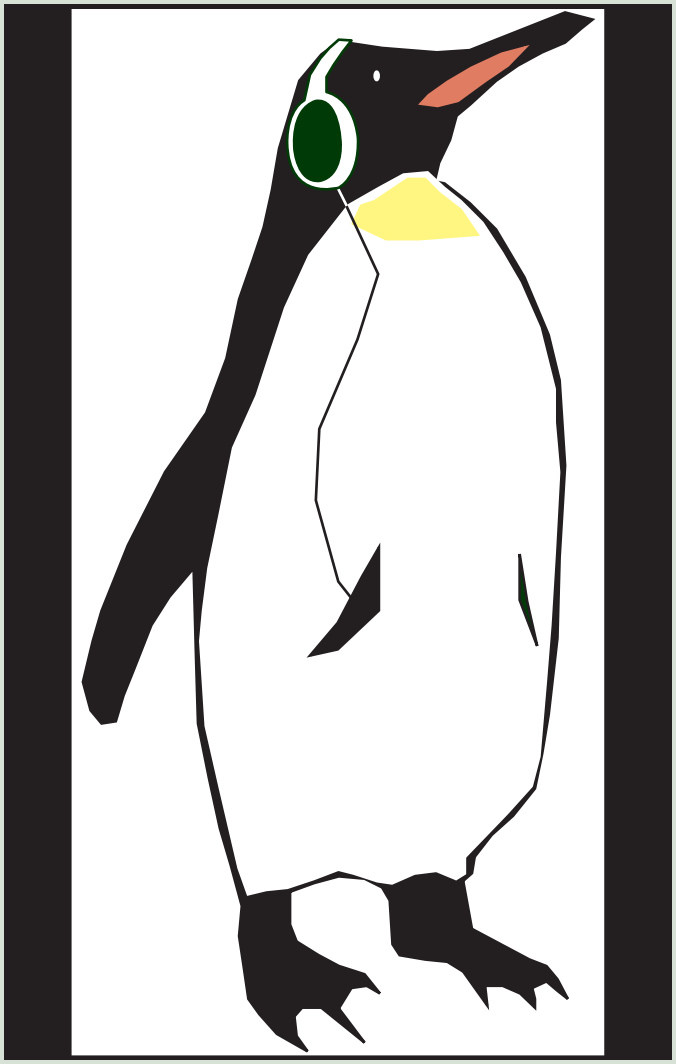


LINUX AUDIO

CONFERENCE

BERLIN

Lectures/Demos/Workshops
Concerts/LinuxSoundnight

5

PROCEEDINGS

TU-Berlin 22.-25.03.07

www.lac.tu-berlin.de

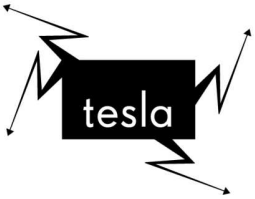
Published by:
Technische Universität Berlin, Germany
March 2007

www.lac.tu-berlin.de

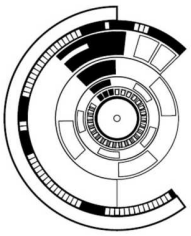
Credits:
Cover design and logos: Alexander Grüner
Layout: Marije Baalman
Typesetting: LaTeX

Thanks to:
Vincent Verfaille for creating and sharing the DAFX'06 "How to make your own Proceedings" examples.

Printed in Berlin by TU Haus-Druckerei — March 2007



MaerzMusik
Festival für aktuelle Musik **16. bis 25.03.2007**



Preface

The International Linux Audio Conference 2007, the fifth of its kind, is taking place at the Technische Universität Berlin.

We are very glad to have been given the opportunity to organise this event, and we hope to have been able to put together an interesting conference program, both for developers and users, thanks to many submissions of our participants, as well as the support from our cooperation partners.

The *DAAD - Berliner Künstlerprogramm* has supported us by printing the flyers and inviting some of the composers. The *Cervantes Institute* has given us support for involving composers from Latin America and Spain. *Tesla* has been a generous host for two concert evenings. Furthermore, *Maerz-Musik* and the *C-Base* have given us a place for the lounge and club concerts.

The *Seminar für Medienwissenschaften* of the Humboldt Universität zu Berlin have contributed their Signallabor, a computer pool with 6 Linux audio workstations and a multichannel setup, in which the Hands On Demos are being held.

Ableton has given us financial support to close the budget.

As in the past two years, all submitted papers have undergone a review process. Each paper was reviewed by at least two independent experts, whose feedback helped the authors improve their paper before the final version that you will find in the proceedings. This year, we also printed in the proceedings the abstracts of the tutorials, workshops and demos, as well as the program texts of the concerts, to create a complete documentation of the conference.

Thanks to everyone who has participated in bringing this Linux Audio Conference to life - authors, presenters, composers, sound artists, reviewers, helpers and anyone we may have forgotten - and we wish everyone a pleasant and enjoyable stay at the TU and in Berlin.

Marije Baalman, Folkmar Hein, Stefan Kersten and Simon Schampijer
Organisation Team LAC2007
Berlin, March 2007

LAC 2007 Organisation Team

Main orga:
Marije Baalman (TU)
Folkmar Hein (TU)
Stefan Kersten
Simon Schampijer (TU)

Assistance:
Miguel Álvarez-Fernández
Dimitar Anastasov (TU)
Philippus Baalman (Universiteit Twente)
Oswald Berthold (HU)
Martin Carlé (HU)
Jef Chippewa (CEC)
Christian Dietz (TU)
Fabian Gawlik (TU)
Florian Goltz (TU)
Doris Graße (TU)
Alexander Grüner
Eckehard Güther (TU)
Matthias Herder (TU)
Andreas Lehmann (TU)
Sebastian Roos (TU)
Sascha Spors (TU/T-Labs)
Torben Hohn (TU)
Wilm Thoben (TU)
Jan Weil (TU)
Stefan Weinzierl (TU)

Tesla/DAAD:
Ingrid Beirer (DAAD)
Carsten Seiffart (Tesla)

Streaming:
Jörn Nettingsmeier
Eric Dantan Rzewnicki

Campusradio:
Andreas Rotter
Robert Damrau

Partners

Ableton

Fachgebiet Audiokommunikation, Institut für Sprache und Kommunikation, Technische Universität Berlin

Berliner Künstlerprogramm des DAAD

C-Base

Seminar für Medienwissenschaft, Humboldt-Universität zu Berlin

Instituto Cervantes Berlin

MaerzMusik

Studio für elektroakustische Musik (SeaM), Musikhochschule Weimar

Tesla

Review Committee

Fons Adriaensen Laboratorio di Acustica ed Elettroacustica, Parma, Italy

Marije Baalman Audiokommunikation, TU Berlin, sound artist

Frank Barknecht GOTO10 (goto10.org)

Ivica Ico Bukvic D.M.A., Composition, Music Technology, CCTAD, and CHCI, Virginia Tech, Dept. of Music

Götz Dipper ZKM | Institute for Music and Acoustics, Karlsruhe

Dr. Albert Gräf Dept. of Music Informatics, Institute of Musicology, Johannes Gutenberg University Mainz, Germany

Steve Harris Garlik Limited, www.garlik.com

Takashi Iwai SuSe, ALSA development

Daniel James 64studio, www.64studio.com

Victor Lazzarini Music Technology Laboratory, Music Department, National University of Ireland, Maynooth

Fernando Lopez-Lezcano Sysadmin/Lecturer/Composer, CCRMA, Stanford University

Stefan Kersten Sound artist and freelance programmer

Jörn Nettingsmeier freelance audio/video engineer

Frank Neumann Harman/Becker Automotive Systems and "Passionate Linux Audio user/kind-of developer and evangelist"

Dave Phillips Linux audio documentation expert

Simon Schampijer Audiokommunikation, TU Berlin, OLPC

Sascha Spors Telekom T-Labs / TU Berlin

Jan Weil Nachrichtenübertragung, TU Berlin

Music Jury

Miguel Álvarez-Fernández Composer and Sound Artist

Marije Baalman Audiokommunikation, TU Berlin

Folkmar Hein Audiokommunikation, TU Berlin

Stefan Kersten Sound artist and freelance programmer

Martin Supper Universität der Künste, Berlin

Simon Schampijer Audiokommunikation, TU Berlin

Wilm Thoben Audiokommunikation, TU Berlin

Addresses

TU Main building
Straße des 17. Juni 135,
Berlin-Charlottenburg
U2 - Ernst Reuter Platz
S-Bahn - Tiergarten

C-Base
Rungestraße 20,
Berlin-Mitte
U8 - Jannowitzbrücke
S-Bahn - Jannowitzbrücke

Instituto Cervantes Berlin
Rosenstrasse 18-91,
Berlin-Mitte
U8 - Weinmeisterstraße
S-Bahn - Hackescher Markt

MaerzMusik
Haus der Berliner Festspiele,
Schaperstrasse 24,
Berlin-Wilmersdorf
U1 - Spichernstraße

Tesla
Klosterstraße 68-70,
Berlin-Mitte
U2 - Klosterstrasse
S-Bahn - Alexanderplatz, Jannowitzbrücke

Contents

Thursday - March 22 - Papers

Friday - March 23 - Papers

Saturday - March 24 - Papers

Sunday - March 25 - Papers

Further Papers

Keynotes and panel discussion

Tutorials

Demos

Workshops

Hands On Demos

Technical Tour

Wave Field Synthesis compositions

Installations

Club Concert at C-Base

Concert at Cervantes

Concert I at Tesla

Concert II at Tesla

Sonic Arts Lounge at MaerzMusik

Unplugged Concert - Lichthof

Proposal for an XML format representing Time, Positions and Parts of Audio Waveforms

Jens GULDEN
Formal Models, Logic and Programming (FLP),
Technical University Berlin
jgulden@cs.tu-berlin.de

Hanns Holger RUTZ
Studio for electroacoustic Music (SeaM)
HfM Weimar
hanns.rutz@hfm-weimar.de

Abstract

A domain-specific model describing the notions of *time*, *positions* and *parts* of audio waveforms and other media is proposed. The model is primarily intended to be part of an overall XML specification for musical instruments and other sound generators, but appears also applicable to other time-based media. A practical software-implementation would provide functionality to mark points and regions in audio waveforms, apply annotational metadata to these positions and parts, cut an audio waveform into pieces and recombine them in a different order, or extract parts of a waveform as individual content.

A prototype implementation that realizes a functional subset is available.

Keywords

Audio, Time, Slice, Position, Metadata

1 Introduction

We describe a proposal for modeling the notions of *time*, *positions* and *parts* of audio waveforms. The model is also applicable for other kinds of musical pieces and time-based media.

Section 2 motivates the need for a common interchangeable standard for locating positions in audio waveforms and marking parts of them. The following section 3 then sketches a conceptual model from scratch which provides the basis for implementing an XML data format. In section 4 such an initial XML-based implementation of the model gets introduced. Section 5 shows a prototypical implementation using the XML format, written in SuperCollider 3. The current status of sound-slices support as implemented by the real-world audio file editor Eisenkraut is covered by section 6. Finally, section 7 sketches future plans and section 8 gives a brief concluding summary.

The appendix contains an XML-Schema definition which formalizes the proposed XML language.

2 Need for a common file format for slicing audio waveforms

The value of multimedia data highly increases with the ability to structure it and refer to individual parts and positions of time-based media rather than treating every physical media file as one unstructured monolithic block. Every additional dimension of structuring broadens the variety of potential uses (e. g. attaching metadata to time-based media), and also unfolds a wide range of possibilities for artistic transformations.

However, until today even for audio data no standard of interoperability for marking positions and specifying regions is available. The field of possible applications for such a standard is huge: it ranges from creative musical activities in transforming audio, e. g. creating sound-collages or break-beat loops, audio-engineering applications in mixing / mastering, to possibly any private usage when attaching annotations to an audio file, e. g. let your friend read “this is the best guitar-solo I ever heard” right in the moment when the corresponding part is played from a music file.

When composing with sound, or creating realtime interactive pieces and installations, the proper sectioning of audio files and the tagging of time positions becomes vital. Therefore, new application fields for the classical sound-editor tool emerge, which is demonstrated later in this article.

3 Proposed Conceptual Model

3.1 Requirements

While terms like "time" and "position" seem clearly defined in a physical sense (locating coordinates in 4D time-space), in a musical context "time" and "position" have more facets and multiple dimensions of meaning. The notions of times and

positions in musical pieces are rarely used in an absolute sense. Instead, they usually depend on each other and are also relative to musical characterstics as e. g. the tempo of a piece, or a waveform's sample-rate. It is thus vital to an adequate domain-specific model of times and positions in musical pieces not to be restricted to a flat, absolute view on times and positions (as for example the umber of milliseconds or sample-frames from the start), but to treat times and positions as recursively interdependent. The model proposed here captures these notions.

Further conceptual requirements that arise from this domain-specific interpretation of the terms are:

- *Time* in a musical piece may simply refer to clock time (e. g. '5.2 s'). But especially in the context of music, time may also refer to a beat-measurement scale (e. g. 'three...and'-beat) or, when handling waveform data, may be given as number of sample-frames. Time can also be understood as being relative to a part (e. g. 'two-thirds of part A').
- *Positions* in musical pieces are understood as inherently relative to other positions (e. g. 'at beat 3 after the end of the second verse'), not just as absolute positions denoted by single time-value. Thus, a *position* is specified recursively by refering to another *position*, combined with a *time* offset that denotes the distance between the two positions.
- *Parts* of musical pieces can be related to each other in several different ways, but *parts* may also be completely unrelated to each other. The specification is thus required to allow flexible declarations of *parts*, e. g. giving parts in a consecutive sequence, placing them in a hierarchy (as *parts* of other *parts (, ...* of other *parts, ...* etc.)), or let them freely overlap.

3.2 Terminology

To embed the above requirements into a formalized model, conceptual terms and additional helper terms are introduced as model elements. These basic model-terms are as follows:

"Waveform"

The physical representation of a sound or musical piece. A waveform has a *sample-rate* and a *length in frames* associated with it, sometimes a tempo can be specified as a *beats per minute* measure (which may default to a fix value and remain unused on sounds that are not tempo-structured), and an *offset* can be specified which denotes the actual start of an audio-piece in the waveform. Optionally, a waveform can also have a *name* to refer to it. Every waveform owns an implicit reference to exactly one *master-slice*, from which any number of *slices* can be derived.

"Anchor"

A technical alias for "position in a waveform". This is sometimes called "marker" synonymously. *Anchors* are always specified relative to another *anchor*, often the *start-anchor* of a parent-slice in the waveform (or implicitly the beginning of the entire waveform, which expresses an absolute position). As each anchor refers to another anchor to which it is relative to, the concept of anchors is inherently a recursive one, and so is the concept of *slices:*

"Slice"

A technical alias for "part of a waveform". A *slice* is delimited by a start-anchor and an end-anchor. Every slice also references exactly one *parent-slice*. The positions specified by the start-anchor and end- anchor usually are relative to the parent-slice. In most cases, slices also have a name which identifies them uniquely.

A slice is recursively specified as always having a parent-slice associated with it. This way, a slice is always a child of another slice, which again is a child of another slice, etc.

Different slices may overlap.

The top-most slice which represents a whole waveform is called the *master-slice*.[1] The master-slice never gets created explicitly, instead there is an implicit 1:1-relationship between each waveform and one master-slice representing it. (I. e., every waveform 'automatically' has one master-slice associated with it.) See the specification of the `<wave>`-element below in section 4 XML implementation of the model.)

"Time"

Time in a sound or musical piece. As demanded by the requirements, *time* can be specified either in number of beats, in sample-frames, as time in seconds or milliseconds, or relative to the length of the current parent-slice. Such different possible descriptions of *time* are named *time-values*.

[1] The idea of a master-slice is introduced as a helper-concept to mark a starting point of the recursive slices-hierarchy. It is not vital to the overall model and might get replaced in future versions of the model, e. g. by allowing any slice to take the role of a top-most slice.

The time offset between a relative base-*anchor* and a specified *anchor* is also represented by an instance of *time*.

"Time-Value"

A symbolic string-value from which *time* can be derived. The string is composed of a numeric part and an optional suffix which determines its type (beats, sample-frames, seconds, etc.). The suffix can be

- "**~**", for beat measurements (`[measr:]beat.frac`)
- "**#**", for sample-frames.
- "**s**", for seconds
- "**ms**", for milliseconds
- or none for a relative value between 0.0 (start of the parent-slice) and 1.0 (end of the parent-slice).

Note that the same *time* can be expressed by different *time-values*.

"Cuelist"

One possible way to gain practical use from cutting musical pieces into slices is to build a new sequence of musical pieces from the slices. As an exampleThis way, e. g., a song could first be divided into slices of verses, choruses, bridge-parts etc., and later get combined in a different order. Such a re-combined sequence is called a *cuelist*.

A *cuelist* is usually composed of multiple *cues*.

"Cue"

A reference to a *slice*, to be used inside a *cuelist*. The advantage of combining *cues* to cuelists (and not directly combining slices to possible 'slicelists') is that by using *cues* as explicit references to *slices*, confusion about identity and multiplicity is avoided. Multiple occurrences of the same *slice* inside a *cuelist* become cleanly modeled as multiple distinct *cues* that reference one and the same *slice*. For convenience, each *cue* should be repeatable / loopable *n* times, so that multiple repetitions of *slice*-occurrences can be described by a single *cue*.

Visualization and Overview

A UML class-diagram ([3]) showing the model concepts and their relationships is displayed in Fig. 1. Table 1 summarizes the terms introduced with the model.

