IMUTUS SCORE PROCESSING COMPONENTS

D.Fober, S.Letz, Y.Orlarey
Grame - Centre national de création musicale
{fober, letz, orlarey}@grame.fr

ABSTRACT

IMUTUS is an IST European project that aims at the development of an open platform for training students on the recorder. Among the results of the project are two open source libraries for music representation and graphic notation: the MusicXML library, intended to support the MusicXML format and to provide music notation exchange capabilities, and the GUIDO library that provides a powerful graphic score engine. This paper introduces both libraries and describes their use in IMUTUS system.

1. INTRODUCTION

The IMUTUS (Interactive MUsic TUition System) project aims at the development of an open platform for training students on the recorder. It is primary focused on performance skills but since it addresses beginners, it also includes fundamentals of music and games which are combined to practicing sessions to create a complete learning setting. Due to this pedagogical orientation, IMUTUS is very demanding in term of music representation.

Although there is a long history of systems capable of graphically representing music in common music notation format, very few systems have reached maturity. The toolbox approach for graphic score layout has been investigated very early [1]. The *Common Music Notation* system [10] may be considered as the best achievement; more recently, the *Expressive Notation Package* (ENP) [7] introduced another promising approach; both systems are Lisp based environments. Another solution consists in compilers producing music sheets from a textual music description. Among them is MusiXTEX: a set of TEX macros to typeset music notation. Lilypond [8]; a more recent initiative; is an open source software partially implemented in the language Scheme. Both systems generate PostScript, EPS or PDF files.

Music representation is another critical issue for music applications: it faces the music complexity as well as the diversity of needs of the various tools that operate on it. Music representations intended for playing, for notation or for information retrieval are generally optimized in different ways. This is probably the main reason that has led to a proliferation of languages and formats for music description [2] [11] [5].

Music applications that want to include graphic score layout capabilities have to solve both the music representation problem and the complexity of the music layout process. From layout viewpoint, none of the existing resources are ready to be embedded into a standalone application. Concerning the music representation for notation, although a wide choice of formats exists, only a few resources are available to support them. Facing these critical issues, the IMUTUS project developed two open source projects, aiming at making up for this lack of components:

- the MusicXML library, intended to support the MusicXML format [4] and to provide music notation exchange capabilities,
- the GUIDO Engine library that provides a powerful graphic score engine, based on the GUIDO Music Notation format [6].

The next sections are intended to give an overview of these libraries and to highlight the essential points of interest for developers. Finally, a concrete example of these libraries capabilities is given with a description of the IMUTUS Score Processing components.

2. THE MUSICXML LIBRARY

The MusicXML format has been introduced in 2000. The MusicXML library is a portable C++ library defined very close to this format. It includes the necessary to read, write, build, browse and modify a MusicXML music representation. It also provides tools to support other formats: export to the GUIDO format is already included. The MusicXML library is an open source project covered by the GNU LGPL license and hosted on SourceForge [1].

2.1. Brief overview of the MusicXML Format

MusicXML is a XML format primarily based on two academic music formats: the MuseData and the Humdrum formats [11]. It organizes the music into a header followed by the core music data. The header contains basic metadata about the music score, such as the title and composer. It also contains the part-list, describing all the parts or instruments in the music score. The core music data are organized as *partwise* or *timewise* data:

- partwise data are organized into parallel parts containing a sequence of measures,

- timewise data are organized into sequence of measures containing parallel parts.

[1] The MusicXML Library: http://libmusicxml.sourceforge.net

A *partwise* measure contains elements grouped under the `music-data` entity. It covers the following purposes:

- Music score description: most of the elements are intended to enumerate the graphic components of a music score. The `note` element is the main one but a measure contains also attributes like `key` or `time` signatures, or `direction` elements (like `dynamics`) attached to a part or to the overall score.
- Time description: using elements to move the time backward (`backup`) or forward (`forward`).
- Playback parameters: the `sound` element allows for tempo, dynamics description, but also for sound control including MIDI instrument assignment, and for structural description (da capo, segno, dal segno...).
- Miscellaneous elements like XLink support (`link`, `bookmark`), printing parameters or music analysis elements (`harmony`, `grouping`).

