USING XML-FORMATTED SCORES IN REAL-TIME APPLICATIONS

Joachim Ganseman, Paul Scheunders
IBBT - Visionlab
Dept. of Physics, University of Antwerp
Universiteitsplein 1, building N
B-2610 Wilrijk (Antwerp), Belgium
{*joachim.ganseman, paul.scheunders*}*@ua.ac.be*

Wim D'haes
Mu Technologies NV
Singelbeekstraat 121
B-3500 Hasselt, Belgium
wim.dhaes@mu-technologies.com

ABSTRACT

In this paper we present fast and scalable methods to access relevant data from music scores stored in an XML based notation format, with the explicit goal of using scores in real-time audio processing frameworks. Quick and easy access is important when accessing or traversing a score, for instance for real-time playback. Any time complexity improvement in these contexts is valuable, while memory constraints are usually less important. We show that with some well chosen design choices and precomputation of the necessary data, runtime time complexity of several key score manipulation operations can be reduced to a level that allows use in a real-time context.

1. INTRODUCTION

In real-time audio processing software, the use of music scores is not commonplace. To fill that gap, we started the construction of a small C++ software library for handling MusicXML files, specially tailored for use in real-time audio processing software frameworks and streaming applications. Since a score is for many people a well-known way to represent music, we consider this important functionality that has been strangely absent in real-time audio frameworks until now.

Being able to use XML-encoded digital music scores natively in real-time environments has clear advantages over the alternatives that are often used now, like MIDI [1] or the development of an own ASCII-based format. The ability to use the countless mature software tools that are available for XML parsing and processing is the main reason to prefer XML-based formats over others. Nowadays most score file formats encode very detailed information, and XML formats can be easily extended or stripped to add or remove information, without needing to adapt the parser, which is much more difficult with binary or plain text file formats. Also most upcoming web developments are centered around XML-based standards (the semantic web etc.). For all of the aforementioned reasons, we will only consider XML-based file formats here.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
© 2009 International Society for Music Information Retrieval.

MusicXML [2, 3] is the most widely used XML-based file format for scores at the moment, but others exist. MEI [4, 5] is a mature alternative and provides valuable functionality to encode versioning and history tracking in documents. The WEDELMUSIC format [6] was developed as all-round multimedia format in an academic context and seems not to be under active development anymore, but its legacy can more recently be found in MPEG SMR [7] and IEEE P1599/MX [8], that are both striving to include score information in a broader multimedia context.

We widen the scope of this paper to all of the aforementioned formats, as they are all XML-based and use similar hierarchies to encode scores. The principles outlined in this paper here thus hold for any of these formats. Also the programming language is of lesser importance: in any major object-oriented programming language, the argumentation for the design decisions will hold.

2. PROBLEM STATEMENT AND REQUIREMENTS

In real-time audio processing, audio data is processed frame by frame, the necessary operations need to be performed within a certain time, and then the results are written to an output buffer. The frames are usually kept small to minimize delays. This leads to very strict time constraints, as also the operating system's scheduler will lay claim to some time for other processes or interrupts.

A music score is layered on several levels (voices, instrument parts, chords), but these layers are often very much interlinked (like voice crossing). This makes it unfeasible to find an ideal single XML hierarchy to represent a score. The result is that notes, voices and/or parts that are active at the same time can be found encoded in very different places in the file. If you need to access all notes occuring at a single moment, you may need to access data at tens to hundreds of different positions in the score file. Processing time is very limited, and user interactivity or display of the

score require also quick access to random positions in the score. This makes it becomes quickly undesirable to do the necessary data structure traversals in real-time, based only on the existing XML hierarchy.

XML-based score data formats tend to produce really large data structures: common uncompressed score files contain easily up to 250KB of text for a single A4 size page of piano solo music. When parsed into memory, this results in a relatively large XML tree. Since we envisioned use in real-time environments, we want to absolutely minimize any calculation time that is needed 'on-line' (during processing). There is no time to traverse an entire XML-tree to find the data that we need to access, as is commonplace in the visitor design pattern [9] as used in libmusicxml [10].

Because data that is 'scheduled' to occur at the same time, can be heavily dispersed throughout the file, a SAX-based approach to XML-parsing becomes difficult, and a DOM model is easier to handle. This made us decide to write our own data structure for the score, firmly based on the existing hierarchy of the format but with a few extra additions in functionality and precalculation of data. In the following sections of this paper, we will elaborate on the additions that we had to make to keep run-time calculation load as low as possible. To be able to parse XML files encoded in UTF-16, routines provided by the Unicode consortium can be used for the conversion of UTF-16 to UTF-8 data [11].