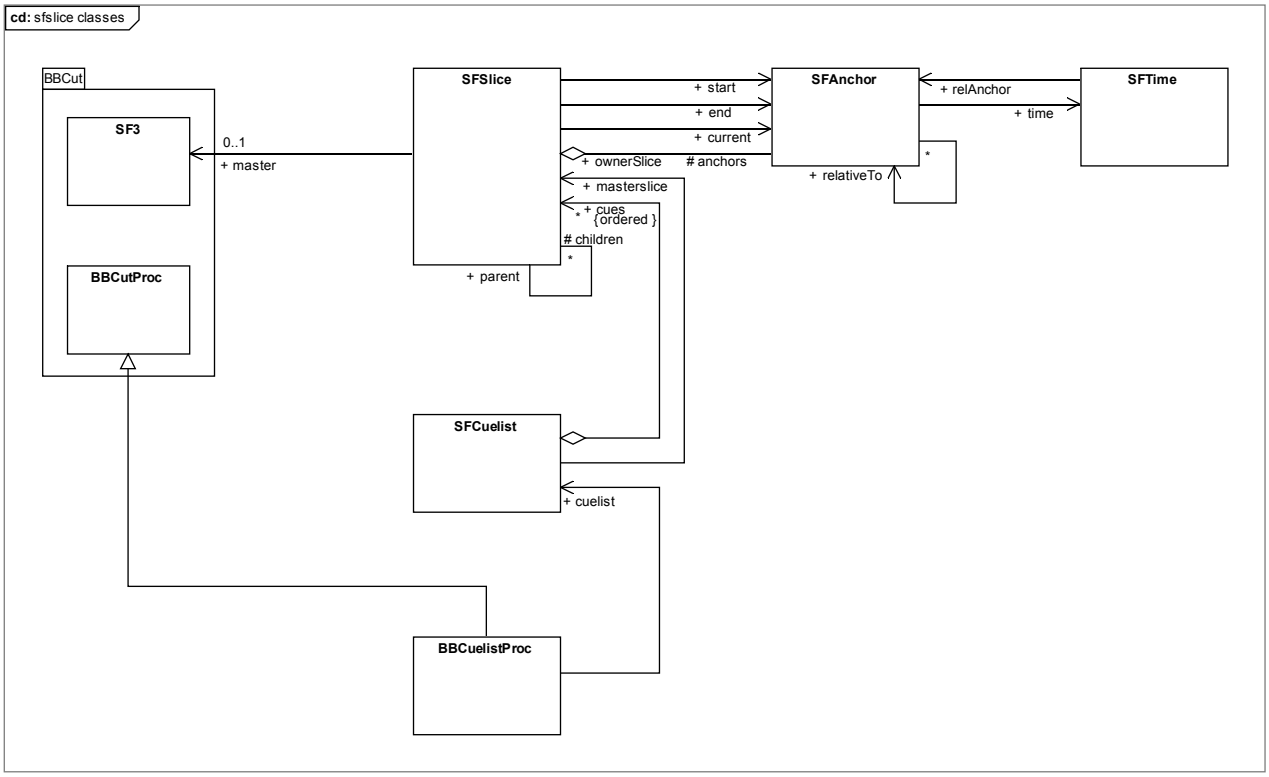


Fig. 1: UML class-diagram of the model

Term	Description
Waveform	The physical representation of a sound or musical piece. Every waveform owns an implicit reference to exactly one master-slice.
Slice	Part of a waveform. A slice is delimited by a start-anchor and an end-anchor.
Anchor	Position in a slice. Always specified relative to another anchor.
Time	Time in a sound or musical piece, either specified as number of beats, in sample-frames, as time in seconds, or relative to the length of the parent-slice.
Time-Value	Description of time. Different time-values can denote identical points in time (because there are multiple different ways to specify time).
Cue	A reference to a slice.
Cuelist	Combined sequence of cues.

Table 1: Summary of introduced terms

4 XML implementation of the model

The conceptual model developed in the previous section is now used as a basis for specifying a set of XML-elements which make up an XML language implementation for structuring audio data.

4.1 Overall Example

Example 1 gives an introductory overview of how a waveform is sliced into parts and re-joined as a cuelist in which the original parts are played in a different order and with repetitions.

```
<instrument>

    <wave src="examples/audio/NosNodOrr.wav">
        <!-- wavefile is implicitly used as 'master-slice' -->

        <slice name="start" to="20.55839s"/>

        <!-- empty slice, starts at end of previous slice: -->
        <pad to="27.69878s"/>

        <!-- next slice, starts at end of previous slice: -->
        <slice name="zwischen" to="34.83637s"/>

        <!-- relative time-values: -->
        <slice name="wummer" from="0.33" to="0.5"/>

        <!-- full syntax variant: -->
        <slice name="basis">

            <start>
                <anchor time="34.9s"/> <!--shortcut-anchor-->
            </start>

            <end>
                <anchor>  <!-- full anchor syntax -->

                    <time>
                        <seconds>49.1638</seconds>
                    </time>

                    <!--<time value="49.1638s"/> (shortcut)-->
                </anchor>
            </end>

        </slice>

        <!-- specifiying length instead of end-anchor: -->
        <slice name="funny" from="49.19s" length="7.14s"/>

        <!-- length only, starts at end of previous slice: -->
        <slice name="melody1" length="10.26s"/>

        <!-- time-value via frame-position: -->
        <slice name="melody2" to="42256#"/>

        <!-- relative time-value: -->
        <slice name="end" to="1.0"/>

    </wave>


    <!-- a cuelist for playing slices in a different order -->
    <cuelist name="song">
        <cue slice-ref="start"/>
        <cue slice-ref="zwischen"/>
        <cue slice-ref="wummer" repeat="4"/>
        <cue slice-ref="basis"/>
        <cue slice-ref="funny"/>
        <cue slice-ref="melody1" repeat="2"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="basis"/>
        <cue slice-ref="melody1" repeat="2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="wummer" repeat="8"/>
        <cue slice-ref="funny"/>
        <cue slice-ref="melody1"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="end"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody1"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="end"/>
    </cuelist>

</instrument>
```

Example 1: sliced waveform in XML representation

Let's look at this example step by step:

The `<instrument>`-element represents an outer container that initiates the document. In an overall XML specification for musical instruments, `<instrument>` could carry far more functionality than here.

The `<wave src="..">`-element declares a wavefile to be loaded.

The `<slice>`-elements inside the `<wave>`-element describe parts into which the waveform is divided. Note that the conceptual model requires all `<slice>`-elements to appear inside other slices (their parent-slice) or to explicitly reference a parent-slice. The `<wave>`-element implicitly provides a master-slice, so `<slice>`-elements can as well appear inside `<wave>`-elements to use the master-slice as their parent-slice.

4.2 Shortcut syntax vs. full syntax

One general aim of the language design is to both support manual editing of the XML document and allow easy machine writability / readability. This is why several declarations are supposed to be expressable either by the use of a shortcut syntax variant for manual editing, or by a full-size syntax in which every model concept is represented by its own XML-element. Generally, the shortcut-syntax allows to specifiy some model-concepts as attributes, while the full-size syntax makes use of one separate XML-element per model-concept.[2]

An implementing application may decide to only support either of the syntax variats, e. g. only the full-sized syntax if the XML format is intended to be used as a file-format for data-storage only (serialization / deserialization of objects from an object-oriented software-environment is easy to achieve in full-size syntax).

For means of readability, most of the upcoming examples make use of the shortcut-syntax.

4.3 Specifying slices

Shortcut syntax

These are examples of how the time-values of start-anchors and end-anchors of a slice could be specified via XML-attributes:

```
<slice name="test" from="0.1s" to="3.5s"/>
<slice name="test2" from="2.5~" to="1.0"/>
```

Example 2: specifying time-values via attributes

Using this shortcut form, anchors are declared by specifying time-values via the attributes `from`, `to` and `length` (e. g. `<slice from="1.5s" to="3.75s"/>`). Time-values of different types can be distinguished via a suffix in the string, which could be "**~**" for beat measurements, "**#**" for sample-frames, "**s**" for seconds, "**ms**" for milliseconds, or none for a relative value between 0.0 (start of the parent-slice) and 1.0 (end of the parent-slice). See also section 3.2 “Time-Value”.

Different ways of using the `from` and `to` attributes in `<slice>`-elements allow to distinguish between different semantics of the shortcut notation: If both a `from` and a `to` attribute are given, the slice should be placed between two anchors which are implicitly created at the specified times.[3] If only a `from` attribute is given, a start-anchor should implicitly get created at the time given by the `from` attribute, and the end of the slice should be denoted by the end-anchor of its parent-slice. Finally, if only a `to` attribute is given, the slice should start immediately behind the previously declared slice (in terms of XML, its previous sibling), or at the beginning of the parent-slice, if the current `<slice>`-element is the first one in its hierarchy level.[4]

```
<wave name="mywave" src="/data/wav/loop3.wav">
    <slice name="intro" to="12.305s"/>
        <!-- first slice starts at time-value 0.0 -->
    <slice name="vocals1" to="37.975s"/>
        <!-- identical: from="12.305s" to="37.975s" -->
    <slice name="bridge1" to="49.534s"/>
</wave>
```

Example 3: context-dependent defaults for `from` and `to` attributes

[2]The distiction between a shortcut-syntax and a full-sized syntax is uncommon to XML-language design, as XML usually is used as an internal data storage format only and human readability is not a design-goal. We believe, however, that in the domain of artistic work on musical material, it is desirable to provide a manually editable view on the model which can undergo any artistic transformation by hand.

[3]It should be a convention to understand start-times as being included in the slice, describing the first discrete point in time that lies within the slice. On the contrary, end-times should be understood as being excluded from the slice, denoting the first discrete point in time after the end of the slice. (Discrete points in time could e. g. be counted by the sample-frame index.)

[4]This introduces context dependency in the parsing-process which is more complex than stand-alone XML-parsers are designed to handle by themselves. However, keeping track of such a context is relatively easy if parsing is done by traversing the document-tree on an application-level ('manually' programmed), or when serializing / deserializing objects to or from XML which usually allows the execution of class-specific code while parsing.

A slice's overall position could thus result from specifying either of the following options:

1. Independently specifying references to the start and the end anchor:

   ```
   <slice from="time-value" to="time-value"
   name="..."/>
   ```

2. Setting a start anchor and letting the end-anchor follow after a given time duration:

   ```
   <slice from="time-value" length="time-value"
   name="..."/>
   ```

3. Setting an end-anchor and letting the start-anchor appear earlier by a specified time duration:

   ```
   <slice to="time-value" length="time-value"
   name="..."/>
   ```

4. Setting an end-anchor and using the start-anchor of the most previously declared slice (or the parent's start-anchor if the decared slice is the first one on its hierarchy level):

   ```
   <slice to="time-value" name="..."/>
   ```

5. Using the end-time of the most previously declared slice as a start-anchor's time and creating an end-anchor of the specified duration after the start-anchor:

   ```
   <slice length="time-value" name="..."/>
   ```

6. Or using the long form for explicit anchor creation inside `<slice>`-elements, see the following sub-section.

A parser should be aware of these different combinations and report an error if invalid attribute sets (e. g. all three `from`, `to` and `length`-attributes set at the same time) are encountered.

Full-size syntax

Slices, anchors and times can also be notated explicitly in a full-size XML notation variant. Using this syntax, the conceptual relations between these entities, as they have been expressed by the conceptual domain model, can get reflected directly by the corresponding XML elements `<slice>`, `<anchor>` and `<time>`.

Just as slices can be notated either in a detailed long-form or a shortcut-form, anchors may also be declared either by the use of a full-size syntax, or alternatively by making use of a shortcut `time`-attribute inside the `<anchor>`-element.

```
<!-- full-size slice and anchor declarations: -->

<slice name="myslice">
    <start>
        <anchor name="my_unique_start_anchor">
            <!-- a name is given for also referring to this
                 anchor from elsewhere -->
            <time>
                <beats>2.5</beats>
            </time>
        </anchor>
    </start>
    <end>
       <anchor ref="my_unique_end_anchor-declared_elsewhere"/>
    </end>
</slice>


<!-- full-size anchor declaration outside a slice: -->

<anchor name="my_anchor">
    <time>
        <beats>2.5</beats>
    </time>
</anchor>

<!-- shortcut anchor declaration: -->

<anchor name="my_anchor2" time="2.5~"/>
```

Example 4: full-size and shortcut anchor declarations

Referencing a parent-slice

In order to provide an easy way of referencing a parent-slice, it should be possible to implicitly specify slices as being children of another slice by placing their declarations inside a sub-tree of the parent's `<slice>`-element.

An alternative way of referencing a parent-slice without nesting a `<slice>`-element inside another `<slice>`-element (or a `<wave>`-element), should be provided by explicitly referencing the parent-slice via the `parent-ref` attribute.

```
<!-- implicitly reference a parent-slice through nesting: -->

<slice name="vocals1" from="9.35s" to="37.975s">
    <slice name="first-half-of-vocals1" from="0.0" to="0.5"/>
</slice>


<!-- explicitly reference a parent-slice by name: -->

<slice name="vocals1" from="9.35s" to="37.975s">
    <!-- ... -->
</slice>

<!-- not an XML-child of vocals1, but uses it as parent: -->

<slice parent-ref="vocals1" name="first-half-of-vocals1"
from="0.0" to="0.5"/>
```

Example 5: referencing parent-slices

Referencing anchors

Just like slices can be referenced via identifier-names, it should also be possible to reference anchors by name. This is especially useful when multiple slices are to share the same anchors.

Additionally, the declaration of named anchors outside any slice should be possible, in order to make anchors referrable from other elements.

```
<wave name="mywave" src="/data/wav/loop3.wav"/>

<slice name="vocals1" parent-ref="mywave">
    <start>
        <anchor name="my_unique_start_anchor">
            <!-- name for referencing from elsewhere-->
            <time>
                <beats>2.5</beats>
            </time>
        </anchor>
    </start>
    <end>
        <anchor ref="my_unique_end_anchor"/>
        <!-- anchor declared elsewhre -->
    </end>
</slice>

<!-- shortcut notation (the application is responsible
     for distinguishing time-values from anchor-refs): -->

<slice parent-ref="mywave" name="first-part-with-vocals1"
       from="0.0" to="my_unique_end_anchor"/>


<!-- a named anchor declared outside a slice: -->

<anchor name="global_start">
    <time>
        <seconds>2.35</seconds>
    </time>
</anchor>

<!-- multiple references to the same named anchor: -->

<slice name="all" from="global_start" to="273.8s"/>

<slice name="vocals1" from="global_start" to="52.4s"/>
```

Example 6: named anchors

An implementation should evaluate references to declared elements after having completely parsed the XML document, in order to allow forward-references to identifiers.[5]

4.4 Padding unused parts

In order to mark areas of the waveform as unused (among the group of children in one parent-slice), a `<pad>`-element should be provided.[6] A `<pad>`-element only makes sense in a sequence of `<slice>`-elements which are specified as consecutively following each other (i. e. tags of the form `<slice to="..." ... >` or `<slice length="..." ... >`). The `<pad>`-element could be used just like the `<slice>`-element, in order to declare an unused part of the sound or musical piece:

1. either by padding to a specific anchor (any slice or other pad created afterwards will start at that end time):

   ```
   <pad to="time-value"/>
   ```

2. or by padding a specific duration of time:

   ```
   <pad length="time-value"/>
   ```

4.5 Specifying cuelists

A *cuelist* is a sequential collection of references to slices. Such an individual reference should be called a *cue*. Cues should allow slice-references to appear in any order and for any number of times in a cuelist. When the cuelist is played-back by an implementing application, it should sequentially output the slices in the same way as if they had been joined to be one major waveform.

It should be possible to attach a name to a cuelist via a `name`-attribute, in order to have a reference handle to the cuelist from a hosting application.

Recursive use of cuelists

Recursive declarations of cuelists inside cuelists should be possible. This is especially useful when the inner cuelist is to be repeated multiple times by use of the `repeat`-attribute.

A `ref`-attribute should allow to refer to cuelists declared elsewhere in the document.

```
<!-- a cuelist containing multiple cues: -->

<cuelist name="myCuelist">
    <cue slice-ref="intro" repeat="2"/>
    <cue slice-ref="vocals1"/>
    <cue slice-ref="bridge1"/>
    <cue slice-ref="intro"/>
    <cue slice-ref="vocals2"/>
    <cue slice-ref="chorus1" repeat="3"/>
</cuelist>

<!-- cuelists inside another cuelist: -->

<cuelist name="song">
    <cue slice-ref="intro" repeat="2"/>
    <cue slice-ref="vocals1"/>

    <cuelist repeat="2">
        <cue slice-ref="bridge1"/>
        <cue slice-ref="intro"/>
    </cuelist>

    <cuelist ref="more"/>
</cuelist>

<cuelist name="more">
    <!-- ... -->
</cuelist>
```

Example 7: some cuelists

[5]This suggestion again imposes an additional requirement on context-sensitivity of the parser.

[6]From an implementer's perspective the `<pad>`-element might seem redundant, as it can equivalently be replaced by a dummy-slice which never gets used. However, we believe that from a domain-specific perspective additional semantics is expressed by explicitly marking a part as unused, which is why the `<pad>`-element is icluded in the proposal.

5 Prototype implementation for testing

There is an experimental prototype software application that demonstrates some features of the proposed XML format. It is called the 'SFSlice classes' and written in the SuperCollider 3 [4] language, running on top of the BBCut [5] class library.

The application comes with an example audio file which gets sliced into several cues and is then re-combined to a new and longer musical piece as described by the supplied cuelist. Example 8 shows the supplied XML file which gets processed by the prototype to play a new song out of the sliced audio file.

The prototype exclusively uses the shortcut syntax.

```
<instrument>

    <wave src="examples/audio/NosNodOrr.wav">
        <slice name="start" to="20.55839s"/>
        <slice name="leer" to="27.69878s"/>
        <slice name="zwischen" to="34.83637s"/>
        <slice name="wummer" from="33.06s" to="34.82s"/>
        <slice name="basis" from="34.9s" to="49.16338s"/>
        <slice name="funny" from="49.19s" to="56.33s"/>
        <slice name="melody1" from="56.35s" to="63.46s"/>
        <slice name="melody2" from="63.5s" to="70.65s"/>
        <slice name="melody3" from="70.7s" to="77.8371s"/>
        <slice name="end" to="1.0"/>
    </wave>

    <cuelist name="song">
        <cue slice-ref="start"/>
        <cue slice-ref="zwischen"/>
        <cue slice-ref="wummer" repeat="4"/>
        <cue slice-ref="basis"/>
        <cue slice-ref="funny"/>
        <cue slice-ref="melody1" repeat="2"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="basis"/>
        <cue slice-ref="melody1" repeat="2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="wummer" repeat="8"/>
        <cue slice-ref="funny"/>
        <cue slice-ref="melody1"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="end"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody1"/>
        <cue slice-ref="melody2"/>
        <cue slice-ref="melody3"/>
        <cue slice-ref="end"/>
    </cuelist>

</instrument>
```

Example 8: XML example file running with the prototype implementation

The prototype implementation, along with example audio files and additional documentation of the conceptual model and the XML implementation, is available for download at [1].

6 Real-world application Eisenkraut

Finally we shall look at a concrete application which handles waveforms and time positions. Eisenkraut is a cross-platform audio file editor that utilizes the SuperCollider 3 server for realtime audio processing [2].

In the current beta-version 0.7, only anchors (called “Markers” here) are supported, but the implementation of slices (or “Regions”) is planned. Markers are displayed in the traditional way as a separate track along with the waveform, where a linear timeline is forming the horizontal axis. In the underlying model, each track is associated with a list of regions which is double-sorted (by start frame and stop frame). This way editing operations such as cutting, pasting, moving etc. can be applied uniformely to any data that can be described by a start and stop frame.

While the underlying framework uses a sample frame timebase, on the user presentation level time-values can be specified in sample frames, milliseconds, hh:mm:ss or percentage (refererring to the “implicit master slice” of the whole document span), as shown in Fig. 2:

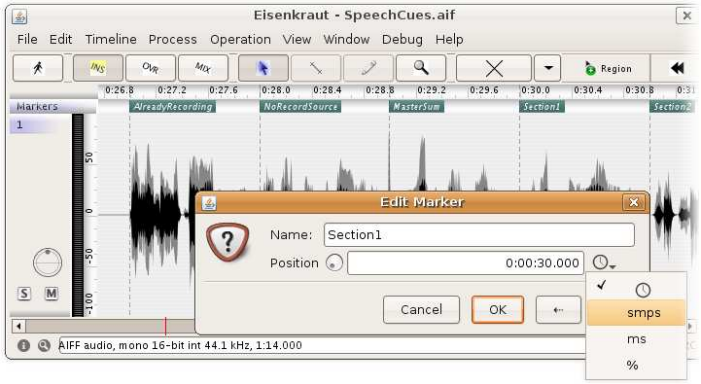


Fig. 2: editing a marker in Eisenkraut

Unfortunately, the notion of markers and regions varies depending on the audio file format used. For example, AIFF files [6] provide built-in support for markers but not regions. WAVE files [7] provide various concurrent concepts of both markers and regions.

