OPEN SOURCE TOOLS FOR MUSIC REPRESENTATION AND NOTATION

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

ABSTRACT

Although numerous systems and applications exist for music representation and graphic notation, there are few resources available to developers. We present two open source projects that aim at making up for this lack of components: 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, based on the GUIDO music notation format.

1. INTRODUCTION

There is a long history of systems capable of graphically representing music in common music notation format. Commercial music publishing software exist for more a decade and actually provide sophisticated but complex solutions for music engraving. Along these *closed* solutions, the toolbox approach has been investigated very early [1]. However, very few systems have reached maturity: the *Common Music Notation* system [2] could be considered as the best achievement; more recently, the *Expressive Notation Package* (ENP) [3] introduced another promising approach; both systems are Lisp based environments.

Another solution consists in designing compilers for producing music sheets from a textual music description. MusiXTEX [4] is among these tools: it is a set of TEX macros to typeset music notation. Since MusiXTEX is powerful but hard to learn, preprocessors such as PMX and M-Tx have been designed to facilitate music input and layout. A more recent initiative is Lilypond [9], an open source software partially implemented in the language Scheme, its input music representation format is more simple and intuitive, it includes automatic layout capabilities. Both systems produce PostScript, EPS or PDF files.

Music representation is another critical issue for music applications: it faces the music complexity on one hand and on the other hand, 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 [5] [6] [7]. These formats are too numerous to be discussed here. An impressive work of Gerd Castan provides a good and updated source of information online [8] concerning music notation formats and including related tools.

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, many applications could benefit of existing knowledge but none of the actual corresponding resources are really usable: systems such as CMN or ENP are Lisp based which impose a constraint generally not acceptable by many applications. Open source resources such as Lilypond are also based on non standard development technologies. Concerning the music representation for notation, although a wide choice of formats exists, only a few resources are available to support these formats. We present two open source libraries that aim at making up for this lack of components:

- the MusicXML library, intended to support the MusicXML format [10] and to provide music notation exchange capabilities,
- the GUIDO library that provides a powerful graphic score engine, based on the GUIDO music notation format [11] [12].

Combined use of these libraries may constitute a simple and efficient solution for a large set of music applications. The next sections are intended to give an overview of these libraries and to highlight the essential points that a developer may want to know to use them.

2. THE MUSICXML LIBRARY

The MusicXML format has been introduced in 2000. It is actually known as the best music notation interchange format. 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 aims at providing tools to export or import other music representation formats: export to the GUIDO format is already supported. The MusicXML library is an open source project covered by the GNU LGPL license [13] 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 format [15] and the Humdrum format [14]. It organizes the music into a header followed by the core music data. The score header contains some basic metadata about a musical score, such as the title and composer. It also contains the part-list, which lists all the parts or instruments in a musical score. The core music data may be 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.

A MusicXML part may be viewed as a music part assigned to a given instrument. There is no correspondence between staff and part: staff assignment is made note by note and a part may include several staves (in case of piano or organ for example).

A MusicXML partwise measure contains elements grouped under the `music-data` entity. These elements cover 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 most important one but the measure contains also measure specific attributes like `clef`, `key` or `time` signatures, `transpose` indications or `barline` description, as well as `direction` elements (like `metronome`, `dynamics`) attached to a part or the overall score.

- time description using elements to move the time backward (`backup`) or forward (`forward`).

- music analysis using elements like `harmony` or `grouping` .

- playback parameters: the `sound` element allows for tempo, dynamics description, but also for sound control including MIDI instrument assignment, and for structural description (dacapo, segno, dalsegno, coda, tocoda).

- miscellaneous elements like XLink support (`link` and `bookmark` elements) or printing parameters (`print` element).