We set out to write a software library that could be used from real-time audio plug-in frameworks, as there are VST [12], AudioUnits [13] or RTAS [14]. In the end, we want to be able to use and manipulate a music score in a sequencer the same way we can use and manipulate an audio track. We need:

- quick access to all data.
- an easy method for timewise browsing through a score.
- easy extendability.
- cross-platform operation, documentation, testing ...

On top of that, in practice we need to adhere to several guidelines for real-time programming, amongst which some important ones are [15] :

- not allocating or deallocating memory
- avoiding denormalized floating point numbers

3. IT'S ABOUT TIME

3.1 Timestamping

Music scores are generally structured as follows: scores contain multiple parts or instruments, each of which consists of a series of measures that contain the notes. A score or a part can be subdivided into several sections, or within a measure, multiple voices may be separately encoded. We leave intermediate levels like these out here for clarity. Coda, segno and other repeat signs influence at what absolute time certain notes need to be played. This makes that music scores are rarely written to file in a way that is linear in time. The standard MIDI file format (SMF) comes close, but was never meant to be used for score encoding.

MusicXML stores timing information different than other formats: it only stores the note order and note length, and not the absolute position in the score at which the note occurs. This system was derived from the MuseData format [16]. In a real-time environment, we need absolute timestamps in order to know at any given time where we are in the score. These timestamps thus need to be calculated if they are not present. The quarter note as a unit of absolute time is the most convenient choice. This is portable across scores, whichever time signature they have, and across recordings, whichever tempo they are played in.

In MusicXML, in order to know at what absolute time in the score a specific note occurs, one needs to add up the length of all previous measures, and the previous notes in the same measure. This is a too intensive computation in real-time, therefore we need to precompute any absolute time values that we need in our application. The easiest way to do this is to keep track of a global absolute time value during parsing, and store a timestamp in every element that we encounter.

Having timestamps in the data structure has the following effect on run-time operations, with n the number of elements that need to have such a timestamp:

- worst-case time complexity to compute absolute time values in real-time decreases from O(n) to O(1),
- when a change occurs in the score, these values need to be updated throughout the score, introducing a penalty of O(n).

These figures assume that the notes in a measure are already sorted based on time, and that the timestamp of previous elements can be used to update the timestamp of later ones. Sorting on time is easy to accomplish by creating or overloading a comparison operator and running a generic sorting algorithm after parsing. In general, all time-modifying elements in the file format need to be processed during parsing to calculate timestamps for all notes. We found it handy to also store the time at which a certain timed element ends.

We need to add here that the increased complexity when changes occur to the score (measures or notes added, deleted or moved), rarely outweighs the benefits of having a timestamp on all elements for the applications that we envision. Fast access to useful data is the most important for us, and

in real-time applications, especially when user interactivity is in play, score access operations tend to occur thousands of times more often than score manipulation operations. If large-scale content manipulation of scores needs to be done very often, tools like XQuery are more fit for the job [17, 18].

3.2 Some notes on tempo and repetitions

In order to be able to accomodate easily for tempo changes or other elements that affect playback (rallentando, accelerando etc.), we tilted this performance timing information out of the score, and transferred it to a separate datastructure. An elegant solution is the use of a warping function, mapping playback time (in seconds) to score time (in quarter notes). The first derivative of this function is the equivalent of the local tempo at a certain time. Smooth increases or decreases in tempo can be modeled using splines. Cubic Hermite splines are a good choice since those can be calculated based only on two points and the tempo at these points. When constraints are applied to keep the function strict monotonously increasing, an inverse of that function exists which could eventually be used to encode information about performance, like lyricism.

Also affecting playback are structural elements, as there are repeat, coda, segno, ... Since these are usually limited in number and smooth transitions are not applicable here, this data can be stored easiest in a simple table, storing the timestamp values of all sections. When repeats should be skipped, one can just adapt this table, eliminating the need to do processing on the entire score data structure.

Going from playback time to score time (illustrated in fig. 1) then comes down to deciding with what time in quarter notes this corresponds, through the previously defined warping function. If sections are repeated or skipped, offsets to this time need to be added or subtracted, according to the information in the structure table. That way, we come to a corresponding timestamp value in the score structure itself, which can be used to access the necessary data.

Note that in this way, changes in tempo or performance information do not require updating the calculated timestamps in the score, which would be a rather costly operation as mentioned previously. As long as the score data itself remains unchanged, data related to performance and overall structure (repeats etc.) are kept outside the score itself and are quickly and easily accessible and modifiable. This is useful in applications needing some form of audio-to-score alignment [19].