While Eisenkraut currently uses the marker / region facilities of the particular audio file formats, the implementation of an independent time positions file as proposed in this paper is highly desirable for serveral reasons:

- The editing of marker positions is often a separate process independent of cutting and transforming the audio waveform. In existing audio file formats such as AIFF or WAVE, the deletion or creation of just a single marker

requires the whole audio file – often hundreds of megabytes – to be rewritten when saving

- Markers in audio files may not be meaningful to some stages of the processing. for example, in the SuperCollider 3 environment, streaming a soundfile off disk that contains hundreds or thousands of markers puts an unnecessary computation burden on the server objects (*scsynth*) reading the file, although the markers are only useful for client side objects (*sclang*).
- Markers can take different roles and often need to be associated with enhanced meta-data. For example, an audio file may be segmented in order to allow dynamic selections of material in a realtime performance or sound installation, requiring additional data such as fade-in and fade-out times, relative gain adjustments, descriptive attributes such as character-categories etc. Common audio file formats do not provide sufficient means to add these meta-data to markers.

The solution to this last issue is to copy the markers into custom (text or binary) files, separate from the audio files. Eisenkraut already facilitates this approach through the supply of an Open Sound Control (OSC) [9] server. Clients, typically written in SuperCollider, can interact with documents of the audio file editor in two ways:

- explicit OSC addresses and commands are provided to access the documents and its tracks. The following example shows how all markers of the current active audio file document can be retrieved from a SuperCollider client:

```
e = Eisenkraut.default;
e.addr.connect;

// queries all markers of the active document
(
fork {
    var msg, rate, num;
    rate  = e.query( '/doc/active/timeline', \rate ).first;
    num   = e.query( '/doc/active/markers', \count ).first;
    msg   = e.get( '/doc/active/markers', [ \range, 0, num ]);
    msg.pairsDo({ arg pos, name;
    ("Marker '"++name++"' at frame "++pos++" = "
        ++(pos/rate).asTimeString( 0.001 )).postln;
    });
}
)
```

Example 9: reading markers through OSC

- inversely, the client can create new markers:

```
// create a geometric series of 20 markers across the file
(
fork {
    var len, scale, pos, names, marks;
    len   = e.query( '/doc/active/timeline', \length ).first;
    pos   = Array.geom( 20, 0.1, 1.2 );
    scale = len / (pos.last * 1.2);
    pos   = pos.collect({ arg t; (t * scale).asInteger });
    names = Array.fill( 20, { arg i; "Mark #" ++ (i+1) });
    marks = (pos ++ names).unlace( 20 ).flatten;
    e.listSendMsg([ '/doc/active/markers', \add ] ++ marks );
}
)
```

Example 10: adding markers through OSC

- implicit OSC addresses and commands are provided through SwingOSC [9]. This framework allows the clients to create and control custom GUI elements inside the Eisenkraut application. A large set of specialized GUI classes for SuperCollider is provided. Consequently, the editor itself can be kept small and generic, while custom marker handling and manipulation dialogs can be programmatically added by the client, as depicted in Fig. 3:

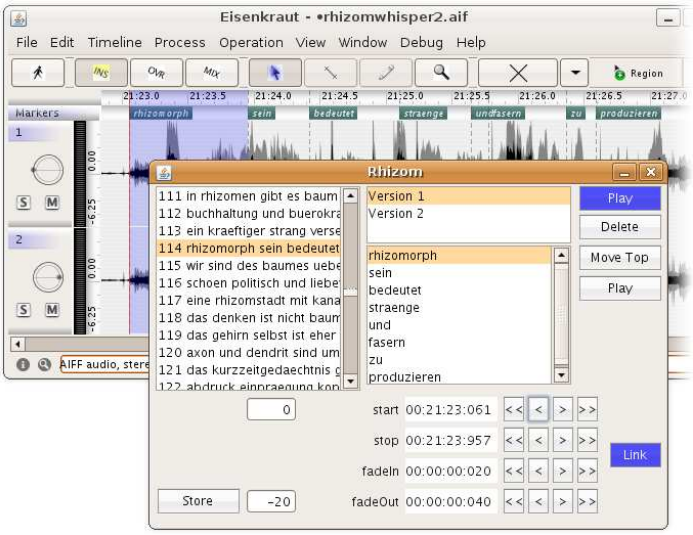


Fig. 3: custom editor using SwingOSC

- although Eisenkraut does not yet provide any container hierarchies for markers, the client can well present them in containers as indicated by the three list views. Also "meta-data" like fade-in and fade-out times as well as relative gains per region are presented in the client view. The client takes responsibility for storing the markers, in fact using XML files in this case.

Some additional work needs to be carried out to smoothen the interaction between OSC client and Eisenkraut. For example, markers could be tagged with flags, indicating their colour or indicating whether they should be saved along with the default

audio file or separately by the client. This way, the greatest flexibility can be maintained without making presumptions about the nature of the markers in the host application.

With this architecture, a new role is given to the traditional tool of the "soundfile editor". Instead of taking the road of merging soundfile editors with multitrack recording applications – as for example exercised in Audacity [10] –, the audio file editor becomes a core *modular* element inside a *networked* environment for the composition of electroacoustic and noise music, realtime performance and sound installation. It is in all forms of musique concrète that the structuring process departs from the analysis and editing of sonic materials, chunking of time being the most fundamental element in this structuring.

A final example is presented in Fig. 4. The image shows a symbolic score for a sound installation in the former prison of the GDR's Minstry of State Security (MfS) in Erfurt, Germany, in summer 2006 [11]. A kind of timeline goes from top to bottom, traversing five different stages or four sound materials. Each sound material is shown as a circular shape, and markers are used for segmentation of each material (indicated by spokes). Again this is a simplification as fading characteristics are not displayed. As offsets and lengths of the sections are chosen randomly, one particular performance is indicated by blue sectors.

All markers have been transferred from Eisenkraut to SuperCollider using drag-and-drop (as the OSC server did not yet exist at that time). Also note that the region structure is partly linked together, as indiated by the connections between the materials "Wald Stein" and "Wald Atmo". In this case, a binary file format was chosen for both the flat and the hierarchical markers, however the presence of a standardized XML format and supportive libraries could have simplified the process.

7 Future Plans

The given proposal is an initial approach which has already started a discussion on improvements and alternatives. It will develop over time, and as the data format might get implemented in other applications, modifications will flow back from there.

Additional potential for further development lies in broadening the focus from audio-data to any time-based media, such as video, or even non-time based media like written documents. In these cases, slices could denote scenes, cuts or chapters and paragraphs. Time-values could be extended to be also specifyable as video-frames, i-frames, page-number / line-number combinations etc.

8 Conclusion

The proposed model covers several issues that have been identified as specific to the domain of marking positions and regions in audio waveform data. It has been shown that these tasks are of practical importance, as confirmed by experiences with the development of the Eisenkraut audio file editor.

The model and its XML implementation propose a number of profoundly elaborated constructs to fulfil musically relevant requirements. The approach thus can be used as an initial basis for further elaborating the model and the XML format. It is one contribution to a much wider range of interoperable data formats describing musical events and other time-based media which yet wait to be invented.

References

[1] Gulden, J., *Proposal for an XML Specification modeling Time, Positions and Parts of Audio Waveforms*, conceptual model and software, http://swiki.hfbk-hamburg.de:8888/MusicTechnology/786

[2] Rutz, H. H., *Eisenkraut – cross-platform audio file editor*, software, http://www.sciss.de/eisenkraut

[3] Gulden, J., *Model-Driven Software-Development with SuperCollider and UML*, this conference

[4] McCartney, J., *SuperCollider – A real time audio synthesis programming language*, software, http://supercollider.sf.net

[5] Collins, N., *BBCut SuperCollider 3 extension library*, software, http://www.cus.cam.ac.uk/~nc272/bbcut2.html

[6] *Audio Interchange File Format* http://www.borg.com/~jglatt/tech/aiff.htm

[7] *WAVE File Format* http://www.borg.com/~jglatt/tech/wave.htm

[8] Open Sound Control, http://www.opensoundcontrol.org

[9] Rutz, H. H., *SwingOSC – Open Sound Control server to script the java language*, software, http://www.sciss.de/swingOSC

[10] Dominic Mazzoni et al., *Audacity – The Free, Cross-Platform Sound Editor,* software, http://audacity.sourceforge.net

[11] Rutz, H. H., *untitled (Zelle 148),* installation, http://www.einschluss.de

[12] Sun Microsystems, *Sun Multi-Schema XML Validator,* software, https://msv.dev.java.net

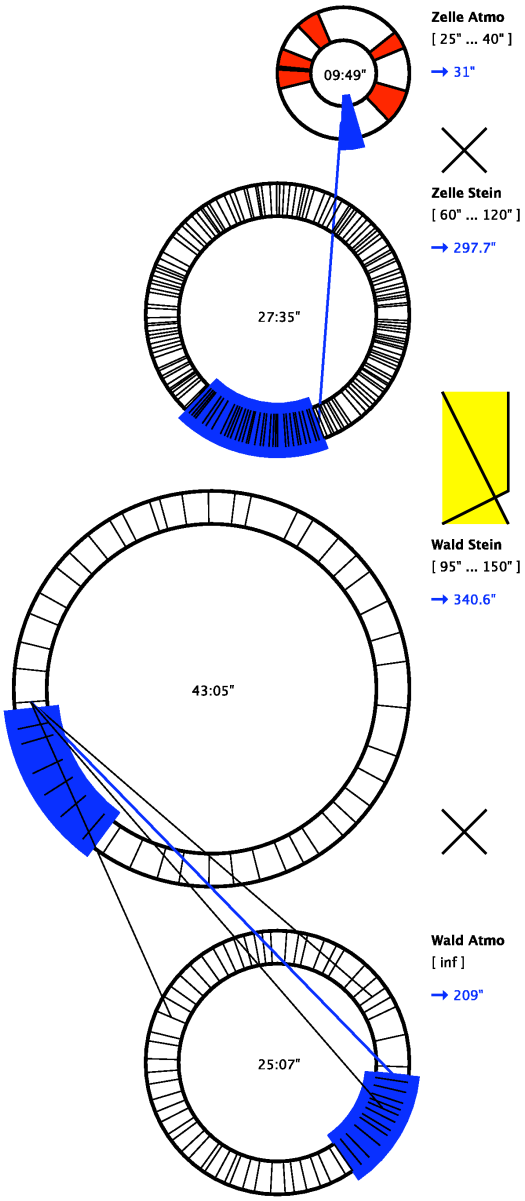


Fig. 4: Score of 'untitled (Zelle 148)', indicating audio file segmentations

Appendix: XML-Schema Definition (XSD)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           name="instrument">

    <xs:element name="instrument">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="wave"/>
                <xs:element ref="cuelist"/>
                <xs:element ref="slice"/>
                <xs:element ref="anchor"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="wave">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="slice"/>
                <xs:element ref="pad"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="src" type="xs:string"
                                     use="required"/>
            <xs:attribute name="tempo" type="xs:decimal"/>
            <xs:attribute name="offset" type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="slice">
        <xs:complexType name="slice-type">
            <xs:all minOccurs="0" maxOccurs="1">
                <xs:element ref="start"/>
                <xs:element ref="end"/>
            </xs:all>
            <xs:all minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="slice"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="parent-ref" type="xs:string"/>
            <xs:attribute name="from" type="xs:string"/>
            <xs:attribute name="to" type="xs:string"/>
            <xs:attribute name="length" type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="pad">
        <xs:complexType>
            <xs:complexContent>
                <xs:restriction base="slice-type">
                    <xs:all minOccurs="0" maxOccurs="1">
                        <xs:element ref="start"/>
                        <xs:element ref="end"/>
                    </xs:all>
                    <xs:attribute name="name"
                                  type="xs:string"
                                  use="prohibited"/>
                    <xs:attribute name="from"
                                  type="xs:string"/>
                    <xs:attribute name="to"
                                  type="xs:string"/>
                    <xs:attribute name="length"
                                  type="xs:string"/>
                </xs:restriction>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>

    <xs:element name="anchor">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element ref="time"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="ref" type="xs:string"/>
            <xs:attribute name="time" type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="start">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="1">
                <xs:element ref="anchor"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="end">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="1">
                <xs:element ref="anchor"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="time">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="beats" type="xs:decimal"/>
                <xs:element name="frames" type="xs:integer"/>
                <xs:element name="seconds" type="xs:decimal"/>
                <xs:element name="milliseconds"
                            type="xs:decimal"/>
                <xs:element name="relative" type="xs:string"/>
            </xs:choice>
            <xs:attribute name="value"
                          type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="cuelist">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="cue"/>
                <xs:element ref="cuelist"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="ref" type="xs:string"/>
            <xs:attribute name="repeat" type="xs:integer"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="cue">
        <xs:complexType>
            <xs:attribute name="slice-ref" type="xs:string"
                                           use="required"/>
            <xs:attribute name="repeat" type="xs:integer"/>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

All XML-examples presented in this article have been validated against this schema using the Sun Multi-Schema XML Validator ([12]).

Real-Time Multiple-Description Coding of Speech Signals

Jan Weil and **Kai Clüver** and **Thomas Sikora**

Communication Systems Group, Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
{weil,cluever,sikora}@nue.tu-berlin.de

Abstract

When sending speech data over lossy networks like the internet, multiple-description (MD) coding is a means to improve the perceived quality by dividing the data into multiple descriptions which are then sent as separate packets. In doing so the speech signal can still be decoded even if only parts of these descriptions are received. The present paper describes the structure of a software which demonstrates the benefits of this coding scheme using a client-server architecture.

Keywords

Multiple-Description Coding, Speech Coding, Real-Time Coding

1 Introduction

When transmitting real-time speech data over the internet to multiple receivers (multicast or broadcast), the quality on the receiving side depends on how many packets are lost on their way. Various receivers may experience greatly differing speech quality due to varying network conditions. Since in the multicast scenario delivery monitoring for all of the subscribed receivers is not feasible, action is needed on the receiving side.

To ensure graceful degradation of the perceived quality with increasing packet loss, MD coding can be applied, i. e. the data which is to be transmitted is divided into two or more descriptions. Even if not all of them are received, the signal can still be decoded, albeit with lower quality.

In the course of our project, two different MD speech codecs were developed. The first one is based on logarithmic pulse code modulation (PCM). The second one is a variation of the G.729 annex A codec, which is based on code excited linear prediction (CELP) and defined by the International Telecommunication Union (ITU). To show the improvement due to MD coding over lossy channels, a demonstrator application has been developed. The structure of this application is described in the present paper.

The rest of this paper is organized as follows: In Section 2 the principles of MD coding are explained. The actual implementation of the demonstration application, which is the main subject of this paper, is described in section 3. After that the usage of the program is illustrated in Section 4. Section 5 contains some concluding remarks.

2 Multiple-Description Coding

MD coding [1] provides a transmission link with diversity in order to improve robustness to channel breakdown. The coded signal is split into two or more descriptions which are transmitted over the same number of different channels. These channels may indeed consist of different physical links, or of different packets transmitted through networks like the internet.

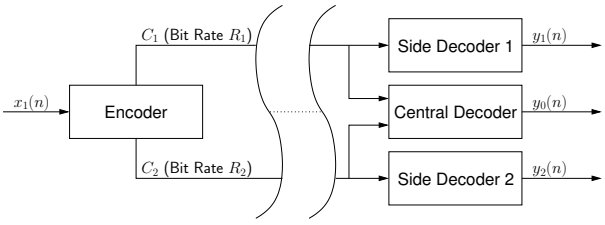


Figure 1: MD coding scheme with two descriptions

The principle of a two-channel MD coded transmission is shown in fig. 1. From the input signal, $x(n)$, the encoder generates two descriptions C_1 and C_2 to be sent over two lossy channels. If no loss occurs, both descriptions will be used by the central decoder to reconstruct the signal $y_0(n)$ with high quality. If one of the descriptions is lost, the received part of the code will enable its corresponding side decoder to yield a reduced-quality version of the output signal, $y_1(n)$ or $y_2(n)$. The transmission will be interrupted only when both descriptions

are lost.

The design of MD coders is subject to conflicting requirements [1]. If the side decoders were optimized for high signal quality, given the bit rates R_1 and R_2 for C_1 and C_2, little would be gained by combining both descriptions in the central decoder, which would then yield a similarly high quality, but at a considerably increased bit rate of $R = R_1 + R_2$. If, on the other hand, the central decoder were designed for minimum distortion at a bit rate of R, any splitting of the code would result in poor performance of the side decoders. Therefore, the usual objective is to find a compromise for central and side decoder qualities.

Many designs aim at balanced descriptions, i. e. equal bit rates ($R_2 = R_1$) and equal distortions of the side decoders. For more than two MD channels, balanced descriptions will yield decoding distortions which do not depend on the individual subset of descriptions but only on the number of descriptions received. The decoded quality will then degrade gracefully with increasing channel failure ratio.

A receiver for L descriptions consists of $2^L - 1$ decoders (including the central decoder). Consequently, the MD decoder will be extremely complex for high values of L if explicit side decoders are employed. This problem can be avoided by using a hierarchical (layered) speech coder together with forward error correction (FEC) codes for the construction of multiple descriptions [2]. The approach consists of applying unequal loss protection to L code layers and re-grouping the symbols of the resulting code words into L descriptions. With $k < L$ descriptions received, the MD decoder is able to decode the basic k layers of the coded speech. The side decoders are constructed implicitly by FEC decoding.

The FEC coding scheme causes high gross bit rates compared to the original codecs. It is possible, though, to trade robustness for bit rate savings if the coarse base layer is made up of more than one description.

3 Real-Time Implementation

3.1 Overview

An overview of the system as a whole is given in fig. 2. It allows multiple client applications to be served concurrently. On the sending side a server waits for incoming requests. Clients connect to the server and request speech data streams. For every requested stream, a new sender process is started by the server which connects to one of the available speech sources, encodes the data frame by frame, and sends the coded frames to the client by which the stream has been requested. It is possible to set up several streams to enable the user to compare different configurations.

3.2 Serving side

In fig. 2 every square-cornered box represents a separate process. Technically there is no need to start a separate process for every newly requested stream. Since, however, in the course of this project a single sender application had already been developed, it was easier to add a simple server and start the senders as subprocesses. This server is written in Python [3], which is a dynamic object-oriented programming language and even provides a `TcpServer` class as part of the standard library.

For demonstration purposes the source of the speech data was supposed to be selectable so that different speech sources could be offered. Using the Jack Audio Connection Kit (JACK) [4] this can easily be achieved. Part of it is `jackd`, a low-latency audio server which allows several different applications to share audio data among them. It is typically used for professional audio processing applications, which means that running `jackd` at a sampling rate of 8 kHz, as we did, is quite unusual. Nevertheless, JACK has proven absolutely appropriate to our needs.

In our case the selectable sources are provided by Ecasound [5]. Ecasound is a software package designed for multitrack audio processing, which, among other things, can be used to play back audio files in loops. For each loop a JACK port is registered so that our senders can connect to these ports.

