MoDB: Database System For Synthesizing Human Motion

Timothy Edmunds S. Muthukrishnan Subarna Sadhukhan Shinjiro Sueda

Rutgers, The State University of New Jersey

{tedmunds,muthu,sadhukha,sueda}@cs.rutgers.edu

Abstract

Enacting and capturing real motion for all potential scenarios is terribly expensive; hence, there is a great demand to synthetically generate realistic human motion. However, it is a central conceptual challenge in character animation to generate a large sequence of smooth human motion, in a synthetic way.

We present a novel, database-centric solution to address this challenge. We propose to generate long, synthetic strips of motion from a database of a small set of real motion captured data, in particular, using various similarity-based "joins" of real motion snippets.

This demo will illustrate our system MoDB and showcase the entire process of (a) capturing small real motion data and generating 'snippets" (b) encoding snippets it into relational data, and (c) generating synthetic, realistic smooth motion by concatenating snippets based on similarity metrics. In particular, the demo will be interactive consisting of an "avatar"—a graphical creature—that continually moves towards users' motion targets in a smooth human-like motion and will rely on real time performance of the database for indexing and similarity searching from snippets. We believe that synthetic motion generation is a new application area where database technology can have a significant impact.

1 Introduction

When creating motion for animated characters, historically, a number of approaches have been taken. Hand-crafted motions by animation artists was the norm before computer-synthesized motion was available. With the advent of computer-assisted animation, the potential for totally synthetic motion generation arises: one can use motion capture systems to acquire either the entirety of the desired motion or a pool of motions from which further motions can be synthesized.

A number of approaches have been taken to the synthesis of novel motions from a pool of motion-capture data; we will describe them in §2. We propose a new approach based on using a database system to index and retrieve motion snippets. We will describe all the steps involved in §3, and describe the demo in §4. Concluding remarks are in §5.

2 Previous Work

The increasing accessibility of high-quality motion-capture systems that record real character actions has generated growing interest in methods for synthesizing character animations from pre-recorded motion data [1, 2, 3, 4]. In order to generate synthetic motions that go beyond simply playing the captured animations, the captured motion frames must be recombined into sequences that were *not* performed by the motion-capture subject. Animators (both artists and automated processes) can use motion sequence recombination to drive the actions of an animated character.

When recombining captured motion frames to form novel sequences for character animation, there are two considerations that must be addressed:

- the feasibility of the motion, and
- the degree to which it accomplishes the action desired by the animator.

Arbitrary reordering of motion capture frames will result in discontinuous, physically infeasible motions; to meet the minimum requirements of feasibility, a synthesized motion sequence must achieve *continuity between adjacent frames*. In previous work [1, 2, 3], authors assess motion continuity by means of a distance function between adjacent frames. The distance function may be a weighted combination of joint angle differences [1, 3], a weighted sum over a skeletally deformed point-cloud [2], or any other cost function that quantifies the feasibility of the motion as perceived by a human observer.

By establishing a threshold for the adjacent-frame distance function, and incorporating other constraints (such as contact with the environment), [1, 2, 3] precompute *motion graphs* whose nodes consist of motion frames and whose edges represent feasible transitions between frames.

In order to synthesize character animations from the motion graphs (other than by random walks), it is necessary to incorporate an objective function that, for a given set of feasible transitions, quantifies the degree to which the resulting motion would accomplish the goals of the animator. The manifestation of this objective function is dictated largely by the application; it can be as concrete as the Euclidean distance between the result and some trajectory [2, 3], or as abstract as matching high-level motion annotations (such as "running", "jumping", or "happy") [4].

A severe limitation of the motion graph approach discussed above is that the continuity constraint is considered only in the pre-processing stage. This makes preprocessing costly. In addition, and more significantly, this dictates that motion quality cannot be traded off against the objective function. That is, the animator cannot force generation of a motion that accomplishes a desired goal that cannot be achieved by the *preset* recombination of the available motions as given by the motion graph! [4] address the tradeoff between the continuity constraint and the objective function by combining the two into a single cost function which is then optimized by a method centred around dynamic programming. Still, the fundamental gap remains: how to provide for flexible continuity constraints in a dynamic, user-driven manner, in synthetic motion generation.

3 MoDB

We propose an approach to synthetic motion generation using database technology. Our approach to the problem of generating long realistic motion sequences from a small pool of motion frames is to leverage the capabilities of a relational database management system.