4. STRUCTURE AND DESIGN

4.1 Indexing collections

A score contains a number of parts, a part contains a number of measures, a measure contains a number of notes: it

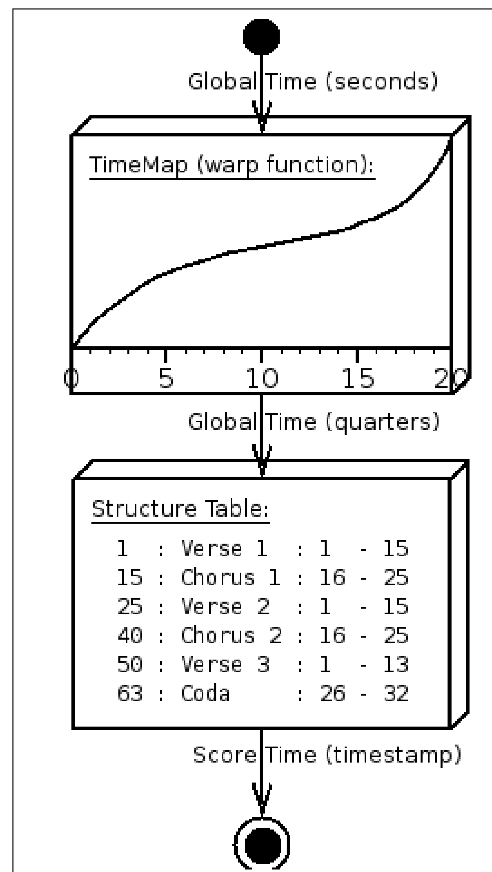

Figure 1. Flowchart: from real time to score timestamps

is clear that collections are an important feature in scores. These collections often need to be accessible in multiple ways. For fast direct access, a vector-like construction is ideal: accessing an element is then done in constant time. But a standard XML parser will store elements in a tree structure. In this tree, notes can be interleaved with other data (like harmony indications, certain dynamics elements), and are not necessarily ordered on timestamp.

To gain fast access to the most important data, we implement indices in the data structure, sorted on the criteria we wish to use for retrieval. We obtain fast access to notes by, during parsing, storing them in a map keyed on the timestamp that it has been given. When we need to search for a certain note occurring at a certain time, we can easily retrieve it (a map generally uses a binary search), and iterating over all notes in order can still be done in constant time using the best fit iterators.

Using sorted indexes to access certain structures introduces:

- searching for a certain element can be done in O(log(n)) instead of O(n).
- accessing a specific element takes O(log(n)) for maps and O(1) for arrays.
- when a change occurs, the indexes might need to be updated. The cost depends on the operation being

performed and the data structure used for the index. Insertion or deletion on a map takes only O(log(n)), but if resorting an array is needed, the penalty is O(n log(n)).

Implementing collections with these properties can be done by designing proper templates for them - [20] is an excellent resource on this. Templates are specifically meant to define operations and algorithms independent of type, and can thus be used for any type of element, while still making specializations based on type possible. Operator overloading is a useful programming trick to add additional accessing functionality.

4.2 Cursors and Listeners

For browsing through a score, an iterator system is most elegant. Most programmers are very familiar with this kind of interface. Preferably, a score iterator for real-time use corresponds to a position marker on a sequencer track, we'll therefore call them cursors. The cursor needs access to the tempo and structure information from the score. Multiple cursors on a single score are an asset, but to avoid discrepancies when multiple cursors are used, only one structure table and tempo function should exist for each score. The cursor's internal logic is then responsible for translating the time information from the sequencer to the relevant position in the score.

To keep track of the position in the score, a cursor keeps track of:

- its current position in quarter notes (timestamp)
- for each part, the current measure (this allows for multimetric music)
- for that measure, the next note that needs to be played.