3.3 Transport protocol

The transmission of audio and video data over the internet is often done using the Real-time Transport Protocol (RTP) which is usually built on top of the User Datagram Protocol (UDP). Compared to pure UDP, RTP additionally provides sequence numbering, time stamping, payload-type identification, and delivery monitoring. Because sequence numbering is the only feature that was needed in our case, we decided not to use RTP. Instead we added a 16-bit header containing a sequence number which is incremented for each packet. To make sure that an overflow of the sequence number does not

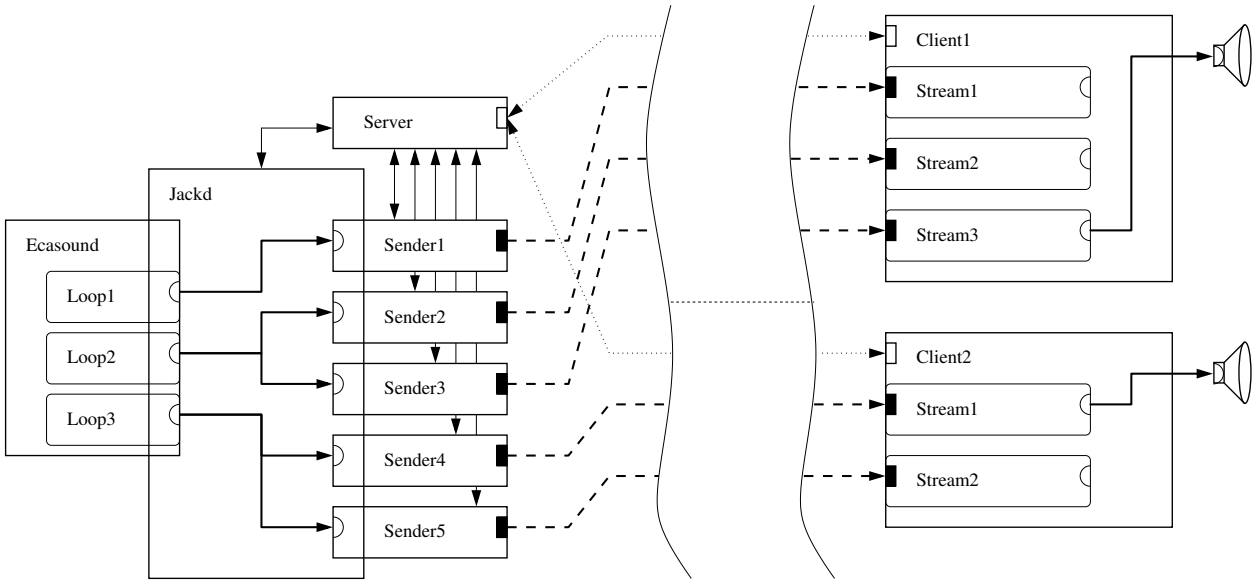


Figure 2: Structure of the system. Clients connect to the server's control port and request streams. For each stream, the server starts a new sender process which encodes the audio data delivered by JACK and sends it to the requesting client.

disturb the order of descriptions, the sequence number range is limited to a multiple of L.

The UDP port number on the receiving side is limited to the range of 55550 to 55569, which means that, on a distinct host, at most 20 separate ports can be served simultaneously.

3.4 Control protocol

In order to control the transmission of data a simple plain-text protocol has been designed. The transport protocol used for this purpose is the Transmission Control Protocol (TCP). The server waits for requests coming in on TCP port 55555. A request consists of a special command word in capital letters, possibly a list of arguments separated by spaces, and the closing two special characters carriage return (CR) and line feed (LF). After the request has been processed the server returns an answer which consists of a return string in capital letters (`OK` on success, otherwise `ERROR`), possibly an additional return string, and once more the concluding sequence CR-LF. There are four known commands:

`SOURCES` This command is used to retrieve the list of available speech sources. It does not allow any arguments. On success a comma-separated list of readable JACK ports is returned.

`PORT` Before a stream is requested, the client asks the server to allocate a port number. The next time the client issues the `OPEN` command this port will be used. On success the port number is returned by the server. If the client is located behind a firewall, incoming UDP traffic is usually blocked. If, however, the client sends an initial packet to the newly allocated port on the serving side, many firewalls will accept the incoming stream as a response to this packet. This concept is known as UDP hole punching.

`OPEN` Exactly six arguments are expected to open a new stream: the number of the receiving UDP port, one of the formerly listed sources, the codec to use (either `pcm` or `celp`), the number of descriptions to use, the number of descriptions which make up the base layer, and an additional argument which is either the segment length (PCM) or the number of CELP segments per MD frame. If a new sender process has been started successfully the server returns the corresponding process ID.

`CLOSE` This command needs the formerly transmitted process ID of a sender as the only argument. By receiving this command the server sends a signal (`SIGINT`) to the appropriate process. To ensure that no other processes are killed maliciously, this is only done if the transmitted ID actually represents one of the formerly spawned child processes.

3.5 Client side

On startup every client must connect to the server. If this connection fails, the client exits with an error message. A client which is connected to the server asks for the available audio sources first. After that it may request up to 20 separate streams. For each stream a new UDP socket is opened on the client side. The audio processing driver has again been implemented based on JACK. For cross-platform purposes audio drivers based on RtAudio [6] and PortAudio [7] have also been added.

3.5.1 Jitter buffer

In the internet, as in any packet-switched network, packets can be lost or delayed, which in real-time applications is the same when a certain delay is exceeded. Due to its best-effort nature, the internet protocol may even result in duplicated packets. Variations in the transmission delay are taken into account by a jitter buffer which collects packets as they come in and delivers them in order and thus ensures continuous playout of the speech data. Our jitter buffer implementation firstly collects the incoming packets in a separate programming thread and, secondly, delivers all available descriptions of the current frame whenever the data of a frame is requested by the decoder. The size of the jitter buffer is initially set to 500 ms. It grows exponentially when a packet is received which cannot be buffered due to its sequence number being too high. Basically, this means a change of size is not supposed to happen more than once. This is, however, also influenced by the clock skew compensation algorithm which is described in section 3.5.3.

3.5.2 Packet loss simulation

To demonstrate the effect of packet loss on the client side, even if actually not a single packet is lost in the internet, a packet loss simulator has been added. Independent random packet losses are simulated. Packet loss is applied after the received packets have been delivered by the jitter buffer.

3.5.3 Clock skew compensation

On the serving side the senders are synchronized by JACK which is driven by a clock as part of the sound card. On the client side another clock is used to drive the audio processing. Since these two clocks are not synchronous it is highly likely that they do not run at exactly the same rate. With typical clocks this skew can amount to up to $\pm 0.5\%$ [8].

Assuming the audio data on the client side is processed faster than on the sending side, the jitter buffer will eventually run empty. If, on the other hand, packets are sent faster than a client plays out the decoded audio data, the jitter buffer will overflow. We tested our client on serveral systems and in some cases we saw the jitter buffer run empty in only 40 seconds. To counter this problem, we added a cubic spline interpolator as suggested in [8]. Depending on the jitter buffer fill level, which was low-pass filtered for this purpose, the playout is accelerated if the level is too high and slowed down if it is too low. Fortunately this interpolation does not degrade the perceived speech quality.

4 Usage

The client side of the demonstrator is implemented as a graphical user interface (GUI), available for GNU/Linux as well as Microsoft Windows operating systems. Fig. 3 shows a screen shot of the demonstrator operating on Linux.

The parameters of a stream that is to be added can be configured in the upper part of the GUI. Beneath this part, the simulated loss rate can be controlled. Every stream is shown as a row in the table on the left side. In this table, several stream parameters are displayed. By selecting one of these rows, the active stream is determined. At any time, there is exactly one active stream of which the audio data is being played out. The history of the active stream regarding all packet loss, both possible losses in the network and simulated losses, is displayed on the right side. The bit rate actually received is drawn in red. The blue graph displays the residual packet loss ratio which counts those frames for which none of the descriptions has been received.

5 Conclusions

Multiple-description coding is a technique to improve the perceived quality when sending multimedia data over lossy packet-switched networks like the internet. A software architecture for real-time transmission of MD coded speech signals over the internet has been developed. The system allows experimental comparison of different configurations of MD speech codecs under varying channel conditions. It forms an extensible framework for further experiments on MD speech and audio coding in general.

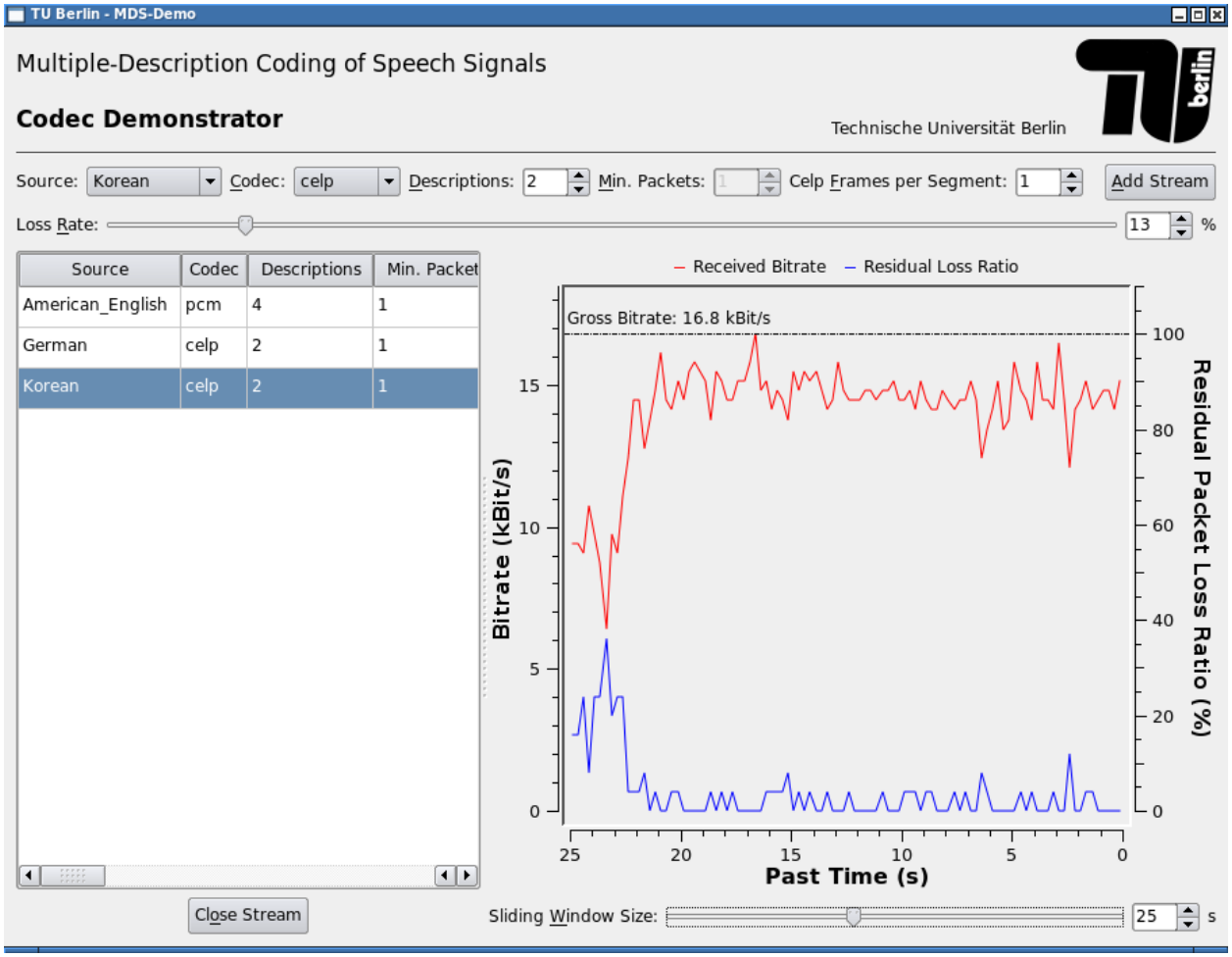


Figure 3: Screen shot of the demonstrator

6 Acknowledgements

The authors would like to thank Rubén Heras Evangelio for providing the initial implementation of the jitter buffer as well as helping with porting the demonstrator to the Windows operating system.

This project was funded by the German Research Foundation (DFG).

References

[1] V. K. Goyal. Multiple description coding: compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, 2001.

[2] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2), April 1997.

[3] Python Software Foundation. Python programming language – official website. http://www.python.org. last checked: 05.01.2007.

[4] P. Davis et al. Jack audio connection kit website. http://www.jackaudio.org. last checked: 05.01.2007.

[5] K. Vehmanen. Ecasound website. http://www.eca.cx/ecasound. last checked: 05.01.2007.

[6] G. P. Scavone. Rtaudio website. http://www.music.mcgill.ca/~gary/rtaudio/. last checked: 07.01.2007.

[7] R. Bencina et al. Portaudio website. http://www.portaudio.com/. last checked: 07.01.2007.

[8] T. Trump. Compensation for clock skew in voice over packet networks by speech interpolation. In *Proceedings of the 2004 International Symposium on Circuits and Systems*, Vancouver, Canada, 2004.

Musical Signal Scripting with PySndObj

Victor Lazzarini
Sound and Digital Music Technology Group,
Music Technology Lab,
NUI, Maynooth
Ireland
Victor.Lazzarini@nuim.ie

Abstract

This article discusses musical signal scripting using a Python language module, PySndObj, based on the Sound Object (SndObj) Library. This module allows for advanced music and audio scripting and provides support for fast application development and on-the-fly synthesis programming. The article discusses the main concepts involved in audio programming with the library. This is complemented by an overview of the PySndObj module with a number of basic examples of its use. The article concludes with the description of a proposed Musical Signal Processing system, which would include the previously discussed SndObj and PySndObj components.

Keywords

Musical Signal Processing, Object-Oriented Programming, Scripting Languages, Music Composition

1 Introduction

The Sound Object (SndObj) Library [1] is an object-oriented[2] audio processing library. It is a collection of classes for synthesis and processing of sound, inspired by the example set by the MUSIC *N* family of programs [3]. These can be used to build applications for computer-generated music. The source code is multi-platform and can be compiled under most C++ compilers. The library is a both an audio programming framework and a fast application development toolkit, with over 100 classes in its current release. For the latter uses, the library is available on C++[4], Java[5] or Python[6]. This article will discuss aspects of audio scripting using the SndObj library Python module, PySndObj.

1.1 What is a SndObj?

A SndObj (pronounced 'Sound Object') is a programming unit that can generate signals with audio or control characteristics. It has a number of basic attributes, such as an output vector, a sampling rate, a vectorsize and an input connection (which points to another SndObj). Depending on the type of SndObj, other attributes will also be featured: an oscillator will have an input connection for a function table, a delayline will have a delay buffer, etc..

SndObjs contain their own output signal. So, at a given time, if we want to obtain the signal it generates, we can probe its output vector. This will contain a vecsize number of samples that the object has generated after it was asked to either process or synthesise a signal. This is a basic characteristic of SndObjs: signals are internal, as opposed to existing in external buffers or busses. SndObjs can interface very easily with external signals, but in a pure SndObj processing chain, signals are internal and hidden.

1.2 Generating output

The basic operation that a SndObj performs is to produce an output signal. This is done by invoking the public member function SndObj::DoProcess(). Each call will generate a new output vector full of samples, so to generate a continuous signal stream, DoProcess() should be invoked repeatedly in a loop (known as the 'processing loop'). Programs will have to feature at least one such loop in order to generate audio signals.

As an alternative to directly programming a loop, users can avail of the services of the SndThread class and its derivatives, which provide processing thread management and a hidden processing loop (see below). The DoProcess() method is overridable, so each different variety of SndObj will implement it differently so that different objects can generate different signals. In

addition, other types of processing might be achieved with some overloaded operators (see below in 'Manipulating SndObjs').

1.3 Connecting SndObjs

Another basic programming concept found in this library is that SndObjs do not have direct signal inputs, because of the fact that signals are internal to them. Instead, they will have input connections to other SndObjs. This way an object will read the output signal of another which is connected to it. Any type of signal input, either a processing input or a parameter modulator input is connected in the same way.

Certain processing parameters will have two types of input: an *offset value* and a *SndObj connection*. The offset value, generally a single floating point value is added to whatever signal the connected SndObj has generated. In most cases, SndObj connections for parameters are optional: if they are not present, then only the offset value is used for it. In this case, they are in fact not an 'offset', but the actual value for the parameter. In other cases, the user will want to set the parameter offset to 0, so that only the SndObj input is used to control that parameter.

1.4 Manipulating SndObjs

Apart from invoking processing, users can manipulate SndObjs in other ways. The first obvious operation is parameter setting, for which different varieties of SndObjs will have different methods. However, a unified message-passing interface is defined by SndObj, with the SndObj::Set() and SndObj::Connect() methods. These can be used to change the status of SndObjs via the various messages defined for them.

Messages are also inherited, so the derived object will have its own set, plus the ones defined for its superclass(es). Set() is used to set offset and single parameter values. Connect() is used to connect input objects, which can be of SndObj, SndIO (input and output objects) or Table (function table objects) types. Messages are string constants. In addition, the output signal buffer can be accessed with a variety of methods such as SndObj::Output(), SndObj::PushIn() and SndObj::PopOut() .

1.5 Input and Output

Signal input and output is handled by SndIOs ('sound ios'), which are objects that can write and read to files, memory, devices, etc. They are modelled in similar ways to SndObjs: signals are internal, use object connections, etc.. However, they are designed to deal with a slightly different type of processing. Their main performing methods are SndIO::Read() and SndIO::Write(). When invoked, these will read or write a vectorsize full of samples from/to their source/destination, respectively. SndIOs can handle multichannel streams, so their output vector actually contains frames of samples (in interleaved format).

SndIOs interact with SndObjs in two basic ways. For signal input, SndIOs can be accessed via SndIn objects. Each channel of input audio has to be connected separately, because SndObjs in general handle only single signal streams. For signal output, SndObjs can be connected directly to SndIOs (again, one for each channel). This can be done at construction time, or more usually using SndIO::SetOutput(). For MIDI input, a number of specialist classes exist, derived from MidiIn, which work in a similar way to SndIn.

1.6 Function Tables

Certain SndObjs, for instance oscillators, will depend heavily on tabulated function tables. For this purpose, a special type of object can be used, a Table object. Tables are very simple objects whose most important attribute is their actual tabulated function, which is created at construction time. Tables can be updated at any time, by changing some of their parameters and invoking Table::MakeTable().

1.7 Frequency-domain issues

The Sound Object Library provides classes for time and frequency-domain (spectral) processing. For the latter, a few special considerations must be made. Time-domain and spectral SndObjs are designed to fit in together very snuggly in a processing chain. For this reason, a certain model was employed, which slightly limits the arrangement of such SndObjs.

For spectral processing, the FFT size must be always power-of-two multiple of the hopsize (usually a minimum four times that value). When connecting time- and frequency-domain SndObjs, the hopsize must be the same as the time-domain vectorsize. Generally for an efficient FFT, the analysis size is set to a power-of-two value. So, in practice, this limits the vectorsize/hopsize and FFT size values to a limited pairing of values. Although at first this looks limiting, it will in fact have little impact of the flexibility of spectral processing using the library. This model, in turn, will facilitate immensely the interaction between

frequency- and time-domain SndObjs. Effectively, if these conditions are met, they can be inter-connected transparently, even though they are dealing with very different types of signals.

1.8 Processing threads

In addition to the basic types of objects discussed above, the Sound Object Library also includes a special thread management class, SndThread. With this type of object, a pthread-library based thread can be instantiated and run. This object encapsulates the main processing loop, calling the basic performing methods of each object that has been add to it.