The `note` element is central to the music description. It may be a *cue* note, a *grace* note or a *regular* note. All of them share common elements which are pitch, chord, and rest information. Unpitched elements are used for unpitched percussion, speaking voice, and other musical elements lacking determinate pitch. A `note` element includes all the necessary for an acurate graphic rendering of all the signs attached to it. It covers the graphic `type` corresponding to the symbolic note duration (whole, half, quarter note...), possible `accidental` and `dot`, `stem` and `beaming` information, the graphic `notehead` shape (triangle, diamond, square...), `staff` assignment, `notation` elements such as `articulation`, `ornament`, `slur`, `tied`, `lyrics`...

Redundant graphic and sound information may live together, for example: a `tuplet` element is present when a tuplet is to be displayed graphically in addition to the sound data provided by the `time-modification` element; or a `tie` element indicates sound while the `tied` element indicates notation.

2.2. The MusicXML library design

The MusicXML library has been developped in C++, with a great care of preserving platform independence. It has been designed very close to the MusicXML format. However, connection between the classes and the XML elements is not a one-to-one relation: for simplification, a class may include several MusicXML elements, provided that these elements are not reused anywhere else. For example, MusicXML defines `dynamics` as separate elements while the library defines a single object. Apart one exception detailled below (section 2.2.5), the library design corresponds to the MusicXML DTDs which may serve as library documentation as well.

2.2.1. Memory management

Each object that describes a MusicXML element is handled using *smart pointers*. A smart pointer is in charge of maintaining an object reference count by the way of pointers operators overloading and of automatically freeing the object when the reference count drops to zero. It supports class inheritance and conversion whenever possible. The implementation prevents direct call to objects constructors; instead of constructor call, a MusicXML object is created directly embedded within a smart pointer using a friend method.

2.2.2. Common entities representation

MusicXML defines entities that are common across multiple component DTDs. Many of them are intended to describe graphic layout like position, placement or orientation.

These entities are defined as separate objects. A class is defined to aggregate the corresponding properties to objects that require them. For example, a `TOrientation` class is defined to describe the `orientation` entity; next an `Orientable` class is intended to provide the corresponding properties to derived classes. MusicXML ele-

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

ments that carry the `orientation` entity have to derive the `Orientable` class.

MusicXML also makes use of entities to enumerate values like the `start-stop` or the `yes-no` examples below:

```
<!ENTITY % start-stop "(start | stop)">
<!ENTITY % yes-no "(yes | no)">
```

For these entities, the library provides conversion classes which names are closely related to the entity names (like a `YesNo` class) that:

- define a type for the corresponding enumeration

- provide conversion methods from/to integer and textual representation. These methods are generally named `xml` and overloaded to access both the integer and string representation.

2.2.3. Common types definition

Many of the MusicXML elements are defined as an alternative bewteen a set of elements. These alternative may be viewed as an inheritance relationship. Each time it has been convenient to do so, this relationship has been made explicit using a specific type, defined as an abstract class and intended to cover the corresponding elements:

- `TMusicData` defines a common type for all the elements covered by the `music-data` entity.

- `TDirectionTypeElement` defines a type for all the elements of a `direction-type` element,

- `TNotationElement` defines a type for all the elements of a `notation` element,

- `TPartListElement` defines a type for all the elements of a `part-list` element.

2.2.4. Music data representation

Most of the objects of the library have direct MusicXML element counterpart. These objects carry a name in the form `Txxx` where `xxx` corresponds to an element name. For example: the `TBarline` object corresponds to the `barline` element, the `TNote` object corresponds to the `note` element, etc... The `TChord` object is an exception since it has no MusicXML counterpart.

Because the `Txxx` objects are embedded into *smart pointers*, they all have a `Sxxx` form which represents the corresponding smart pointer. For example, the root of the music representation is the MusicXML `score` element, which corresponds to the `TScore` object, which is handled as a `SScore` smart pointer within the frameworks.