At the high level, our approach works as follows. We acquire small sequence of motion data using appropriate tools, and decompose the motion data into a set of contiguous sequences or "snippets". We load the snippets and their attributes into a database. In order to generate synthetic motions, we phrase both the continuity constraint as well as the animation objective function (discussed in §2) as queries over the database. By adjusting the parameters of the query, we can control the tradeoff between the continuity constraint and the objective function. We balance database retrieval time versus motion execution delays to get real-time realistic synthetic motion for very long durations from a small pool of snippets. In what follows, we describe each of the steps in more detail.

3.1 Data Acquisition

The character motion data that forms the pool of motions available to our system is obtained using a Vicon optical motion tracking system. The system uses 6 cameras, recording at 60Hz, to track the 42 markers worn by the subject as he performs the motions within a 4mx4mx2m capture volume. The raw outputs of each camera are combined to triangulate the 3D marker locations at each video frame.

Through a semi-automated process, we use the triangulated marker locations to fit a skeleton to the recorded motion (by varying joint parameters such as the angle of the elbow or the orientation of the hip). The processing takes into account temporal continuity between consecutive frames.

Each captured motion sequence is divided up into several short snippets, each comprising $\sim$30 consecutive frames from the original motion. The extraction of the snippets is performed in an overlapping fashion (the first snippet is composed of frames 0 to 29, the second snippet from 1 to 30, etc.). The final result of the preprocessing is the pool of motion snippets extracted from all of the captured example motions.

There are data-quality issues to be considered when working with motion capture data. Since the quantities of interest (joint angles) are modelled and estimated from other quantities (marker locations), errors can be introduced by marker mis-estimation, marker mislabelling, unwanted/unmodelled movement of markers with respect to the subject's bones, etc. The motion capture processing software we use provides a mechanism for rapidly cleaning the data to eliminate/minimize many of these errors - working on entire sequences of motions rather than frame by frame. Even after processing, however, there is still the possibility that the joint parameters recorded in our data will not entirely accurately represent the motions that occurred. As in any real database application scenario, our system needs to work with such data quality issues.

3.2 Relational Representation

The motions synthesized from our data set are composed by stringing together short ($\sim$0.5s) sequences (*snippets*) of the original motion capture data into longer, continuous sequences. Within each snippet, there is frame continuity because the frames appear *in*

the same order as originally recorded; however, continuity *between* snippets must be maintained by choosing only compatible snippets, which we do based on the configuration of the synthetic character—avatar—at the beginning and end of each sequence, as follows.

A configuration of the character is represented as a relation (*MotionTerminator*) whose attributes describe the joint parameters in a 17 joint skeleton. The joints are each one of two different types: free joints with a full 6 degrees of freedom (3 position, 3 orientation), or ball joints with 3 degrees of rotational freedom. 7 attributes for each free joint and 4 for each ball joint (orientations are stored as normalized *quaternions*) thus allows for the detection of C^0 continuity. In order to also allow for consideration of C^1 continuity, ball joints are issued an extra 3 attributes for angular velocity, and free joints an extra 6 attributes (for velocity and angular velocity). In total, the *MotionTerminator* relation contains 137 double format attributes, and 1 primary key attribute.

As well as maintaining continuity between our selected snippets, we also have a preference for some snippets over others based on the objective of the animation. In our application, the objective is to move the character (in a feasible way) toward some target in the character's environment. Thus, for each motion snippet, we need to be able evaluate how effective the snippet is at moving toward the target position. Our database contains a relation (*SnippetDescriptor* that describes the pertinent aspects of an entire motion snippet; as well as references to a starting and an ending *MotionTerminator*, it contains 3 attributes that give the 3 dimensional net translation of the character that results from playing a snippet. Since we can start playing a motion snippet in any position and orientation in the environment (to match the ending position and orientation of the previous snippet), the net translation is recorded with respect to the character's configuration at the start of the motion snippet.

Since a snippet's net translation is recorded with respect to the snippet's starting configuration, we need to be able to transform the target location from world coordinates into the current local coordinates of the character. To keep track of the character's local coordinate frame, we need to know the net rotation resulting from playing a motion snippet (as well as the net translation). Though this can be obtained by patiently playing through the snippet, in order to be able to compute the pertinent state independently of animation process, we record 4 extra attributes (the normalized quaternion recording the final orientation) in the *SnippetDescriptor* relation. In total, the *SnippetDescriptor* contains 7 double format attributes, 2 foreign key attributes, and 1 primary key attribute.

When serialized in the most convenient format for driving an animated character in realtime, each motion snippet is $\sim$100kB in size. For a non-trivial data set, retaining the snippets in memory rapidly becomes infeasible. Since a DBMS is expected to be at least as fast at retrieval from disk as ordinary application I/O, we opted to store the snippets serialized as BLOBs in a third database relation.