Using SndThreads ('sound threads') is very simple. Once an object has been created and a chain of SndObjs/SndIOs has been defined, a processing list is initialised using SndThread::AddObj() or SndThread::Insert(). To start processing a signal, SndThread::ProcOn() is invoked. To stop processing, SndThread::Proc-Off() can be used. SndObjs can be deleted from the processing list using SndThread::DeleteObj(). Multiple SndThreads can be used for parallel processing with SndBuffer objects being used to obtain the signals from each thread.

2 PySndObj

PySndObj is a python module that wraps the SndObj C++ code in a very useful way to provide support for Python scripting. It allows for a nice scripting interface to the library for fast application development, prototyping of applications and general on-the-fly synthesis and processing.

PySndObj can be added, provided you have the _sndobj dynamic module (.so on Linux, .dylib on OSX and .dll on Windows) and csnd.py (the python bindings) in the right places (check your Python documentation), using the import command:

```
import sndobj
```

or

```
from sndobj import *
```

This will allow access to all SndObj library classes available to your platform, plus some extra utility classes for array support. The latter form allows for accessing the SndObj classes directly, without the package name as a prefix ('namespace'). For sake of simplicity and economy of space, we will be using the latter form as the basis for all further code examples. However it is important to point out that the recommended Python coding style is to explicitely use namespaces.

2.1 Python SndObj classes and objects

Python SndObj classes look very similar to their C++ counterparts. The main difference is that in Python all objects are dynamically allocated, so they are equivalent to C++ pointers to objects. Since the library uses pointers to connect object (See 'Programming Concepts'), using SndObjs in Python is very straightforward and transparent.

Connecting objects is very simple. Let's say we want to create a sine wavetable and connect an oscillator to it:

```
tab = HarmTable()
osc = Oscili(tab,440, 16000)
```

Here as variables hold object pointers, we have the case where 'tab' can be passed directly to osc, with no need for any extra complications. The same works for SndObj and SndIO connections, so if we set up a RT output object, we can connect our oscillator to it:

```
outp = SndRTIO(1)
outp.SetOutput(1,osc)
```

This works between SndObjs as we would expect, if we want to, say, connect a modulator to our oscillator:

```
mod = Oscili(tab, 2, 44)
osc.SetFreq(440, mod)
```

2.2 Running SndObjs

In order to get audio processing out of a SndObj, it is necessary to invoke its DoProcess() method. This runs the processing once and produces an output vector full of samples, eg.

```
osc.DoProcess()
```

For a continuous output, continued calls to DoProcess() are required, so we need to set up a loop, where this can happen. In addition, any SndIOs in the chain have also to call their Read() or Write() methods (for input or output respectively).

```
# 10 seconds of audio output
timecount = 0
end = 10 * osc.GetSr()
```

```
vecsize = osc.GetVectorSize()
while(timecount < end):
  mod.DoProcess()
  osc.DoProcess()
  outp.Write()
timecount += vecsize
```

2.3 Using a processing thread

However, the simplest way to get SndObjs producing audio is to use a SndThread object to control the synthesis. This will take care of setting up a processing loop and call all the required methods. We start by setting the object up:

```
thread = SndThread()
thread.AddObj(mod)
thread.AddObj(osc)
thread.AddObj(outp, SNDIO_OUT)
```

Then we can turn on the processing to get some audio out:

```
thread.ProcOn()
```

When we are done with it, we can turn it off:

```
thread.ProcOff()
```

In addition to SndThread, it is also possible to use SndRTThread, which has default realtime objects for input and output that can be connected to. In this case the user only needs to set up his/her SndObj chain and add this to the thread object.

Any asynchronous Python code can also be called, by setting up a process callback, that will be invoked once every processing period. This happens after the input signal(s) has been obtained, but before any processing by SndObjs.

For instance, if we have a method:

```
def callb(data):
    ... # callback code
```

This sets the callback, `data` is any data object to be passed to the callback:

```
thread.SetProcessCallback(callb,data)
```

The callback can be used for things like updating a display, changing parameters cyclically, polling for control input, etc..

2.4 Support for arrays

In order to facilitate certain ways of programming and to make possible the use of C arrays with the library, some utility classes have been added for int, float and double arrays, named, respectively: intArray, floatArray and doubleArray. These classes can be used as follows

```
# create an array of two items
f = floatArray(2)
# array objects can be manipulated
# by index as in C
f[0] = 2.5
```

In addition, a special type of array is also available, the sndobjArray, which holds SndObjs (internally C++ SndObj pointers). Objects of this type can be used similarly to the above array:

```
objs = sndobjArray(2)
objs[0] = mod
objs[1] = osc
```

However, when these are used as SndObj pointer arrays, they will need to be cast as that. Hopefully the class has a handy method for doing just that:

```
objp = objs.cast()
```

These can be used with objects that take arrays of SndObjs as input, such as SndThread:

```
thread = SndThread(2, objp, outp)
```

in which case we are setting up a thread with two SndObjs (which is similar to the example above). Other objects that take SndObj arrays are for instance SndIO-derived objects and Mixer SndObjs. But remember, SndObj arrays are not sndobjArray objects, but can be retrieved using sndobjArray::cast().

3 Simple examples

Two simple scripts are provided here for realtime audio processing and synthesis.

3.1 Simple echo using a comb filter

This example demonstrates realtime audio IO and some delayline processing. The example uses a SndRTThread object as introduced above. This takes care of all realtime input/output.

```
from sndobj import *
import time
import sys

if len(sys.argv) > 1:
  dur = sys.argv[1]
else:
  dur = 60

# SndRTThread object has its own
# IO objects.
# By the default it is created wth
```

```
# 2 channels
t = SndRTThread(2)

# Echo objects take input from
# SndRTThread inputs
comb_left = Comb(0.48,0.6,t.GetInput(1))
comb_right = Comb(0.52, 0.6,t.GetInput(1))

# We now add the echo objects to
# the output channels
t.AddOutput(1, comb_left)
t.AddOutput(2, comb_right)

# This connects input to output
# directly
t.Direct(1)
t.Direct(2)

# turn on processing for
# dur seconds
t.ProcOn()
time.sleep(float(dur))
t.ProcOff()
```

3.2 Oscillator with GUI (using wxPython)

Here we present a complete GUI-based synthesis program (albeit a trivial one). This demonstrates how a GUI toolkit (such as wxPython[7]) can be used to create complete computer instruments.

```
from sndobj import *
from wxPython.wx import *
import traceback
import time

class ControlPanel(wxPanel)
# Override the base class
# constructor
def __init__(self, parent):
wxPanel.__init__(self, parent, -1)
self.ID_BUTTON1 = 10
self.button1 = wxButton(self,     \
    self.ID_BUTTON1, "On/Off",    \
    (20, 20))
# Bind the button to its event
# handler.
 EVT_BUTTON(self, self.ID_BUTTON1,\
  self.OnClickButton1)
# Create a slider to change pitch
 self.ID_SLIDER1 = 20
 self.slider1 = wxSlider(self,    \
   self.ID_SLIDER1, 300, 200, 400,\
   (20, 50), (200,50),            \
     wxSL_HORIZONTAL | wxSL_LABELS)
 self.slider1.SetTickFreq(5, 1)
# Bind the slider to its event
#  handler.
  EVT_SLIDER(self, self.ID_SLIDER1, \
        self.OnSlider1Move)
EVT_CLOSE(parent, self.OnClose)
# Default pitch.
self.pitch = 300
# Sine wave table
self.tab = HarmTable()
# Envelope (just attack actually)
self.line = Interp(0, 10000, 0.05)
# oscil
self.osc = Oscili(self.tab,   \
   self.pitch, 0, None, self.line)
self.out = SndRTIO(1,SND_OUTPUT)
self.out.SetOutput(1, self.osc)
self.thread = SndThread()
self.thread.AddObj(self.line)
self.thread.AddObj(self.osc)
self.thread.AddObj(self.out, SNDIO_OUT)
self.play = False

  def OnClickButton1(self, event):
  if(self.play):
# create an envelope decay
self.line.SetCurve(10000, 0)
   self.line.Restart()
   self.play = False
 else:
# create an envelope attack
   self.line.SetCurve(0, 10000)
   self.line.Restart()
   self.play = True
   self.thread.ProcOn()

# slider movement
def OnSlider1Move(self, event):
  self.pitch = event.GetInt()
  self.osc.SetFreq(self.pitch)

# stop performance
def OnClose(self, event):
   try:
     self.thread.ProcOff()
     time.sleep(1)
     self.GetParent().Destroy()
   except:
     print traceback.print_exc()

# Create a wx application.
application= wxPySimpleApp()
# Create a parent frame
frame= wxFrame(None,-1,"PySndObj example")
# Create the controls
controlPanel= ControlPanel(frame)
# Display the frame.
frame.Show(True)
# Run the application.
application.MainLoop()
```

4 Towards a Musical Signal Processing System

The SndObj library and PySndObj, in fact, form part of a larger picture of a proposed system that will incorporate a third software component in the form of graphic 'patching application'. These three elements would then form a three-layer Musical Signal Processing System (fig.1), allowing different entry levels of user interaction.

At the top, the patching application will provide a GUI layer. At this level, users can set up patches of SndObjs, event patterns, automation, etc., without the use (or at least with reduced use) of textual declarations.

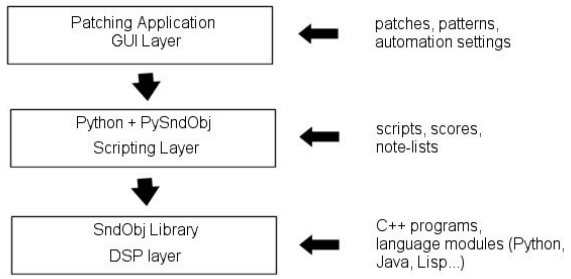


Figure 1. The three levels of a proposed Musical Signal Processing System

The middle layer is provided by Python and PySndObj (possibly with some Python-written add-ons for event processing and control). This provides the interface between the patching application (itself written in Python) and the lower-level SndObj library. Users will be able to use this layer directly, bypassing (or combining with) the patching application, for scores and, more general, scripting (for instance, alternative top-level applications can be written using this layer).

The lower level is then provided by the SndObj library itself, the DSP engine for the system. This layer can be accessed directly by the user in C++ code, providing standalone applications, modules for languages (similarly to PySndObj, for Java, Lisp, etc...).

Of the three components for this proposed system, the lower end is complete and functional. The middle layer (as discussed in this article) is functional, but perhaps needing some extensions for better event processing support. The top level does not exist yet, although some idea of how it might look like was provided by the AIDE software[8], developed using the SndObj library.

5 Conclusion

PySndObj provides a good support for audio processing in Python. The simplicity of the language, allied to the modularity and comprehesiveness of the library proves to be a powerful combination.

All of the SndObj tools are Free software, GPL licensed, and are available from sourceforge download (full releases) or anonymous CVS at:

http://sndobj.sf.net

Developers are also encouraged to join the project and can do so by contacting the author at his e-mail address.

6 References

[1] V Lazzarini. 2000. The Sound Object Library. *Organised Sound 5 (1),* pages 35-49. Cambridge Univ. Press., Cambridge.

[2] M Abadi and L Cardelli. 1996. *A Theory of Objects*, Springer-Verlag, New York.

[3] C Dodge and T Jerse. 1985. *Computer Music: Synthesis, Composition and Performance.* Schirmer Books, New York.

[4] B Stroustrop. 1991. *The C++ Programming Language*, second edition. Addison-Wesley, New York.

[5] K Arnold and J Gosling. 1996. *The Java Programming Language.* Addison-Wesley, New York.

[6] G Van Rossum and F Drake. 2003. *The Python Language Reference Manual*. Network Theory, Bristol.

[7] http://www.wxwidgets.org

[8] V Lazzarini and R Walsh. 2004. AIDE, a New digital audio effects development environment. *Proc. of the 7th Int. Conference on Digital Audio Effects (DAFx-04*), pages 53-57. Univ.of Naples, Naples.

Interfacing Pure Data with Faust

Albert GRÄF
Dept. of Music Informatics, Johannes Gutenberg University
55099 Mainz, Germany,
ag@muwiinfa.geschichte.uni-mainz.de

Abstract

This paper reports on a new plugin interface for Grame's functional DSP programming language Faust. The interface allows Faust programs to be run as externals in Miller Puckette's Pd (Pure Data), making it possible to extend Pd with new audio objects programmed in Faust. The software also includes a script to create wrapper patches around Faust units which feature "graph-on-parent" GUI elements to facilitate the interactive control of Faust units. The paper gives a description of the interface and illustrates its usage by means of a few examples.

Keywords

Computer music, digital signal processing, Faust programming language, functional programming, Pd, Pure Data

1 Introduction

Faust is a modern-style functional language for programming digital signal processing algorithms being developed at Grame [1; 2], which was already presented in-depth at last year's Linux Audio Conference [3]. Faust provides an executable, high-level specification language for describing block diagrams operating on audio signals. Signals are modelled as functions of (discrete) time and DSP algorithms as higher-order functions operating on signals. The main advantages of this approach over using graphical block diagrams is that the building blocks of signal processing algorithms can be combined in much more flexible ways, and that Faust can also serve as a formal specification language for signal processing units.

Faust programs are compiled to efficient C++ code which can be used in various environments, including Jack, LADSPA, Max/MSP, SuperCollider, VST and the Q programming language. This paper reports on Faust's plugin interface for Miller Puckette's *Pure Data* a.k.a. *Pd* (`http://puredata.info`). The new interface allows audio developers and Pd users to run Faust programs as Pd externals, in order to test Faust programs using Pd's convenient graphical environment, or to extend Pd with new custom audio objects. The package also includes a Q script `faust2pd` which can create wrapper patches featuring "graph-on-parent" GUIs around Faust externals, which further facilitates the interactive control of Faust units in the Pd environment.

The software described in this paper is free (GPL'ed). It is already included in recent Faust releases (since version 0.9.8.6), and is also available as a separate package `faust2pd` from `http://q-lang.sf.net`, which includes the `puredata.cpp` Faust architecture file, the `faust2pd` script and supporting Pd abstractions, as well as a bunch of examples. You can also try Faust interactively, without having to install the Faust compiler, at Grame's Faust website (`http://faust.grame.fr`).

Because of lack of space we cannot give an introduction to Faust and Pd here, so the paper assumes a passing familiarity with both. More information about these systems can be found in the documentation available at `http://faust.grame.fr` and `http://puredata.info`.

2 Building Faust externals

Faust Pd plugins work in much the same way as the well-known `plugin~` object (which interfaces to LADSPA plugins), except that each Faust DSP is compiled to its own Pd external. Under Linux, the basic compilation process is as follows (taking the freeverb module from the Faust distribution as an example):

```
# compile the Faust source to a C++ module
# using the "puredata" architecture
faust -a puredata.cpp freeverb.dsp
  -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp
  -o freeverb~.pd_linux
```

By these means, a Faust DSP named `XYZ` with N audio inputs and M audio outputs becomes a Pd object `XYZ~` with $N+1$ inlets and $M+1$ outlets. The leftmost inlet/outlet pair is for control messages only. This allows you to inspect and change the controls the unit provides, as detailed below. The remaining inlets and outlets are the audio inputs and outputs of the unit, respectively. For instance, `freeverb.dsp` becomes the Pd object `freeverb~` which, in addition to the control inlet/outlet pair, has 2 audio inputs and outputs.

When creating a Faust object it is also possible to specify, as optional creation parameters, an extra unit name (this is explained in the following section) and a sample rate. If no sample rate is specified explicitly, it defaults to the sample rate at which Pd is executing. (Usually it is not necessary or even desirable to override the default choice, but this might occasionally be useful for debugging purposes.)

In addition, there is also a Q script named `faust2pd`, described in more detail below, which allows you to create Pd abstractions as "wrappers" around Faust units. The wrappers generated by `faust2pd` can be used in Pd patches just like any other Pd objects. They are much easier to operate than the "naked" Faust plugins themselves, as they also provide "graph-on-parent" GUI elements to inspect and change the control values.

Note that, just as with other Pd externals and abstractions, the compiled `.pd_linux` modules and wrapper patches must be put somewhere where Pd can find them. To these ends you can either move the files into the directory with the patches that use the plugin, or you can put them into the `lib/pd/extra` directory or some other directory on Pd's library path for system-wide use.

3 The control interface

Besides the DSP algorithm itself, Faust programs also contain an abstract "user interface" definition from which the control interface of a Faust plugin is constructed. The Faust description of the user interface comprises various abstract GUI elements such as buttons, checkboxes, number entries and (horizontal and vertical) sliders as well as the initial value and range of the associated control values, which are specified in the Faust source by means of the builtin functions `button`, `checkbox`, `nentry`, `hslider` and `vslider`. Besides these "active" elements which are used to input control values into the Faust program, there are also "passive" elements (`hbargraph`, `vbargraph`) which can be used to return control values computed by the Faust program to the client application.

It is also possible to specify a hierarchical layout of the GUI elements by means of appropriate "grouping" elements which are implemented by the Faust functions `hgroup`, `vgroup` and `tgroup` (`hgroup` and `vgroup` are for horizontal and vertical layouts, respectively, whereas `tgroup` is intended for "tabbed" layouts). Each GUI element (including the grouping elements) has an associated label (a string) by which the element can be identified in the client application. More precisely, each GUI element is uniquely identified by the path of labels in the hierachical layout which leads up to the given element. For further details we refer the reader to the Faust documentation [4].

To implement the control interface on the Pd side, the control inlet of a Faust plugin understands a number of messages which allow to determine the available controls as well as change and inspect their values:

- The `bang` message reports all available controls of the unit on the control outlet. The message output for each control contains the type of control as specified in the Faust source (`checkbox`, `nentry`, etc.), its (fully qualified) name, its current value, and its initial, minimum, maximum and stepsize values as specified in the Faust source.
- The `foo 0.99` message sets the control `foo` to the value 0.99, and outputs nothing.
- Just `foo` outputs the (fully qualified) name and current value of the `foo` control on the control outlet.

Control names can be specified in their fully qualified form (giving the complete path of a control, as explained above), like e.g. `/gnu/bar/foo` which indicates the control `foo` in the subgroup `bar` of the topmost group `gnu`, following the hierarchical group layout defined in the Faust source. This lets you distinguish between different controls with the same name which are located in different groups. To find out about all the controls of a unit and their fully qualified names, you can bang the control inlet of the unit as described above, and connect its control outlet to a `print` object, which will cause the descriptions of all controls to be

printed in Pd's main window. (The same information can also be used, e.g., to initialize Pd GUI elements with the proper values. Patches generated with `faust2pd` rely on this.)

You can also specify just a part of the control path (like `bar/foo` or just `foo` in the example above) which means that the message applies to *all* controls which have the given pathname as the final portion of their fully qualified name. Thus, if there is more than one `foo` control in different groups of the Faust unit then sending the message `foo` to the control inlet will report the fully qualified name and value for each of them. Likewise, sending `foo 0.99` will set the value of all controls named `foo` at once.