In real-time software, a cursor is moved forward or backward by very small increments. Using the adapted internal score structure, moving the cursor forward comes down to:

- calculate the target timestamp of the cursor.
- for each part in the score, repeat the following until the next note's timestamp is scheduled after the target timestamp:
  - if the next note's timestamp is before the target timestamp, go to the next note
  - if there are no more notes in the current measure, go to the next measure

This could be even more simplified improved upon by collecting all notes of a score together and abstracting away the different measures and parts. But in practice, we found that we often needed to know at a certain moment which

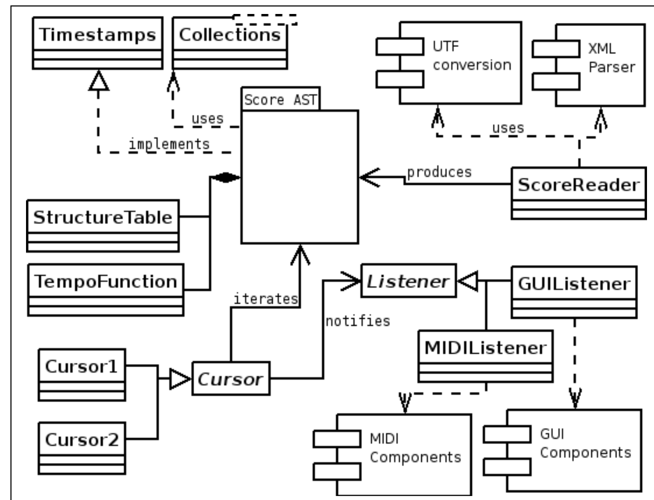

Figure 2. Simplified component diagram of the overall design

measure that a currently played note was in, and to what part it belonged. We consider it easier to keep track of this information in the cursor and request it from there, then to ignore it there but later have to obtain it through the score anyway. The number of measure crossings and the number of parts is also usually limited compared to the number of notes.

As mentioned earlier, intermediate levels in the score hierarchy, like voices or sections, can exist. Each supplementary level may add a nested loop to the aforementioned cursor moving algorithm, so it is beneficial to keep the number of different levels low. The trade-off between removing intermediate levels, keeping the original structure and accessing its information, and performance benefits or losses in different scenario's is difficult to make, and in the end the performance is highly dependent on the scores used: if a large amount of nested loops is kept, a single voice melody score will likely still be iterated over very fast, while traversing an orchestral score will go much slower. On the other hand, by eliminating too many loops we risk to need to introduce a large amount of intermediate variables in the cursor to keep track of all the necessary information, and updating and testing against these also takes time.

A cursor interface can be easily combined with the implementation of an observer pattern [9]. That way, when the cursor passes a note, it can notify another part of the software and trigger an event. In the observer pattern, one or more listeners can be attached to a cursor. The cursor then only needs to notify all of its listeners that an event was encountered. This system can be further generalized to enable notifications to be triggered at whatever element that is encountered in a score, and have them pass data for the listeners.

A simplified component diagram of the overall design of the score handling library is shown in fig. 2 .

5. IMPLEMENTATION

The library that we developed is part of a larger project, implemented as VST plugin [12]. It is meant to serve as prototyping platform for applications that use both audio and scores in real-time hosts. A screenshot is shown in fig. 3.

As a practical example, if we want simple playback, a cursor is put on the score at the beginning of the score, and incremented in small steps corresponding to the length of the audio data in the processing function of the plugin. When notes are encountered, an event is sent to a specific listener, that will take the note and some other necessary information to generate MIDI events out of it. The start events are sent back to the host, while the stop events are stored in a scheduler to be used when they are necessary. If another event should happen when a note, measure, crescendo, or whatever element in the score is encountered, a developer would only need to write his/her own listener, connect it to the cursor, and configure the cursor in such a way that it reacts to the element needed.

To display the score, we use exactly the same setup, only now the cursor is not moved over a timeline, but over a frame on the screen. A cursor is set on the position corresponding with the left viewport boundary as defined by the GUI's zoom data, scroll data (scrollbars) and window plane. When the screen is redrawn, the cursor is moved to the right viewport boundary. A listener is notified at each note that is encountered, which draws the note onto the window when necessary. There can be several thousands of these events triggered to draw a single score when zoomed out. Nevertheless we experienced that zooming and scrolling go fluently using this design, even if they force several redraws of the screen each second, each time generating a large flow of events.

While fig. 3 only shows a piano roll representation of the notes in the score, the cursor system combined with an implementation of the observer pattern, allows to create other visualizations as well. We might add dynamics information, incorporate or leave out information about the tempo, or leave the notes out and only show rests - the list goes on, and any custom visualization can be created based on the same system that is used for MIDI playback or setting parameters in a plugin. In our work we haven't gone that far though, and we will focus in the near future on the development of real-time plugins using scores for e.g. audio-to-score matching, rather than on visualization.