The `note` element is central to the music description. It includes all the necessary for an acurate graphic rendering of all the signs attached to it (`accidental`, `dot`, `stem`, `beaming`, `articulation`, `ornament`, `slur`, `lyrics`...)

2.2. The MusicXML library design

The MusicXML library is a set of platform independent C++ classes. It has been designed very close to the MusicXML format but connection between the classes and the XML elements is not a one-to-one relation: for simplification, a class may cover several MusicXML elements. For example, MusicXML defines `dynamics` as separate elements while the library defines a single object. Apart one exception detailled in section 2.2.3, the library design corresponds to the MusicXML DTDs which may serve as library documentation as well.

2.2.1. Smart pointers for memory management

Each object that describes a MusicXML element is handled using *smart pointers*. Smart pointers simplify the programmer task by automatically freeing the objects when they are not any more used.

2.2.2. Element to class correspondence

A clear relationship between objects and MusicXML elements has been preserved all along the library, for example, the root of the music representation is the MusicXML `score` element, implemented by the `TScore` object and handled as a `SScore` smart pointer within the framework.

2.2.3. Chords

The library includes a `TChord` object that differs from the MusicXML `chord` element, part of the `full-note` entity and included in a `note` to indicate that the note is an additional chord tone with the preceding note. The `TChord` class includes a container to group all the notes of a chord.

2.3. Browsing the representation

The MusicXML representation is a tree which root is a *timewise* or *partwise* score. All the objects defining the score support the *visitor* design pattern [3] and accept a `TScoreVisitor` as the base class of the visitor design. Visitors may be implemented or derived for various purposes: browsing the music representation to collect information, to edit the score or to convert it to another format. Several *visitors* are included in the library, providing different ways for traversing the music structure or maintaining information along the *visite* process. It includes:

- a `TRolledVisitor`: provides a straightforward traversing of the score,
- a `TUnrolledVisitor`: *unrolls* repeats and structural jumps like 'da capo' or 'to coda'
- a `TMidiContextVisitor`: maintains a context for MIDI generation.

The code sample below implements a raw transposition. It illustrates how the visitor mechanism allows to focus on the elements of interest .

```
class Transposer : public TRoutedVisitor {
   int   fInterval;         // the transposing interval
   public:
      Transposer(int interval) : fInterval(interval) {}
      virtual ~Transposer() {}
      void visite (SNote& n) { n->pitch()+= fInterval;}
};
```

Figure 1. A *transposer* visitor

3. THE GUIDO ENGINE LIBRARY

The GUIDOLib project aims at the development of a generic, portable library for the graphical rendering of musical scores. The library is based on the GUIDO Music Notation format [6] as the underlying data format. It is an open source project covered by the GNU LGPL license and hosted on SourceForge [2]. The project has started in December 2002, based on the source code of the GUIDO NoteViewer mainly developed by Kai Renz [9].

3.1. The GUIDO Music Notation

The GUIDO Music Notation format (GMN) is a general purpose formal language for representing score level music in a platform independent plain text and human readable way. Its design concentrates on general musical concepts (as opposed to graphical features). A key idea of the GUIDO approach is adequacy which means that simple musical concepts should be represented in a simple way and only complex notions should require complex representations.

3.2. The GUIDO Engine

The GUIDO Engine operates on a memory representation of the GMN format: the GUIDO Abstract Representation (GAR). This representation is transformed step by step to produce graphical score pages [9]:

[2] The GUIDOLib home page: http://guidolib.sourceforge.net

- GAR to GAR transformation represents the logical layout transformation: part of the layout (such as beaming for example) may be computed from the GAR as well as expressed in GAR,
- the GAR is next converted into a GUIDO Semantic Normal Form (GSNF), which is a canonical form such that different semantically equivalent expressions have the same GSNF.
- the GSNF is finally converted into a GUIDO Graphic Representation (GGR) that contains the necessary layout information and is directly usable to draw the music score. This final step includes notably spacing and page breaking algorithms.