3.3 Snippet Selection

As alluded to above, when the application chooses which snippet to play next, two criteria are involved: the beginning of the chosen snippet must be "close enough" of a match to the ending of the previously played snippet, and given a set of candidate snippets, the snippet that will move the character the closest to its current goal is preferred. These two criteria are expressed in the selection query by a WHERE clause and an ORDER BY clause respectively.

The WHERE clause of the database selection uses inequality comparisons to reject motion snippets whose starting configurations have joints that deviate more than a specified amount from the ending configuration of the previous snippet. The challenge presented in composing this query is in choosing the metric by which joints are compared. For the position components of free joints, the L_2 norm of the distance between the joints is most appropriate, but requires mathematical aggregation of multiple attributes; the L_1 norm, though less accurate a representation of 3 dimensional distance, is easily expressed using only comparisons between two values of the same attribute. For the joint orientations, the situation is even more dire since the L_1 norm of a quaternion difference is a very poor metric of orientation discontinuity. Again, however, it is possible to obtain a more meaningful quantity (the magnitude of the rotation necessary to go from one orientation to the other) by computing mathematical aggregates of all 4 of the quaternion attributes.

The ORDER BY clause that prefers snippets that take the character closer to its goal is simpler than the WHERE clause. After the application has transformed the target position into the character's current local coordinates, all that is necessary is to define an attribute in the output relation that contains the difference between the target location and the snippet's net translation. Again, the choice of difference metric determines the attribute aggregation that is necessary.

3.4 Query Tuning

Given the query structure described above, there is a great deal of flexibility in the actual query issued to select motion snippets.

The thresholds used to parameterize the similarity tests on the joint configurations can be varied to increase the number of candidate snippets at the cost of degrading the motion's feasibility (and vice-versa). In addition to tuning the query to trade motion quality off against the objective function, the query can be modified to improve the query execution time (also at the expense of motion quality). As mentioned above, the L_1 norm joint difference is more likely to optimizable by a DBMS than the l2 norm which involves aggregation of multiple attributes. Also, joints closer to the root of the skeleton's hierarchy contribute more to the perceived continuity of motions; by eliminating selected joints from the query, the complexity of the join condition is reduced. Similarly, joint velocity and angular velocity can be ignored for some or all of the joints.

Reducing the complexity of the WHERE clause is not without its cost (as well as the degradation of the motion quality). Fewer constraints on the continuity match results in a larger number of selected records, which results in a larger sorting operation for the ORDER BY clause.

The importance of query response time depends on the purposes of the application. For an off-line animator assistance tool, query execution times on the order of a second would be reasonable. For our application, however, we desire real-time user interactivity; the selection of the next motion snippet to play must not delay the animation. In this situation, the query response time governs the latency of the character's response to user input. The minimum latency is the time between two frames of animation ($\sim$17ms for a standard 60Hz animation); if the query response time is less than this period, the snippet selection adds no overhead to the user-response latency. As the system scales to a larger data set, and the minimum response time becomes unachievable, motion quality can be maintained by increasing the user-response latency; by starting the selection of the next snippet before the current snippet has finished playing, the DBMS is given more time to process the query at the cost of ignoring user input that occurs after the selection is started.

4 Demo

Our demo will show MoDB, our system for synthesizing long sequences of realistic motion from a small set of captured, real motion data. The demo will render, in real-time, an *avatar* that responds to user input. The user will be able to interactively select a goal for the avatar to pursue. Each time a new motion snippet is required, the system will query the DBMS for a matching motion snippet that will take the avatar as close to the goal as possible.

We will demonstrate the effect that the parameters controlling the selection of motion snippets have on query response time in the DBMS. By changing such parameters as the distance norm used or different joints on which to perform the joins, the user will be able to, at run time, balance the visual fidelity and responsiveness of the avatar. The goal is to convince the audience that (a) real-time synthetic generation of realistic motion is highly feasible, and (b) the database technology crucially impacts the performance of the application.

Under the hood, the system relies on mySQL for database technology, OpenGL for animation and Java for the application logic and database/graphical glue. Much of the database queries are far too complex to hand code because of the number of attributes involved in the queries and the sophisticated constraints on them. So, we had to develop scripting programs for generating queries. An example query is presented in the Appendix. A video tape of the avatar's synthetically generated motion is shown in the attached file.