Concerning the naming of Faust controls in Pd you should also note the following:

- A unit name can be specified at object creation time, in which case the given symbol is used as a prefix for all control names of the unit. E.g., the control `/gnu/bar/foo` of an object `baz~` created with `baz~ baz1` has the fully qualified name `/baz1/gnu/bar/foo`. This lets you distinguish different instances of an object such as, e.g., different voices of a polyphonic synth unit.
- Pd's input syntax for symbols is rather restrictive. Therefore group and control names in the Faust source are mangled into a form which only contains alphanumeric characters and hyphens, so that the control names are always legal Pd symbols. For instance, a Faust control name like `"meter #1 (dB)"` will become `meter-1-dB` which can be input directly as a symbol in Pd without any problems.
- "Anonymous" groups and controls (groups and controls which have empty labels in the Faust source) are omitted from the path specification. E.g., if `foo` is a control located in a main group with an empty name then the fully qualified name of the control is just `/foo` rather than `//foo`. Likewise, an anonymous control in the group `/foo/bar` is named just `/foo/bar` instead of `/foo/bar/`.

Last but not least, there is also a special convenience control named `active` which is generated automatically. The default behaviour of this control is as follows:

- When `active` is nonzero (the default), the unit works as usual.
- When `active` is zero, and the unit's number of audio inputs and outputs match, then the audio input is simply passed through.
- When `active` is zero, but the unit's number of audio inputs and outputs do *not* match, then the unit generates silence.

The `active` control frequently alleviates the need for special "bypass" or "mute" controls in the Faust source. However, if the default behaviour of the generated control is not appropriate you can also define your own custom version of `active` explicitly in the Faust program; in this case the custom version will override the default one.

4 Basic example

Let's take a look at a simple example to see how these Faust externals actually work in Pd. The patch shown on the right of Figure 1 features a Faust external `tone~` created from the Faust source shown on the left of the figure. The Faust program implements a simple DSP, a sine oscillator with zero audio inputs and stereo (i.e., two) audio outputs, which is controlled by means of three control variables `vol` (the output volume), `pan` (the stereo panning) and `pitch` (the frequency of the oscillator in Hz). Note that in the patch the two audio outlets of the `tone~` unit are connected to a `dac~` object so that we can listen to the audio output produced by the Faust DSP.

Several messages connected to the control inlet of the `tone~` object illustrate how to inspect and change the control variables. For instance, by sending a `bang` to the control inlet, we obtain a description of the control parameters of the object printed in Pd's main window, which in this case looks as follows:

```
print: nentry /faust/pan 0.5 0.5 0 1 0.01
print: nentry /faust/pitch 440 440 20 20000 0.01
print: nentry /faust/vol 0.3 0.3 0 10 0.01
```

Clicking the `vol 0.1` message changes the `vol` parameter of the unit. We can also send the message `vol` to show the new value of the control, which is reported as follows:

```
print: /faust/vol 0.1
```

```
import("music.lib");

// control variables

vol     = nentry("vol", 0.3, 0, 10, 0.01);
pan     = nentry("pan", 0.5, 0, 1, 0.01);
pitch   = nentry("pitch", 440, 20, 20000, 0.01);

// simple sine tone generator

process = osci(pitch)*vol : panner(pan);
```

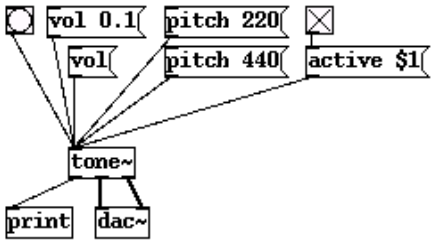


Figure 1: Basic Faust example

In the same fashion we can also set the `pitch` control to change the frequency of the oscillator. Moreover, the `active` control (which is not defined in the Faust source, but created automatically by the Pd-Faust plugin interface) allows to switch the unit on and off. The example patch allows this control to be operated by means of a toggle button.

5 Wrapping Faust DSPs with faust2pd

Controlling bare Faust plugins in the way sketched out in the preceding section can be a bit cumbersome, so the `faust2pd` package also provides a Q script `faust2pd.q` which can generate "wrapper" patches featuring additional graph-on-parent GUIs. Most of the sample patches in the `faust2pd` package were actually created that way. To use the script, you'll also need the Q interpreter available from `http://q-lang.sf.net`. The `faust2pd` package contains instructions on how to install the script and the supporting Pd abstractions on your system.

The graph-on-parent GUIs of the wrapper patches are *not* created from the Faust source or the compiled plugin, but from the XML descriptions (`dsp.xml` files) Faust generates when it is run with the `-xml` option. Such an XML file contains a readable description of the complete hierarchy of the control elements defined in the Faust program, and includes all necessary information to create a concrete rendering of the abstract user interface in the Faust source. The `faust2pd` script is able to read this XML description and create the corresponding Pd GUI along with the necessary control logic.

The script is run as `faust2pd filename.dsp.xml`; this will create a Pd patch named `filename.pd` from the Faust XML description in `filename.dsp.xml`. The `faust2pd` program understands a number of options which affect the layout of the GUI elements and the contents of the generated patch; you can also run `faust2pd -h` for information about these additional options.

On Linux, the compilation of a Faust DSP and creation of the Pd patch typically involves the following steps (again taking the freeverb module from the Faust distribution as an example):

```
# compile the Faust source and generate
# the xml file
faust -a puredata.cpp -xml freeverb.dsp
  -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp
  -o freeverb~.pd_linux
# generate the Pd patch from the xml file
faust2pd freeverb.dsp.xml
```

Just like the Faust plugin itself, the generated patch has a control input/output as the leftmost inlet/outlet pair, and the remaining plugs are signal inlets and outlets for each audio input/output of the Faust unit. However, the control inlet/outlet pair works slightly different from that of the Faust plugin. Instead of being used for control replies, the control outlet of the patch simply passes through its control input (after processing messages which are understood by the wrapped plugin). By these means control messages can flow along with the audio signal through an entire chain of Faust units. Moreover, when generating a polyphonic synth patch using the `-n` a.k.a. `--nvoices` option there will actually be two control inlets, one for note messages and one for ordinary control messages. (This is illustrated by the examples in the following section.)

The generated patch also includes the necessary GUI elements to see and change all (active and passive) controls of the Faust unit. Faust control elements are mapped to Pd GUI elements in an obvious fashion, following the hori-

zontal and vertical layout specified in the Faust source. The script also adds special buttons for resetting all controls to their defaults and to operate the special `active` control.

This generally works very well, but you should be aware that the control GUIs generated by `faust2pd` are somewhat hampered by the limited range of GUI elements available in a vanilla Pd installation:

- There are no real "button" widgets as required by the Faust specification, so "bangs" are used instead. There is a global delay time for switching the control from 1 back to 0, which can be changed by sending a value in milliseconds to the `faust-delay` receiver. If you need interactive control over the switching time then it is better to use checkboxes instead, or you can have `faust2pd` automatically substitute checkboxes for all buttons in a patch by invoking it with the `-f` a.k.a. `--fake-buttons` option.
- Sliders in Pd do not display their value in numeric form so it may be hard to figure out what the current value is. Therefore `faust2pd` has an option `-s` a.k.a. `--slider-nums` which causes it to add a number box to each slider control. (This flag also applies to Faust's passive bargraph controls, as these are implemented using sliders, see below.)
- Pd's sliders also have no provision for specifying a stepsize, so they are an awkward way to input integral values from a small range. On the other hand, Faust doesn't support the "radio" control elements which Pd provides for that purpose. As a remedy, `faust2pd` allows you to specify the option `-r MAX` (a.k.a. `--radio-sliders=MAX`) to indicate that sliders with integral values from the range `0..MAX-1` are to be mapped to corresponding Pd radio controls.
- Faust's "bargraphs" are emulated using sliders. Note that these are passive controls which just display a value computed by the Faust unit. A different background color is used for these widgets so that you can distinguish them from the ordinary (active) slider controls. The values shown in passive controls are sampled every 40 ms by default. You can change this value by sending an appropriate message to the global `faust-timer` receiver.
- Since Pd has no "tabbed" (notebook-like) GUI element, Faust's "tgroups" are mapped to "hgroups" instead. It may be difficult to present large and complicated control interfaces without tabbed dialogs, though. As a remedy, you can control the amount of horizontal or vertical space available for the GUI area with the `-x` and `-y` (a.k.a. `--width` and `--height`) options and `faust2pd` will then try to break rows and columns in the layout to make everything fit within that area.
- You can also exclude certain controls from appearing in the GUI using the `-X` option. This option takes a comma-separated list of shell glob patterns indicating either just the names or the fully qualified paths of Faust controls which are to be excluded from the GUI. For instance, the option `-X 'volume,meter*,faust/resonator?/*'` will exclude all `volume` controls, all controls whose names start with `meter`, and all controls in groups matching `faust/resonator?`.
- Faust group labels are not shown at all, since there seems to be no easy way to draw some kind of labelled frame in Pd.

Despite these limitations, `faust2pd` appears to work rather well, at least for the kind of DSPs found in the Faust distribution. Still, for more complicated control surfaces and interfaces to be used on stage you'll probably have to edit the generated GUI layouts by hand.

6 Faust2pd examples

Figure 2 shows the Faust program of a simple chorus unit. On the right side of the figure you see the corresponding object generated with `faust2pd` with its graph-on-parent area, as it is displayed in a parent patch. The object has three inlet/outlet pairs, one for the control messages and two for the stereo input and output signals. For this abstraction, we ran `faust2pd` with the `-s` a.k.a. `--slider-nums` options so that each `hslider` control in the Faust source is represented by a pair of horizontal slider and number GUI elements in the Pd patch.

If you open the `chorus` object inside Pd you can have a closer look at the contents of the patch (Figure 3). Besides the graph-on-parent area with the GUI elements, it contains the `chorus~` external itself along with inlets/outlets and receivers/senders for the control and audio

```
import("music.lib");

level   = hslider("level", 0.5, 0, 1, 0.01);
freq    = hslider("freq", 2, 0, 10, 0.01);
dtime   = hslider("delay", 0.025, 0, 0.2, 0.001);
depth   = hslider("depth", 0.02, 0, 1, 0.001);

tblosc(n,f,freq,mod)
        = (1-d)*rdtable(n,waveform,i&(n-1)) +
          d*rdtable(n,waveform,(i+1)&(n-1))
with {
  waveform        = time*(2.0*PI)/n : f;
  phase           = freq/SR : (+ : decimal) ~ _;
  modphase        = decimal(phase+mod/(2*PI))*n;
  i               = int(floor(modphase));
  d               = decimal(modphase);
};

chorus(d,freq,depth)    = fdelay(1<<16, t)
with { t = SR*d/2*(1+depth*tblosc(1<<16, sin, freq, 0)); };

process                 = vgroup("chorus", (c, c))
with { c(x) = x+level*chorus(dtime,freq,depth,x); };
```

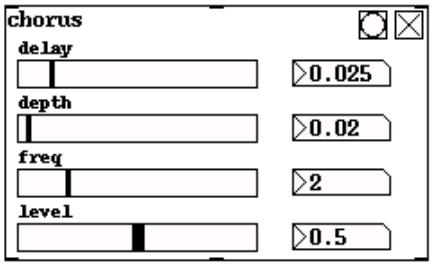


Figure 2: Faust chorus patch

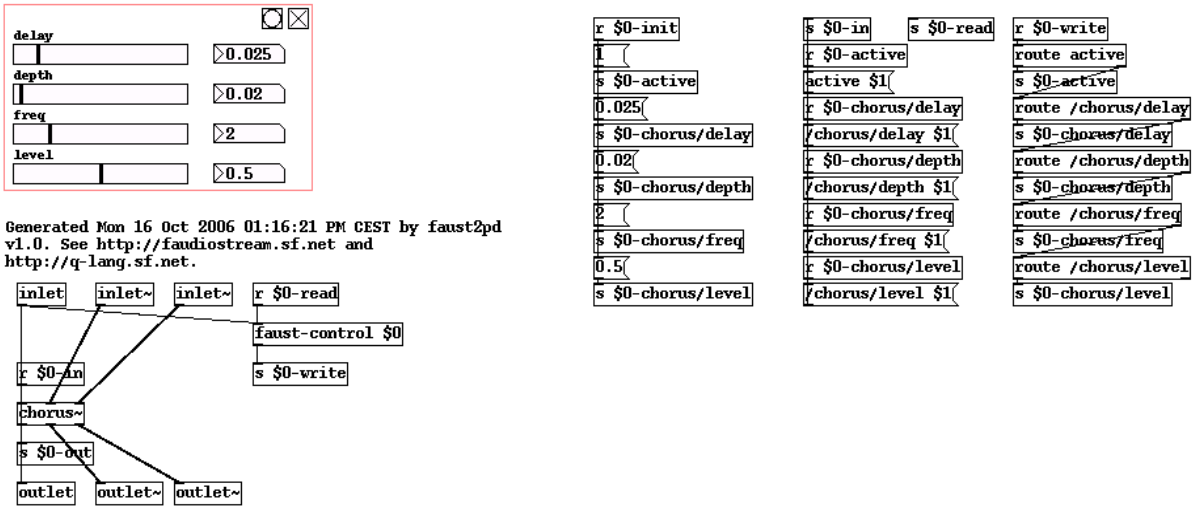


Figure 3: Inside the chorus patch

inputs and outputs (on the left side of the figure, below the GUI area) and the control logic for the GUI elements (on the right side). Of course, the generated contents of the patch can also be edited manually as needed.

It is also possible to generate polyphonic synth patches. Figure 4 shows a simple example, an additive synthesizer. On this Faust program we invoked `faust2pd` with the `-n` a.k.a. `--nvoices` option which specifies the desired number of voices. The generated abstraction then contains as many instances of the Faust external as given with the `-n` option. The resulting patch does not have any audio inputs, but *two* control inputs instead of one. While the right control inlet takes Faust control messages which are sent to all the Faust objects (a.k.a. "voices") simultaneously, the left inlet takes triples of numbers consisting of a voice number, a note number and a velocity value and translates these to the appropriate `freq`, `gain` and `gate` messages for the corresponding voice.

(At this time, the names of the three special voice controls are hard-wired into the `faust2pd`

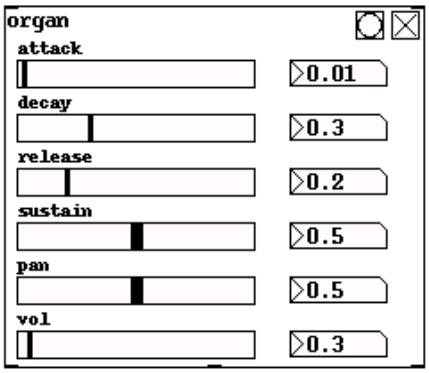
```
import("music.lib");

// control variables

vol     = hslider("vol", 0.3, 0, 10, 0.01);
pan     = hslider("pan", 0.5, 0, 1, 0.01);
attack  = hslider("attack", 0.01, 0, 1, 0.001);
decay   = hslider("decay", 0.3, 0, 1, 0.001);
sustain = hslider("sustain", 0.5, 0, 1, 0.01);
release = hslider("release", 0.2, 0, 1, 0.001);

// voice controls

freq    = nentry("freq", 440, 20, 20000, 1);
gain    = nentry("gain", 0.3, 0, 10, 0.01);
gate    = button("gate");
```




```
// additive synth: 3 sine oscillators with adsr envelop

process = (osc(freq)+0.5*osc(2*freq)+0.25*osc(3*freq))
  * (gate : vgroup("1-adsr", adsr(attack, decay, sustain, release)))
  * gain : vgroup("2-master", *(vol) : panner(pan));
```

Figure 4: Faust organ patch

script, so Faust programs must follow this standard interface if they are to be used as synth units.)

Both kinds of patches can then easily be arranged to the usual synth-effect chains, as shown in Figure 5. In this example we combined the organ and chorus patches from above with another effect unit generated from the freeverb module in the Faust distribution, and added a frontend which translates incoming MIDI messages and a backend which handles the audio output and displays a dB meter. (The latter two components are just plain Pd abstractions.)

These sample patches can all be found in the `faust2pd` package, along with a bunch of other instructive examples, including the full collection of example DSPs from the Faust distribution, more polyphonic synth examples and a pattern sequencer demo.

7 Conclusion

The Pd-Faust external interface and the `faust2pd` script described in this paper make it easy to extend Pd with new audio processing objects without having to resort to C programming. Faust programs are concise and comparatively easy to write (once the initial learning curve has been mastered), and can easily be ported to other plugin architectures such as LADSPA and VST by simply recompiling the Faust source. Still the efficiency of the generated code can compete with carefully handcoded C routines, and sometimes even outperform these, because of the sophisticated optimizations applied by the Faust compiler.

The Pd-Faust interface is especially useful for DSPs which cannot be implemented directly in Pd in a satisfactory manner, like the Karplus-Strong algorithm, because of Pd's 1-block minimum delay restriction for feedback loops. But it is also suitable for implementing all kinds of specialized DSP components like filter designs which could also be done directly in Pd but not with the same efficiency. Last but not least, the interface also gives you the opportunity to make use of the growing collection of readily available Faust programs for different audio processing needs.

The Pd-Faust interface is of course only suitable for creating audio objects. However, there is also a companion *Pd-Q* plugin interface for the Q programming language [5], also available at `http://q-lang.sf.net`. Together, the `faust2pd` package and Pd-Q provide a complete functional programming environment for extending Pd with custom audio and control objects.

Future work on Faust will probably concentrate on making the language still more flexible and easy to use, on providing an extensive collection of DSP algorithms for various purposes,

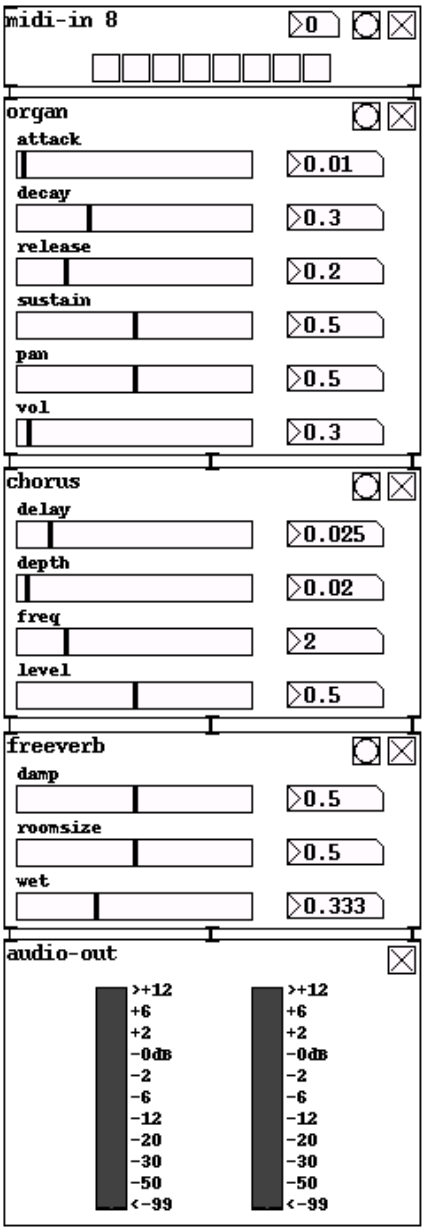


Figure 5: Synth-effects chain

and on adding support for as many target DSP architectures and platforms as possible. Considering the considerable size of these tasks, contributions (especially Faust implementations of common DSP algorithms, and additional plugin architectures) are most welcome. Interested audio developers are invited to join the Faust community at `http://faust.grame.fr`.

References

[1] Y. Orlarey, D. Fober, and S. Letz. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMC 2002)*. International Computer Music Association, 2002.

[2] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.