Note that although the GMN format allows for accurate music formatting, the GUIDO Engine provides powerful automatic layout capabilities.

3.3. Main library services

3.3.1. Score layout

The library provides functions to parse a GMN file and to create the corresponding GAR and GGR. GAR and GGR are referenced by opaque handlers, used as arguments of any function that operates on a score. For example: `GuidoParseFile` provides conversion of a GMN file into a GAR handler and the `GuidoAR2GR` function converts it into a GGR handler, which may be next used to draw the score using the `GuidoOnDraw` function.

3.3.2. Browsing music pages

Result of the score layout is a set of pages which size may be dynamically changed according to an application or a user needs. The library provides the necessary to setup the page size, to query a score pages count, the current page number or to retrieve the page number corresponding to a given music date.

3.3.3. The GUIDO Factory

The GUIDO Engine may be feeded with computer generated music using the GUIDO Factory. The GUIDO Factory API provides a set of functions to create a GAR from scratch.

A music score dynamic construction is very close to the textual GUIDO description: the Factory API handles GAR objects that have a one to one relationship with the notation format. Once the score has been dynamically build, a call to `GuidoFactoryCloseMusic()` returns a GUIDO handler to a GAR, which may be next converted into a GGR handler using `GuidoAR2GR()`.

3.3.4. GUIDO Mappings

Along with the GGR, the Guido Engine maintains a tree of graphical musical elements for each page of the score (figure 2). Each element has a bounding box, a date and a duration. The GUIDO library API includes a set of functions to browse this tree and to retrieve elements by type, date or position.

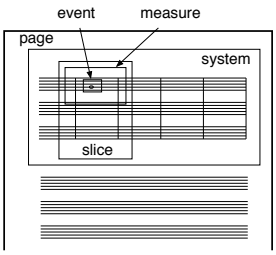

Figure 2. The score graphic elements tree.

4. IMUTUS SCORE PROCESSING

MusicXML and GUIDO libraries have been used to design an extended score viewer and a simple score editor. Both components are ready to be embedded into any application by the way of a *Score Processing library*.

4.1. The Extended Score Viewer

4.1.1. General architecture

The score viewer is organized in two main modules :

- a players module: in charge of the audio and MIDI rendering. It synchronizes the graphic module to display the current position in real time.
- a graphic module: in charge of producing and merging several graphic representations in order to draw what is actually seen on the screen according to the current position in the score. It handles also the mouse clicks and selections on the score and may trigger the audio or MIDI playback.

The graphic module is at the heart of the system. All the information linked to pedagogical issues are related to the corresponding musical material and displayed on the score: it represents pedagogical annotations or graphical signs to highlight a specific section. This information is attached to a given exercise or dynamically computed by the system. Additional information a student may require is also related to the music score and therefore is obtained by interacting with the score: for example, a student can listen to a given note by clicking on this note. The score viewer may therefore be viewed as the user interface of the players mentioned above.

4.1.2. Segment mappings to relate graphic and sound

The role of a *segment mapping* is to relate *time based resources* defining correspondencies between *segments* of resources. In the framework of IMUTUS such mappings are typically used to link graphical positions, musical positions and audio positions. IMUTUS time based resources use different time representations: absolute time in frames for the audio player, musical time expressed in bar/beat/unit

for the MIDI players, graphic time in page number and x, y coordinates for the graphic module.

A *segment mapping* provides the required information to define *time convertion* functions. Appropriate links to synchronize all the related time based resources are created by combining such mappings (figure 3). This design is notably supported by the GUIDO library mappings API.

