



Correlation of the control points.

By eliminating the less correlated pairs (using a threshold >0.4) we can identify the most correlated points.



We can't simply remove one out of two correlated points for the same reason as given previously. For instance the two eyelids are correlated (most of the time, we blink with both eyes altogether) but one may solely blink with the eye for which control point has been removed and this would be impossible to detect. In the end, 13 markers could be removed without losing essential information, leaving a 26 input dimension.

The results are on average only 4 % better for scenarios like #1 and #2 when using full dimension input compared to the truncated one. For #3 and #4, the difference is respectively of 5 and 8 %.

It is possible to find other points to remove but it would be more practical to use a different data representation for the search of similar frames and switch back to the control points.

Thus to be able to use arbitrary dimensional input, we should first convert the input query and the database as a set of defined markers in cartesian coordinates, and tweaking the number of markers used. Then the original database is used to reproduce the motion. This requires only little modification of the implementation (loading step) as it already uses two databases : one for the motion search, and the original one for the synthesis.

Conclusion and further work

In this paper, we have presented a method for retrieving similar motions to a given input animation in an animation database. We have seen some of its potential application, like denoising an input – partial or not - already contained in the database. This task is the one which gives best results, with fast accurate matching even with a very low signal to noise ratio. However when it comes to other type of motions, the results are much more mixed. This is partially due to the fact that animation output is highly linked to the database used. Indeed the data-driven reconstruction is done based on the database. One way to improve the results would be to enhance it in both quality and quantity. It also appears we can't easily interpolate the result without risking to alterate it. Moreover, we must keep in mind that jumps between poses is a problem for any online method when confronted to ambiguous inputs. Generally, smoother results lead to high-frequency detail loss, while close matching give potentially visually inconsistent reconstructed motions. The method implementation has shown to be very efficient and suitable for real-time applications. It can be interesting to work more on a different data representation which would allow for a more flexible control on the input, and adapt the implementation consequently.

Bibliography

[1] Motion Reconstruction Using Sparse Accelerometer Data, *Jochen Tautges, Arno Zinke, Björn Krüger, Jan Baumann, Andreas Weber, Thomas Helten, Meinard Müller, Hans-Peter Seidel, and Bernd Eberhardt*, 2011

[2] Fast Local and Global Similarity Searches in Large Motion Capture Databases, *Björn Krüger, Jochen Tautges, Andreas Weber and Arno Zinke*, 2010

Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases, *Bohm C., Berchtold S, Keim D.A.*, 2001

Performance animation from low dimensional control signals. *Hodgins J.K, Chai J.*, 2005

ANN: A Library for Approximate Nearest Neighbor Searching. Programming manual

libLBFGS: a library of Limited-memory Broyden-Fletcher-Goldfarb-Shanno