6. CONCLUSION

The use of music scores is not yet commonplace in many real-time applications - usually a MIDI representation is used as substitute. In this paper we have presented our efforts to create a library to enable the use of music scores file formats natively in such environments. We have singled out the design desicions that were taken in order to

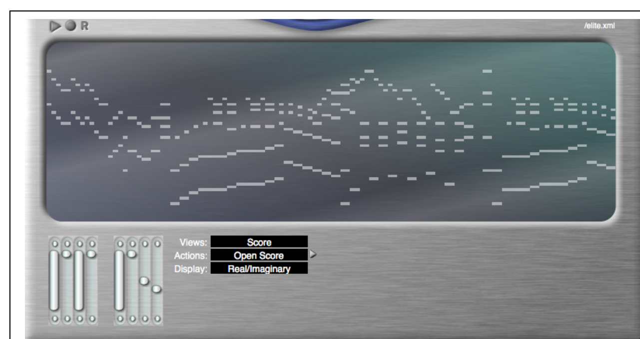

Figure 3. GUI of a prototype application, showing a basic piano roll representation of Joplin's Elite Syncopations.

come to a performant library. These considerations are applicable over the boundaries of file formats and computer languages.

In the applications that we envision, the benefits of fast information retrieval from the score and score browsing outweigh the slightly increased complexity on rarely used operations and the precomputation needed. The design considerations presented herein ensure that the computational load during processing is kept to a minimum.

7. ACKNOWLEDGMENTS

This work was funded by a specialization grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

8. REFERENCES

[1] MIDI Manufacturers Association, *Complete MIDI 1.0 Detailed Specification, 2nd ed.*, 2001

[2] Recordare LLC: "MusicXML definition, version 2.0," *Available at http://www.recordare.com/xml.html*

[3] M. Good: "Lessons from the Adoption of MusicXML as an Interchange Standard," *Proc. XML 2006 Conference*, 2006.

[4] P. Roland: "The music encoding initiative (MEI)," *Proc. 1st International Conference on Musical Applications Using XML*, pp. 55–59, 2002.

[5] P. Roland and J.S. Downie "Recent Developments in the Music Encoding Initiative Project: Enhancing Digital Musicology and Scholarship," *Proc. 19th Joint International Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing (Digital Humanities 2007)*, pp. 186–189, 2007.

[6] P. Bellini and P. Nesi: "WEDELMUSIC format: an XML music notation format for emerging applications," *Proc. 1st International Conference on Web De-*

livering of Music (WEDELMUSIC 2001), pp. 79–86, 2001.

[7] P. Bellini, P. Nesi and G. Zoia: "Symbolic music representation in MPEG," *IEEE Multimedia* , Vol. 12, No. 4, pp. 42–49, 2005.

[8] D. Baggi and G. Haus: "The Concept of Interactive Music: the New Standard IEEE P1599 / MX," *Proc. 2nd International Conference on Semantic and Digital Media Technologies, (SAMT 2007)*, pp. 185–195, 2007

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, 1994

[10] GRAME Computer Music Research Lab: "libmusicxml", *Available at http://libmusicxml.sourceforge.net/*

[11] Unicode Inc.: "Conversions between UTF32, UTF-16, and UTF-8", *Available at http://www.unicode.org /Public/PROGRAMS/CVTUTF/*

[12] Steinberg Media Technologies GmbH: "Virtual Studio Technology," *Available at http://www.steinberg.net*

[13] Apple Inc.: "Audio Unit Programming Guide" *Available at http://developer.apple.com /documentation/MusicAudio/Conceptual /AudioUnitProgrammingGuide /AudioUnitProgrammingGuide.pdf*, 2007

[14] Avid Technology, Inc: "Real Time Audio Suite" *Available at http://www.digidesign.com*

[15] L. de Soras: "Denormal numbers in floating point signal processing applications" *Available at http://ldesoras.free.fr/*, 2005

[16] E. Selfridge-Field: *Beyond MIDI: the handbook of musical codes*, MIT Press, Cambridge, MA, 1997

[17] World Wide Web Consortium (W3C): "XQuery 1.0: An XML Query Language - W3C Recommendation 23 January 2007" *Available at http://www.w3.org/TR/xquery/*

[18] J. Ganseman, P. Scheunders and W. D'haes: "Using XQuery on MusicXML Databases for Musicological Analysis" *Proc. 9th International Conference on Music Information Retrieval (ISMIR)*, Philadelphia, PA, USA, 2008.

[19] H. Heijink, P. Desain and L. Windsor: "Make Me a Match: an evaluation of different approaches to Score-Performance Matching," *Computer Music Journal* , Vol. 24, No. 1, pp. 43–56, 2000.

[20] D. Vandevoorde and N.M. Josuttis: *C++ Templates: the complete guide*, Addison-Wesley, Boston, 2002