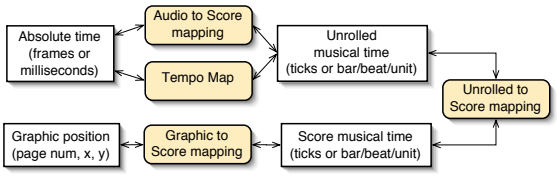

Figure 3. Time to graphic mappings.

4.2. The Simple Score Editor

The simple score editor is basically a score viewer extended with editing capabilities. It provides a simple way to write a short score and includes all the automatic layout capabilities to help in writing a well formatted score.

4.2.1. Underlying concepts

A score is viewed as a collection of measures containing symbols (notes or rests). The score editor basic concepts are reduced to two simple ideas:

- symbols are considered similarly to alphabetical characters in a text editor: they are carrying attributes that may be compared to characters styles (bold, italic...) and are edited similarly.
- time space within a measure is always consistent, it pre-exists to symbols and plays a role similar to magnetic grids in drawing applications.

4.2.2. Measure time space

Let's consider an empty 4/4 measure: such a measure is actually including 4 virtual quarter notes in place of 4 real notes. This approach has some important repercussions for the design as well as from pedagogical viewpoint:

- a measure is internaly always complete,
- since a measure is always complete, there is no *insert* nor *delete* operation: new symbols replace existing virtual symbols and symbols to be deleted are tranformed into virtual symbols.
- since a new symbol can only replace a virtual symbol, its time location will be that of the virtual symbol. Therefore, place and size of the virtual symbols operate like a magnetic grid.

From pedagogical viewpoint, the idea of magnetic grid may be used to guide the student in the music writing process. The system also allows to write scores with "*holes*" (i.e. with measures that appear to be incomplete) and to design *fill-in* like games for example.

5. CONCLUSION

Combined use of the MusicXML and GUIDO libraries may constitute a simple and efficient solution for a large set of music applications. They provide music notation interchange capabilities and relieve the programmer of the tricky task of the score layout. Both solutions are freely available to developers on the SourceForge public repository.

ACKNOWLEDGEMENTS

This work was supported by the European Community under the Information Society Technology (IST) RTD programme, contract IST-2001-32270

6. REFERENCES

[1] Assayag G., Timis D. "A ToolBox for Music Notation." *Proceedings of the International Computer Music Conference*, ICMA, 1986, pp.173-178

[2] Dannenberg R. B. "Music Representation Issues, Techniques and Systems" *Computer Music Journal*, 17(3), MIT Press 1993, pp. 20-30

[3] E Gamma, R Helm, R Johnson, J Vlissides "Design Patterns - Elements of Reusable Object-Oriented Software" Addison-Wesley 1999

[4] Good M. "MusicXML for Notation and Analysis" in *The Virtual Score*, ed. W. B. Hewlett and E. Selfridge-Field, MIT Press, 2001, pp.113-124

[5] Hewlett W. and Selfridge-Field E. eds "The Virtual Score: Representation, Retrieval, Restoration" *Computing in Musicology* 12, MIT Press, 2001

[6] H. Hoos, K. Hamel, K. Renz, J. Kilian "The GUIDO Music Notation Format - A Novel Approach for Adequately Representing Score-level Music" *Proceedings of the ICMC'98*, ICMA, 1998, pp.451-454

[7] M Kuuskankare, M Laurson "ENP - Music Notation Library based on Common Lisp and CLOS" *Proceedings of the International Computer Music Conference*, ICMA, 2001, pp.131-134

[8] Han-Wen Nienhuys, Jan Nieuwenhuizen "LilyPond, a system for automated music engraving" *Proceedings of the XIV Colloquium on Musical Informatics*, Firenze, Italy, 2003

[9] Renz Kai "Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation" *PhD Thesis*, Technischen Universität Darmstadt, 2002

[10] Schottstaedt B. "Common Music Notation" in *Beyond MIDI, The handbook of Musical Codes.*, Selfridge-Field E. ed. MIT Press, 1997 pp.217-222

[11] Selfridge-Field E. ed "Beyond MIDI, The handbook of Musical Codes" MIT Press, 1997