5 Concluding Remarks

This demo shows the power of the database in a new application of generating long sequences of synthetic motion from a small set of captured, real motion data. Our system relies on decomposing input data into snippets and relying on the DBMS to index the snippets and retrieve them based on similarity criteria for any desired synthetic motion. Our system MoDB demonstrates the real-time performance of this application. DBMSs promise to have a significant impact in solving the challenge of generating synthetic motion, and likely to have a commercial impact in the emerging industry of synethetic motion generation for animation.

Acknowledgements

Data acquired using the HAVEN facility at Rutgers. Development of the HAVEN was supported in part by NSF grants EIA-0215887 and IIS-0308157 (PI: Pai).

References

[1] O. Arikan and D. A. Forsyth, "Interactive motion generation from examples," in *Proceedings of SIGGRAPH 2002*, July 2002.

[2] L. Kovar, M. Gleicher, and F. Pighin, "Motion graphs," in *Proceedings of SIGGRAPH 2002*, July 2002.

[3] J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. S. Pollard, "Interactive control of avatars with human motion data," in *Proceedings of SIGGRAPH 2002*, July 2002.

[4] O. Arikan, D. A. Forsyth, and J. F. O'Brien, "Motion synthesis from annotations," in *Proceedings of SIGGRAPH 2003*, July 2003.

Appendix

The following is an example of one of the queries generated by our system. Note that this query is a very small subset of a usual query; it deals with only one of the skeleton's joints (the thorax). The symbol '?' denotes a variable parameter.

```
SELECT
  descriptor.SnippetID AS SnippetID,
  SQRT(((descriptor.PelvisTranslationX - ?)*(descriptor.PelvisTranslationX - ?))
     + ((descriptor.PelvisTranslationY - ?)*(descriptor.PelvisTranslationY - ?))
     + ((descriptor.PelvisTranslationZ - ?)*(descriptor.PelvisTranslationZ - ?))) AS targetProximity
FROM
  SnippetDescriptors descriptor, MotionTerminators previousEnding, MotionTerminators candidateBeginning
WHERE
  candidateBeginning.TerminatorID = descriptor.BeginningTerminatorID
  AND
  previousEnding.TerminatorID = ?
  AND (
    SQRT(
      (candidateBeginning.ThoraxPositionX - previousEnding.ThoraxPositionX)
     *(candidateBeginning.ThoraxPositionX - previousEnding.ThoraxPositionX)
    + (candidateBeginning.ThoraxPositionY - previousEnding.ThoraxPositionY)
     *(candidateBeginning.ThoraxPositionY - previousEnding.ThoraxPositionY)
    + (candidateBeginning.ThoraxPositionZ - previousEnding.ThoraxPositionZ)
     *(candidateBeginning.ThoraxPositionZ - previousEnding.ThoraxPositionZ)
    ) <= ?
  )
  AND (
    2*ACOS(
      candidateBeginning.ThoraxOrientationW * previousEnding.ThoraxOrientationW
    + candidateBeginning.ThoraxOrientationX * previousEnding.ThoraxOrientationX
    + candidateBeginning.ThoraxOrientationY * previousEnding.ThoraxOrientationY
    + candidateBeginning.ThoraxOrientationZ * previousEnding.ThoraxOrientationZ
    ) <= ?
  )
  AND (
    SQRT(
      (candidateBeginning.ThoraxVelocityX - previousEnding.ThoraxVelocityX)
     *(candidateBeginning.ThoraxVelocityX - previousEnding.ThoraxVelocityX)
    + (candidateBeginning.ThoraxVelocityY - previousEnding.ThoraxVelocityY)
     *(candidateBeginning.ThoraxVelocityY - previousEnding.ThoraxVelocityY)
    + (candidateBeginning.ThoraxVelocityZ - previousEnding.ThoraxVelocityZ)
     *(candidateBeginning.ThoraxVelocityZ - previousEnding.ThoraxVelocityZ)
    ) <= ?
  )
  AND (
    SQRT(
      (candidateBeginning.ThoraxAngularVelocityX - previousEnding.ThoraxAngularVelocityX)
     *(candidateBeginning.ThoraxAngularVelocityX - previousEnding.ThoraxAngularVelocityX)
    + (candidateBeginning.ThoraxAngularVelocityY - previousEnding.ThoraxAngularVelocityY)
     *(candidateBeginning.ThoraxAngularVelocityY - previousEnding.ThoraxAngularVelocityY)
    + (candidateBeginning.ThoraxAngularVelocityZ - previousEnding.ThoraxAngularVelocityZ)
     *(candidateBeginning.ThoraxAngularVelocityZ - previousEnding.ThoraxAngularVelocityZ)
    ) <= ?
  )
ORDER BY targetProximity LIMIT 1
```