2.2.5. Chords

The library includes a `TChord` object that differs from the MusicXML `chord` element. The MusicXML `chord` element is part of the `full-note` entity and is included in a `note` to indicate that the note is an additional chord tone with the preceding note. The `TChord` class includes a container intended to group all the notes of a chord. Although the two representations may be deduced from each other, they are semantically different. The library representation is intended to facilitate chords handling as a single object.

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 [16] and accept a `TScoreVisitor` as the base class of the visitor design.

To preserve the choice of different strategies for the traversing the music representation structure, traversing has not been implemented in the tree components: it is assumed that it's the visitor responsability. A visitor that implements a basic traversing of the music structure is included in the library: the `TRoutedVisitor`. It propagates the *visit* to the subclasses in the following order:

- subclasses representing attributes are called first,

- subclasses representing elements are called in the MusicXML defined order.

Visitors may be implemented for various purposes: browsing the music representation to collect information (for analysis purpose for example), to modify the representation or to convert it to another format. MusicXML export to the GUIDO format is achieved using the visitor mechanism: the `TXML2GuidoVisitor` is a visitor that transform a MusicXML tree into a *guido tree*.

3. THE GUIDO ENGINE LIBRARY

The GUIDOLib project aims at the development of a generic, portable library and API for the graphical rendering of musical scores. The library is based on the GUIDO Music Notation format [11] [12] as the underlying data format. It is an open source project covered by the GNU LGPL license [13] and hosted on SourceForge [2]. The project has started in December 2002, mainly based on the source code of the GUIDO NoteViewer developed by Kai Renz [17]. It also includes various resources to support other music representation formats. The MusicXML format is supported by the way of the library above.

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. It is based on a conceptually simple but powerful formalism: its design concentrates on general musical concepts (as opposed to graphical features). A key feature of the GUIDO design is adequacy which means that simple musical concepts should be represented in a simple way and only complex notions should require complex representations. This design is reflected by three specification levels: the basic, advanced[3] and extended[4] GUIDO specifications.

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

[3] partially available

[4] not yet publicly available

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. Two kinds of processing are first applied to the GAR:

- GAR to GAR transformations which represents a 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 converted into a GUIDO Semantic Normal Form (GSNF). The GSNF is a canonical form such that different semantically equivalent expressions have the same GSNF.

This GSNF is finally converted into a GUIDO Graphic Representation (GGR) that contains the necessary layout information and is directly used to draw the music score. This final step includes notably spacing and page breaking algorithms [17].

Note that although the GMN format allows for precise music formatting (in advanced GUIDO), 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 which are used as arguments of any function that operates on a score. For example: `GuidoParse (const char * filename)` provides conversion of a GMN file into a GGR handler returned as the function result. This handler may be next used to draw the score using the `GuidoOnDraw` function.

A typical code to draw a score from its GMN description is given by the figure 1.

3.3.2. Score pages access

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 tochange the page size, to query a score pages count, the current page number or the page number corresponding to a given music date. It also allows to change the current page (note that only the current page is drawn by the `GuidoOnDraw` function).

3.3.3. Engine settings

Score layout algorithms are controlled by a set of parameters which are global to the GUIDO engine. The library provides an API to query and modify these parameters. It includes optimal page fill control, springs and space force control, systems distance and systems distribution.

3.3.4. 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 and to convert it into a GGR. The GUIDO Factory is a state machine that operates on implicit current elements: for example, once you open a voice (`GuidoFactoryOpenVoice()`), it becomes the current voice and all subsequent created events are implicitly added to this current voice.

The GUIDO Factory state includes the current score, voice, chord, note (or rest) and tag. Some elements of the factory state reflects the GUIDO formal specification; unless otherwise specified, new notes will implicitly carry the current duration and octave.

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, directly usable with `GuidoFactoryMakeGR()`, which returns a GUIDO handler to a GGR, directly usable with the main services of the library. Logical layout is performed before returning the GAR handler and graphical layout is performed before returning the GGR handler.