[3] Yann Orlarey, Albert Gräf, and Stefan Kersten. DSP programming with Faust, Q and SuperCollider. In *Proceedings of the 4th International Linux Audio Conference (LAC06)*, pages 39–47, Karlsruhe, 2006. ZKM.

[4] Yann Orlarey. Faust quick reference. Technical report, Grame, 2006.

[5] Albert Gräf. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference (LAC05)*, pages 21–28, Karlsruhe, 2005. ZKM.

Getting Linux to produce Music fast and powerful

Hartmut NOACK
www.linuxuse.de/snd
Max-Steinke-Strasse 23
13086 Berlin,
BRD,
zettberlin@linuxuse.de

Abstract

At the LAC 2006 I introduced a plan to build a PC to serve as a Linux Audio Workstation (L.A.W.). Now we have a prototype of this machine, that we would like to demonstrate. The box shall be displayed, tried and reviewed by visitors of the conference to find out, how its concept can yield its intended results. The machine has a set of essential GUI-oriented Linux audio software that works with jackd and is integrated with scripts, presets and templates that allow to start complex scenarios with a single click and comes with extensive user-oriented documentation.

Keywords

usability, integration, suite, interface, workstation

1 Introduction

Musicians often state that they don't know very much about computers, that they only want to use these boxes and that they have neither the time nor motivation to learn how a computer or software works. Proprietary software vendors try to adapt to such attitudes by designing all-in-one applications, such as Steinberg's Cubase, with simple-looking interfaces and automatic setup assistants that leave power usage and fine tuning to the experienced users and hide possible options and - of course - the source code from the "end user" to keep their products under control and consistent.

This is not the way that Linux audio can go, since it is free/libre open source(floss). Free software authors, as well as the distributors, need to develop other ways to achieve usability and a trustworthiness that meets the needs of productive use. This can be done if the whole audio workstation is developed and built as an integrated combination of software **and** hardware components, that fit together and are Linux compatible down to the last screw. The complexity of the software environment, that comes with the way, this software is developed, cannot and should not be simplified to achieve better usability. But it can be made much more accessible by explaining how it ticks in user oriented documentation and tutorials.

Some may ask:

> is this still possible without charging the users for software?

I say: It is! - if we find a way to charge the users for integration and support... it is my belief that the time is nigh to make people invest in free audio software development. People like Fons Adriaensen, Rob C. Buse or Paul Nasca should be rewarded for their work. A reasonable part of the revenue that could be generated with floss-based audio workstations should go directly to the developers of the most important software for these boxes - especially if these developers do not yet receive any reasonable financial support.

The prototype of the L.A.W. is able to prove, that musicians can get not only the same, but very new and exciting results when using Linux for production. So we also intend, to find musicians that are willing to invest in free software development by purchasing a preconfigured hardware system such as the L.A.W. We commit 30 Percent of the profit that is made with each box sold to be donated as a contribution to linux audio developers. Though this can only be a symbolic act for we lack the resources to gain a big market-share, we consider it a start to better funding of Linux audio development.

2 Gathering functionality the UNIX-Way

Most users coming from proprietary tools, such as Steinberg Nuendo or Magix Samplitude, have a rather ambivalent impression of the system when looking at Linux solutions. On the one hand they like tools such as Jamin or SND, but on the other hand they are disappointed by the lack of features that they have commonly come to expect in proprietary applications. The concept that a complex system can be built out of

many little tools that provide each other with functions is not very common amongst users of proprietary systems.
Testing demos of commercial suites on MS-Windows and comparing this way of working with the work in a Linux audio environment I came to the conclusion, that the sometimes uncomfortable diversity and modularity of Linux audio is indeed its strength - as an artist I want to do new and unusual things, I want to experiment, I want to go beyond limits, I want to be free. Jackd provides such freedom by allowing to combine virtually anything one can do with audio on computers. As the tools derive from the free software culture, they are built to cooperate - there are no reasons to avoid integration with tools from others and there are no reasons to provide many features that are provided by others already in a single software suite.

Some may argue that applications like Rosegarden and Muse try to be just that: one-stop applications like Cubase. But to honour reality: has anyone seen significant progress for the wave tracks of these applications in the past 2-3 years? And still those applications have seen significant development indeed - in their special domains as sequencers. Truth is: those applications do not need powerful wave editing features because their users can edit wavefiles to be merely played in them with a plethora of other programs and/or sync with Ardour. The recent Muse and Rosegarden allow calling an external audio editor to edit wave segments and can reintegrate the results - that's the UNIX-way and it is implemented perfectly.

So the L.A.W. does not make the futile attempt to mimic proprietary audio suites but embraces the diversity of free software and tries to give the user a hand to ease the use of the Linux audio tools. This primarily means 2 things:

1. Integrating all the little, cute and powerful free tools to make them work properly together, without limiting their flexibility
2. Providing readable, usage oriented documentation.

3 Not much else matters - reliability or death

If one asks a musician/producer, what could be the most important feature of an audio workstation, there may be different results. But if you ask, if occasional crashes would be an acceptable price for many extra-features they will consider you crazy. If you work for (paid) hours with a band there will always be a good explanation why a certain feature is missing - if the box freezes and the work is destroyed, not even the most creative producer will find a reason to have this accepted. So to make a producer feel comfortable with a workstation it must be rock-solid. There are 3 prerequisites for this type of comfort:

1. Every part of the suite works with the given drivers.
2. The parts - soft and hard, do not conflict with each other
3. The documentation points the user to bugs and pitfalls that still exist in the software and shows ways to work around them.

Point one is a matter of testing, experience and understanding of predictable user behavior (as far as there is something like predictable user behavior...). Point one and two can be achieved with free components under Linux, but it takes a lot of effort to set it up and there is still no integrated user interface that allows truly intuitive work with the full power of available possibilities without unwanted surprises. The ingredients for a workstation that is both powerful and stable are available for Linux and PC hardware properly supported by Linux is available too. So the first steps to build a Linux Audio Workstation would be:

1. To design a hardware setup for a reasonable price, that fits perfectly together and is completely Linux proof.
2. To develop a set of scripts and consistent templates and presets that allow the user to switch on the full force of a jack-environment with a single click without breaking something.
3. To write complete and up to date documentation with the user in mind.

If Linux software is simply installed as a binary package and automatically set up, then Linux will not be as stable and consistent as is required. Many developers work and build on Fedora or Debian and most of them massively alter/tune their systems - so that users who try to run preconfigured applications on out-of-the-box systems from Novell, Canonical or Mandriva will be confronted with the unpredictable

effects of diversity. No installation program can predict everything and certainly no installer is capable to set an audio card to a single IRQ. So a reliable, full force system for audio can only be set up with direct human intervention. Point three of the prerequsites - the documentation of pitfalls - is a matter of testing, experience and understanding of predictable user behavior (as far as there is something like predictable user behavior...). The goal is, to provide the flexibility of the multitude of combinations, that can be used with Linux audio software and to make the unevitable learning as easy and non-frustrating as possible.

Since we cannot rely on consistency - we need to *use* the diversity and freedom of free software to prove that free development can lead to the same and even better functionality as known from commercial vendors. Whereas the basic system can safely be installed automatically, the important applications (Jackit, Ardour, Muse and some 5-6 more) are compiled on the box with sane ./configure-flags and the whole system setup is examined, adapted and thoroughly tested by humans.
We did so on the prototype with very good results. Still not everything is as perfect as it should be on a workstation ready for unlimited commercial use. Especially kernel hacking is still an issue, because we want a box that is not only a good audio machine but also usable as a desktop computer. As we use Ubuntu on the machine, we also suffer from Canonical's decision to drop realtime support and other audio related settings in Ubuntu's default kernels due to problems reported with laptops. Using a selfmade kernel or a third party package works well but breaks compatibility with certain drivers and programs like VM-Ware. There is an announcement from Canonical to provide an unsupported rt-kernel package with the needed driver support for the upcoming release called feisty fawn. If this will not solve the situation, we need to find another free distribution like 64Studio or Jacklab or help pushing the development of the Ubuntustudio-project, that aims for a multimedia metadistro for Ubuntu.
We have a tutorial in the wiki of http://www.audio4linux.de, that describes every step to set up the stuff on Ubuntu dapper drake. It provides information on adapting it for the recent Ubuntu edgy eft and links to download the needed files. The L.A.W.-documentation is online and a chapter that describes the hardwaresetup is on its way.

3.1 What is being done and how we are different

The user has the opportunity of choosing between 3-4 preconfigured setups:

- Audio workstation (preset configuration with factory support, guaranteed functionality, limited configurability)
- Audio workstation experimental (same preset configuration as above, but with full configurability, limited support and no guarantee - this would also be available for download)
- PC Workstation (Standard home computer with all the power of a Linux desktop system, Internet, office, graphics/layout etc.)
- Rescue system (with direct access to scripts, that reset configurations and replay backups)

The desktop and standard audio system uses XFCE4 as desktop, the experimental audiosystem and the rescue will use Fluxbox. All have the same menu, except the rescuesystem of course (there will be a printed documentation on how to use the rescuesystem from commandline in case X is broken, a Live-CD to rescue the system is planned for later). However, all the scripts, templates and presets can be used in any Desktop environment - templates and presets rely on fitting versions of the respective audio applications and the scripts only require bash. KDE-base must be installed also, since the wrapper scripts utilise kdialog to display messages and konqueror is used to show HTML help pages.

There are both integrated hardware solutions with Linux and CD and/or metadistros available for installation out there. The hardware systems are designed and priced for semi-professionals and we don't know of any Linux-based audio solution that is also a decent desktop PC. Our aim is to provide a box that can be built for about 700,- EUR and that can also serve as a desktop/internet computer.
The installable Distros such as Demudi, Jacklab or CCRMA all have one thing in common: they work with binaries, they do little to integrate the apps and they leave the choice and setup of the hardware to the user. All these people still do great work and it is definitely not our

intention to replace any of them, but rather to collaborate with them.
We are not trying to build our own little world, but wish to incorporate things that are already there and glue them together in a useful way. As mentioned before, we primarily deliver simple wrapper scripts, presets, templates and samples. These will work on any distro that has the software installed, which is called by these scripts and can handle these presets etc. On the box that we ship, the same things will run like a charm (no kidding - our concept allows intense testing of the very machine that will be delivered - so unwanted surprises will be seldom...) - if one wants to use the stuff in a similar but different environment, adaptations may be needed.

We follow a paradigm that favours a grassroots style growth of the project. So we have built a standard desktop PC with an Intel PIV 2.8 CPU 1024MB DDR-RAM and a Terratec EWX 24/96 audio card - rather average real world hardware available for less then 700,- . If we can muster the financial resources until then, we will have a second prototype based on AMD 64 with an M-Audio Audiophile card to bring to the LAC.

We have tested the setup by using it to record several multitrack sessions, composing and manipulating music with midi-driven softsynths and by editing and optimising a noisy-tape-to CD - job for the complete "Ring" by Richard Wagner (plus several similar smaller jobs for string quartet and for rehearsal tapes). We also tested the machine with a Midi-Keyboard/controller with very good results. The setup works well and stable and provides everything we need.

The issues regarding compatibility/stability are solved so far (though it would not be wise, to actually *guarantee* full stability for all needed programs under all conditions...)

3.1.1 The additionals

We have built the previously mentioned framework of helping elements, consisting of 4 major components:

- XFCE configuration scripts, wich allow access to all needed features in a logical and comfortable manner
- an extensive manual in HTML
- several scripts and hacks to load complex scenarios and to provide additional help text
- a set of templates for all major applications that also involve the collaboration between them
- about 300 free licensed presets, sounds and patterns

Starting setups of several collaborating applications *could* be trivial - if all involved elements were aware of each other. Today we still face the problem of certain softsynths that can be attached to jackd via their own command line, and others that need to be called up via tools such as *jack connect*. These are not serious obstacles of course, but there is no reason not to address smaller annoyances as well as great todos. The communication between the several processes is particularly critical and often leads to ruin, especially if LADSPA-FX is involved - this should be remarked amongst developers and distributors.

The website http://www.linuxuse.de/snd offers downloads of sample scripts and templates plus some help texts. More help is to be found at http://www.audio4linux.de and we are also working on an extensive user manual that can serve as an introduction to Linux audio, this can be found at: http://gnupc.de/ zettberlin/law/Documentation/ . In addition to the help provided by the programmers, there are also 3 levels of extra help for the users:

- kdialog popups explaining things that happen, ask the user for needed interaction and point to more detailed help.
- HTML help files for *every* application that explain the basics and how the whole system works. It will be possible to start scripts directly from these files, which will be shown in KDE's konqueror. (Security people may be not that enthusiastic about the help system...)
- an online forum (a simple wiki linked to the forum of audio4linux.de) with daily appearance of at least one developer/help author.

4 Next steps

I would like to present a wish list to the developers and to the user community as well. Developers should improve their documentation and users should start to read it.... . Back in 2003 we had a set of experimental stuff that could

be used but was limited by some crucial weaknesses, especially the lack of stability. Today we have the tools complete and solid enough to start to discover how to deploy them as perfectly as possible. To do this, there must be a vivid community of more than just about 100 users. We do our best to spread word about linux audio and we find new linux audio users every day.

We have combined Ardour, Muse, Hydrogen, AMS, ZynaddSubFX and about 20 tools such as LADSPA, qalsatools etc. into a setup that is powerful and usable for us as experienced Linux users, and we now work intensively on making the setup powerful and usable for everyone that wants to deal with music and computers. We monitor the development of the Linux Audio Session Handler (LASH) constantly and try to encourage developers searching for new goals to join its development. LASH could do the tricks, we now perform to integrate applications much better then the quite clumsy scripts we use now. So we will switch to LASH as soon as it gets ready.

We now begin to give users the opportunity to test the complete Linux Audio Workstation, to give sound tech people and musicians a chance to find out that Linux audio is ready to run for everybody.

5 Conclusions

We believe that free audio software can be an important, powerful way to make Linux visible to the public and thus to make the very concept of collaborative, open and free production of software a success. We not only believe that Linux audio is somewhat usable now and could have a niche - we believe that within 2-3 years it can develop into a superior solution for creative sound people in the same manner as Linux has become a success in motion picture animation/CGI - production.
This can be done if it becomes easier to use, more logically integrated, more stable and more consistent **without** hiding anything of its great opportunities from the user.

At LAC I would like to present our approach to making Linux audio usable for everyone to developers and users as well and to demonstrate the prototype of the Linux Audio Workstation in a workshop and/or as a presentation.

6 Acknowledgements

Our thanks go to ...Paul Davis, Werner Schweer, Paul Nasca, Chris Cannam and to all the other great Linux audio developers out there, to the people at ZKM and Linuxtag that allowed us to present our project at Karlsruhe and Wiesbaden in 2006, and to the people at www.audio4linux.de (especially linuxchaos/Olaf Giesbrecht), jacklab.de (Metasymbol/Michael Bohle, www.mde.djura.org (Thac and Ze), ccrma.stanford.edu/planetccrma and all the other websites, mailing lists and BB's where Linux audio springs to life...

Music composition through Spectral Modeling Synthesis and Pure Data

Edgar Barroso
PHONOS Foundation
P. Circunval.lació 8 (UPF-Estacío França)
Barcelona, Spain, 08003
ebarroso@iua.upf.edu

Alfonso Pérez
MTG – Pompeu Fabra University
P. Circunval.lació 8 (UPF-Estacío França)
Barcelona, Spain, 08003
aperez@iua.upf.edu

Abstract

A major problem of the composition process involving music technology is the selection of the tools within the broader context of the ever-changing language of multimedia production.

The focus of this contribution will be on creative applications of the Spectral Modeling Synthesis (SMS) and Pure Data (PD) in two specific works: Searching your Synesthesia (2005) for flute, clarinet, cello, piano and live electronics and ODD (2006) an electro-acoustic multi channel 5.1 Surround piece.

This paper will provide a brief view on how this open source software offers the user the possibility to experiment with a vast number of compositional techniques in very flexible programming environments.

This view is based on the analysis of the research that has been carried out within the compositional course of action concerning several pieces using these systems. It is intended to share a particular framework for enabling interactions between specific tools for the creative compositional process.

Keywords
Composition, Pure Data, Audio tools, Interaction, Spectral Synthesis.

1 Introduction

During the last recent years there has been a huge proliferation of music software designed to build interactive music systems and audio signal processing. The rapid development in personal computers and the increasing number of audio application users, have situated composers into endless aesthetics, and ethical crossroads, that could result overwhelming and perhaps discouraging. This is particularly true for those initiating composers in computer-generated music composition. Even though experimentation is a fundamental part of almost every composition process, time spent familiarizing with endless different softwares could be enormous, and will take away precious time to deal with the core of the creative process.

In this contribution we present two compositions that deal with interactive systems and spectral modeling synthesis and make use of open source software, in order to show specific practical examples of the usage of those technologies. The aim is to facilitate and quicken the user decision on whether or not this tools have meaningful applications for their own compositional necessities. In the first composition "Searching your Synesthesia" the composer makes use of the graphical programming language Pure Data (PD) developed by Miller Puckette in the 1990s for the creation of interactive computer music and multimedia works. PD is an open source project and has a large developer base working on new extensions to the program. In the second composition, ODD, is inspired in SMS developed by Xavier Serra [1] which is a set of techniques and software implementations for the analysis, transformation and synthesis of musical sounds. These techniques can be used for synthesis, processing and coding applications, while some of the intermediate results might also be applied to other music related problems, such as sound source separation, musical acoustics, music perception, or performance analysis. Two open source implementations of SMS are cited, for Octave and CLAM frameworks.

2 Searching your Synesthesia – Musical Control and Live Performance

This is a piece for flute, clarinet, cello, piano and live electronics, first performed in the Paine Hall of the Harvard University Department of Music on March of 2006. Composer selected this piece to explain a succinct overview of the software he programmed in Pure Data, to show aspects of its flexibility and some of the features

and possibilities that can be achieved with this programming environment. The only intention of this paper is to share a specific thinking framework within a complete performed work, any aesthetic or technical statement must be interpreted only as a subjective opinions of the authors.

2.1 The Patch

All the programming for the performing of the piece was made in Pure Data version 0.40-2 downloaded from the Miller Puckette [2] web page. It uses only two external objects, the first named *counter* and the second one is the *freeverb* object written by Olaf Matthes [3].

The patch contains the following features:

- Recording Machine: allows to record and store in tables and/or hard disk up to 128 audio files. There is a possibility to do it manually or automatically. It also has the capability to load and send this files to any of the sample triggers modules located in the interface.
- Multi Triggering Sampler: Permits to trigger up to eight different simultaneous audio files, each one of them has an independent pitch shift, amplitude, volume, panning, and duration parameters.
- Loop Sampler Triggering: triggers in loop up to 16 voices of the same file, each one of them will have individual controls for the panning, pitch shifting, volume, duration of the loop, starting reading point, and number of voices.
- Microphones / Mixer: Controls all the microphones input and output gains.
- Audio Effects: Resonator-Multi tap Delays and three different reverbs.
- Score Follower: Order the sequence of the events in time during the piece, it uses a MIDI pedal switch to advance to the next musical event, all the rest of the parameters are automated.
- Scratcher: Records in real time and store the recordings in tables, that can be later retrieved in different positions, repetitions and speed .
- Pitch Follower Device: It works with filtered oscillators that can follow or make aleatory counterpoint and harmonization of the incoming signal.
- Amplitude contour device: It uses an amplitude threshold to decide a series of conditions and constrains.
- MIDI: A module that sends midi notes to a midi synthesizer or to a controller. You can activate or de-activate a continuous stream of midi notes by detecting the input gain of the microphone.
- Quadraphonic & Stereo diffusion system: can diffuse every module and sampler trigger individually up to four individual speakers.