4. CONCLUSION

The GUIDO library is an original solution that may provide score layout capabilities directly embedded into an application. The MusicXML library provides scores interchange capabilities. Both are cross-platform libraries and may collaborate to constitute a powerful and portable solution.

5. REFERENCES

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

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

[3] MIKA KUUSKANKARE AND MICHAEL LAURSON, *ENP - Music Notation Library based on Common Lisp and CLOS* Proceedings of the International Computer Music Conference, ICMA, 2001, pp.131-134

[4] DANIEL TAUPIN, ROSS MITCHELL, ANDREAS EGLER, *MusiXTEX Using TEX to write polyphonic or instrumental music.* available at http://icking-music-archive.org/

```
void DrawGMNFile (char * filename)  {

    // for simplicity , we use the cross-platform GDevicePostScript
    GDevicePostScript dev (210, 297, "outfile.eps", "", "guido.eps");
    // data structure for engine initialization, uses fonts "guido2" and "times"
    GuidoInitDesc gd = { &dev, 0, "guido2", "times" };
    GuidoOnDrawDesc desc;              // declare a data structure for drawing
    GuidoInit (&gd);                   // Initialise the Guido Engine first

    // and parse the GMN file to get a GGR handle directly stored in the drawing struct
    desc.handle = GuidoParse (filename);
    desc.hdc = &dev;                    // we'll draw on the postscript device
    desc.page = 1;                      // draw the first page only
    desc.updateRegion.erase = true;     // and draw everything
    desc.scrollx = desc.scrolly = 0;    // from the upper left page corner
    desc.zoom = 1;                      // no zoom
    desc.sizex = desc.sizey = 0;        // do not override the zoom factor parameter

    GuidoOnDraw (&desc);                // draw the score to the PostScript device
}
```

Figure 1. Example of code to draw a score.

[5] ROGER B. DANNENBERG, *Music Representation Issues, Techniques and Systems.* in Computer Music Journal, 17(3), MIT Press 1993, pp. 20-30

[6] SELFRIDGE-FIELD E. ed, *Beyond MIDI, The handbook of Musical Codes.* MIT Press, 1997

[7] HEWLETT, W. AND SELFRIDGE-FIELD, E. eds, *The Virtual Score: Representation, Retrieval, Restoration.* Computing in Musicology 12, MIT Press, 2001

[8] GERD CASTAN, *Music Notation Formats* http://www.music-notation.info/

[9] HAN-WEN NIENHUYS AND JAN NIEUWENHUIZEN, *LilyPond, a system for automated music engraving.* Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003), Firenze, Italy, May 2003

[10] GOOD, M., *MusicXML for Notation and Analysis.* In The Virtual Score, Selfridge-Field E, ed. W. B. Hewlett and E. Selfridge-Field (Cambridge, MA, 2001), MIT Press, 2001, pp.113-124

[11] H. H. HOOS, K. A. 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

[12] H. H. HOOS, K. A. HAMEL, *The GUIDO Music Notation Format Specification - Version 1.0, Part 1: Basic GUIDO.* Technical Report TI 20/97, Technische Universitat Darmstadt - Germany, 1997

[13] *GNU Lesser General Public License.* http://www.fsf.org/licenses/lgpl.html

[14] DAVID HURON, *Humdrum and Kern: Selective Feature Encoding.* in Beyond MIDI, The handbook of Musical Codes, Selfridge-Field E. ed, MIT Press, 1997, pp.376-401

[15] WALTER B. HEWLETT, *MuseData: Multipurpose Representation.* in Beyond MIDI, The handbook of Musical Codes, Selfridge-Field E. ed, MIT Press, 1997, pp.402-447

[16] E. GAMMA, R.HELM, R.JOHNSON, J.VLISSIDES, *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley 1999

[17] RENZ KAY, *Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation.* PhD Thesis, Technischen UniversitŁt Darmstadt, 2002