2.1.1 The Interface

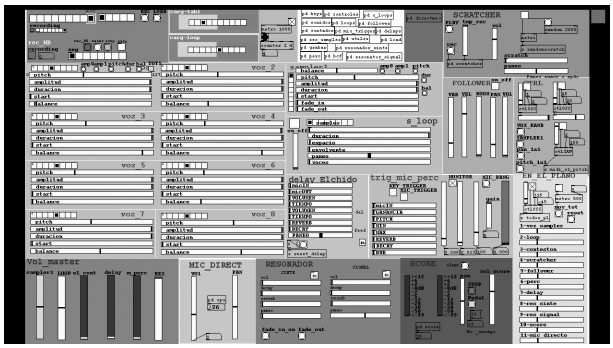


Figure 1: Searching your Synesthesia Pure Data Interface

In this particular work, the attainment of the interface was much more oriented to be an aesthetic platform defining musical structures and a practical controller of the sound engine, in opposition to the idea of conceiving the interface as an instrument itself as in [4]. It was built with a series of modules that work independently and provide the performer the necessary data to control the complete piece only with the use of a MIDI switch pedal (which is controlled by the conductor) which sends commands to the score follower (see figure 1).
One of the goals in the designing process of this interface was the riddance of any direct human control, (except for the MIDI pedal) therefore, the complete piece is built with automated parameters. All the rest of the real time processes are carried out by the computer itself in all senses. One particular challenge of the research was to avoid the overly mechanical sounding, typical of computers that produce every time the exact same parametric control quantities. In the presented patch the computer controllers were implemented trying to simulate (at some level) the human error with the use of random parameters, and inconstant sliders movements to achieve gestural

expressiveness and versatility in the outcome audio results.
Another important aspect that the composer tried to achieve in this piece, was the experimentation with dynamically changing sound processing, were one or more morphological characteristics of the input sound are analyzed and immediately used to control one or more aspects of treatment [5]. The fact that sound itself controls its own treatment was implemented in the programming, although only at specific rather short parts of the sore.

2.1.2 The Score Follower

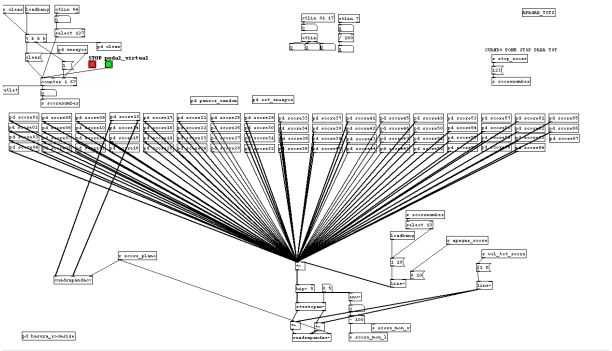


Figure 2: Searching your Synesthesia Pure Data Score Follower

The core of the program is the score follower. It is a work-in-progress device which allows you to divide the score with several cues, and create a single patch for each one of them. The advantage of this system is the economization of processing resources of the computer. The composer created sub-patches (see figure 2), each one having a totally independent self-contained patch that communicates with the interface. These events are always following the triggers coming form the MIDI pedal, using a counter to define the position in time during the performance. Inside this boxes all audio output parameters are controlled. This method was develop since it was much more easy to follow short events rather than long periods of time in which the liability of the analysis could get inaccurate, specially due to the relationship among the measured loudness and other external conditions. Even though the system is divided in this boxes, it allows you to make transitions between particular events in different ways, they can be done in a sudden movement, smoothly or even mixing two or three events simultaneously. For the score following the composer used 4 features that can be recognized: 1) MIDI switch pedal controlled by the conductor or the computer performer. 2) Tone events (notes of the score), 3) Phrase events, 4) Amplitude contour.

2.1.3 Synchronizing vs Chaotic Interaction

In the early stages of the creative process of this piece, the aim was to take advantage of the flexibility of the software and try to combine these tools in order to include the following ideas.

- Elaborate a system in which the interaction could be as clear as possible, however preventing from being too obvious (action - reaction).

- Elaborate a system that generates some indeterminacy and is able to control computer composition that would help performers to add tension and coherence to the musical ideas, yet only for specific parts of the piece.

Some of the strategies that the composer followed to achieve these two goals was to simulate the interaction of the classical period chamber music, where the interaction between the musicians is constant and very clear, however, the idea of variation is constantly present in the music. The composer particularly studied the string quartets from L. V. Beethoven Op. 59 and 132 and tried to apply the same way of thinking about interactivity as he did with the electronics and the instrumentalists.

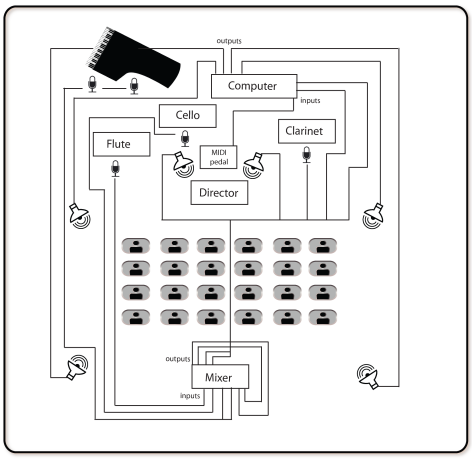


Figure 3: Searching your Synesthesia performing technique diagram.

Another issue concerning the composition of the piece and the use of the synchrony–chaos idea was the diffusion of the total outcoming audio of the ensemble. The first performance took place at the Paine Hall of the Harvard University, and was diffused with the loudspeaker orchestra *Hydra* [6] which has 24 speakers. For this performance,

special arrangements were made to the patch in order to get advantage of the number of speakers. Withal, it was necessary that an additional performer controlled the diffusion system in real time. In normal conditions, the piece uses a four channel configuration (see figure 3). As previously mentioned, every individual sample trigger and module of the interface can have two panning options: left - right and front - back. The aim of using this system is that it allows to make transitions very rapidly and effectively, from totally chaotic individual panning to very subtle panning remarks. Again the same principal: chaos and synchronization in the same piece.

2.1.4 Indeterminacy and Pure Data flexibility.

The possibilities to create an open work with the use of pure data are endless. The composer also includes in the design of the software a few indeterminacy elements. He lefts some composition and structure decisions to the computer using the parameters of sound itself, and some probability results from the analysis of pitch and amplitude that the program re-uses, for instance, to decide when to start or finish producing sound or determine the pitch and location of a prerecorded samples among others. Even though these considerations, there are in fact (few) specific moments of this indeterminacy, the composer considered the compositional structures to be linear, and with sections of music ordered sequentially as in a traditional score. From the composer point of view the piece leaves no room for improvisation coming from the musicians, it was a better idea to leave all indeterminate actions only to the computer. As the composer is not a programmer, the capacity to adapt the system to a new environment and resilience in recovering from a crash or from a human error (MIDI switch pedal) was also a very important and not at all an easy task. Nevertheless Pure Data proved to be a robust, liable environment in which the flexibility can be determined by your imagination.

3 ODD – Composition with Spectral Modelling Synthesis

ODD was conceived based on the SMS technology. SMS is a set of techniques and software implementations for the analysis, transformation and synthesis of musical sounds. The aim of this work is to get general and musically meaningful sound representations (specifically for harmonic sounds) based on analysis, from which musical parameters might be manipulated while maintaining high quality sound. There are two main open source implementations of SMS, one is based on Octave (code can be found at [8]), and the other one on CLAM [7].
As is shown in ODD, this technique can also be used as a "**creative"** tool. (Although there is no single, authoritative perspective or definition of creativity. Most experts agree that is a mental process involving the generation of new ideas or concepts, or new **associations** between existing ideas or concepts.) The composer makes use of the SMS technology dealing with sound morphing, transformations, separation of residual-noise components of sound objects and also is inspired in the graphical representation of analysis data that provokes and stimulates musical concept associations as if it was a score.

3.1 SMS Overview

During analysis, sound is divided into temporal frames that are spectrally analyzed. At each frame the signal is separated into harmonic content and residual or noise. The harmonics of a frame are matched with the ones in the next frame, giving rise to tracks (usually called sinusoidal tracks) that indicate the temporal evolution of each harmonic. This way we can represent any sound as a parametric harmonic part plus a noise residual, and we can get back to the original waveform representation by the resynthesis process.

The parametric representation of the harmonic part of the sound is very useful in order to make sound transformations. If any parameter is modified, the re synthesized sound will be different. SMS lets the composer to experiment with the analysis parameters can produce very interesting and unexpected results.

3.2 SMS Transformations

Transformations are another important process that allows you to have a way of making "variations" of the same sound, from very subtle changes to totally unrecognizable sound source.

Typical transformations include pitch shift, time stretch, timber manipulations, etc. and a special type of transformations called morphing that consists of a cross-synthesis between two sounds, resulting in a new one that has hybrid characteristics. In this process it is quite interesting to provoke ambiguity of identity of the sound source, by approaching sounds in their inner spectral content. Among the endless possibilities of transformation we can emphasize morphing

between instruments and voice, or between "natural" sounds and synthetic ones, the so called "hybrid sounds".

In am more semantic or conceptual context, transformation are defined as the genetic alteration of a cell sound resulting from the introduction of foreign DNA. In music, transformation refers to any operation or process that a composer or performer may apply to a musical variable. The transformation concept of the SMS tools reminded me the so called "Variations" which is a formal musical technique where material is altered during repetition; reiteration with changes. The internal structure of "ODD " is constructed thinking in variations of all sounds, to keep the piece in constant movement during time.
Also the concept of Morphing lead us to the concept of curiosity, which by definition is the unknown result of combining two more more different elements or in this case sounds. It is a very useful tool to generate complex unique sounds. Many experts agree that curiosity is any natural inquisitive behavior. and is the emotional aspect of living beings that engenders exploration, investigation and learning.

3.2 The residual component

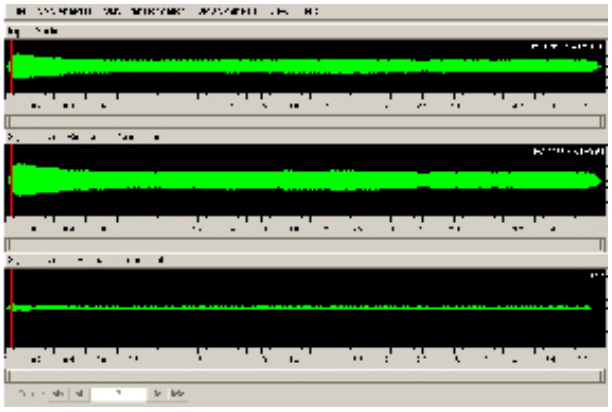


Figure 4. Residual Graphic Representation

Apart from the transformations above mentioned, the possibility of isolating the residual component of the sound is fascinating for creative aims. As an example, we can extract the noise part of an instrumental sound or the non-pitched element of a vocal sample. In figure 4 we can see at the top the original sound , in the middle the resynthesized sound and the bottom the residual component. The composer used the concept of residual , understanding the residual as the quantity left over at the end of a process; a remainder. The composer found that recycling this sounds, in other words to put or pass through a cycle again, as for further treatment would be a interesting idea. This thought became the core of the composition, the subjective analogy of extracting useful materials from otherwise unwanted frequencies, and make a complete composition structure with it..

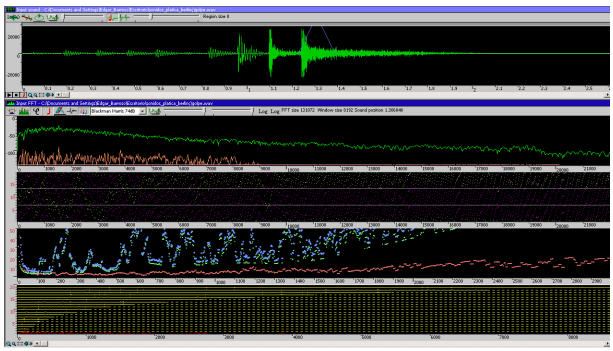


Figure 5. Graphical Representation of Analysis parameters.

3.3 Graphical Representation of Analysis parameters as a score generator

3.0.1 Evolution in time of SMS analysis data can be easily visualized. Among this data we can mention the FFT, sinusoidal tracks, residual part, pitch, sine amplitudes, etc. (see Figure 5)

Non-traditional music notation has always looked for inspirational ideas on graphic structures. The graphical representation of the analysis parameters can be seen as such.

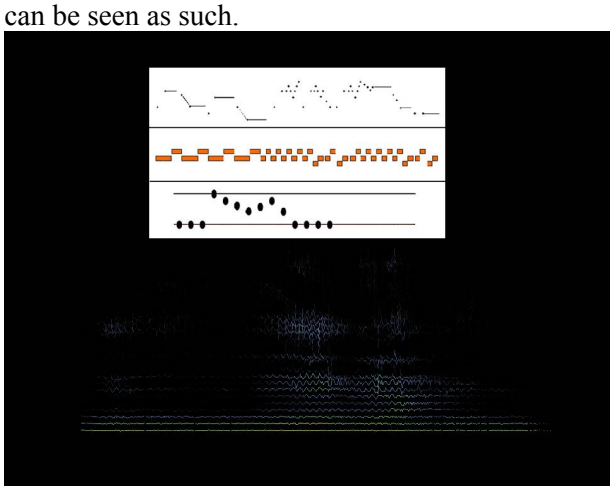


Figure 6. Score generator graphic

Graphical representation of some parameters resulting from the analysis of sounds created for the piece, is used as feed-back by the composer to determine the upcoming musical ideas (see figure 6). The word "fractal" in colloquial usage, denotes a shape that is recursively constructed or self-similar, that is, a shape that appears similar at all

scales of magnification and is therefore often referred to as infinitely complex.

4 Conclusion

We had introduced the creative applications of the Spectral Modeling Synthesis (SMS) and Pure Data (PD) in and its use for music composition. We have shown that the open source software SMS and PD could be very well suited to the design and development of music compositions. Future work will involve increasing the flexibility of the interactive system and extending the experimentation with SMS to obtain new processed sounds.

5 Acknowledgements

The authors would like to thank to Hans Tutschku, Matteo Sistisette, Ricard Marxer, Berio Molina and Günter Geiger for the advise in the programming and musical aspects of the pieces. We also thank Folkmar Hein, Stefan Kersten, Miguel Álvarez-Fernández and the Inventionen Festival, the MTG and the PHONOS Foundation for providing the infrastructure for developing these two pieces.

References

[1] Xavier Serra. "Musical Sound Modelling With Sinusoids Plus Noise". Published in C. Roads, S. Pope, A Picialli, G. De Poli, editors. 1997. "Musical Signal processing", Swets&Zeitlnger Pubishers.

[2] Miller Puckette. http://crca.ucsd.edu/~msp/software.html 2006.

[3] Schroeder/Moorer reverb model implementation, version 0.2c written by Olaf Matthes, based on Freeverb by Jezar. <olaf.matthes@gmx.de>

[4] Thor Magnusson. "ixi software: The Interface as Instrument". Proceedings of the 2005 International Conference on New Interfaces for Musical Expression (NIME05), Vancouver, BC, Canada.

[5] Hans Tutschku. "Portfolio of Compositions" Department of Music School of Humanities. The University of Birmingham March 2003

[6] Harvard University Studio for Electroacoustic Composition. http://huseac.fas.harvard.edu/pages/05hydra/1hydra.html 2007

[7] CLAM Framework. http://clam.iua.upf.edu/

[8] Octave code for SMS Framework. http://www.iua.upf.es/mtg/sms/

Qtractor – An Audio/MIDI multi-track sequencer

Rui Nuno CAPELA
rncbc.org

rncbc@rncbc.org

Abstract

Qtractor is an Audio/ MIDI multi-track sequencer application written in C++ around the Qt4 toolkit using Qt Designer. The initial target platform will be Linux, where the Jack Audio Connection Kit (JACK) for audio, and the Advanced Linux Sound Architecture (ALSA) for MIDI, are the main infrastructures to evolve as a fairly-featured Linux Desktop Audio Workstation GUI, specially dedicated to the personal home-studio.

Keywords

Linux, audio, MIDI, JACK, ALSA, multi-track, recording, sequencer

1 Introduction

As a new Linux Audio offering, Qtractor targets and positions itself comfortably tagged as for the techno-boy bedroom home-studio. However, in general, it is just yet another digital audio and MIDI multi-track composition and arranger software application. The design and functionality model takes as fundamental the now usual multi-track composing techniques for modern music-making. It aims to be intuitive and easy to use. Unfortunately, for the classic and erudite musician, there are and will be no plans to integrate any kind of score editor.

Qtractor is tentatively to be a part of the Linux home-studio. Even though built on a monolithic application architecture, it is not about to be the whole of it. Besides its name, it should be noted that it is not the same kind nor genre of that once called as *tracker* software. However, it has all the possibilities to be as much the same playground.

Currently the hobby work of one developer, development has started circa 2Q2005, initially as a Qt3 application. Since October 2006 it's officially a Qt4 application. It is still a work in progress project.

It is natively hardwired and exclusive to the JACK audio infrastructure and the ALSA sequencer for MIDI, thus currently being a Linux-only application. There are not known intentions on support to any other platforms.

Qtractor is free open-source software, licensed under the GPL and is welcoming all collaboration and review from the Linux Audio developer and user community in particular and the public in general.

2 Installation

2.1 Requirements

The software requirements for build and run-time are listed as follows:

Mandatory:

- Qt4 (core, gui, xml) [2]
- JACK [3]
- ALSA [4]
- libsndfile [5]

Optional (opted-in at build time):

- libvorbis (enc, file) [6]
- libmad [7]
- libsamplerate [8]

2.2 Download

Qtractor is still in alpha stage of development, but already fully functional. The source code is publicly available from sourceforge.net CVS repository [1].

2.3 Build

The standard procedure for source distributions based on *autoconf* follows, through the usual `./configure && make && make install`, which will end installing the `qtractor` binary executable and desktop icon files.

2.4 Configuration

Qtractor holds its run-time settings and configuration state per user, in a file located as `$HOME/.config/rncbc.org/Qtractor.conf`. Normally, there is no need to edit this file, as it is recreated and rewritten every time `qtractor` is run.

3 GUI

Qtractor user interface design is thought as mainstream and standard as it could be on modern computer desktop environments. The presentation layer is supposed to be easy for the average user to interact and to discover the full potential and functionality of the inner core application layer (i.e. Audio and MIDI engines, in a nutshell).

Figure 1 shows an overall aspect of the GUI, with example session loaded into the workspace.

The main presentation window includes the usual *menu*, *toolbars* with common action icon-buttons, and the main application workspace where one finds the *track list* on the left and the *track time* scale ruler at the top of the main arranger *track view*, which is where main action takes place and tracks and clips are graphically displayed.

Core application functionality is complemented with a couple of top-level floating windows: the *mixer* window, resembling a mixing console which assists in the control and monitoring operations, and the *connections* window, featuring integrated inter-application device connectivity.

Two utility windows are additionally featured: the *messages* window, specially suited for debugging, and the *files* window where audio and MIDI files are organized and selected on demand.

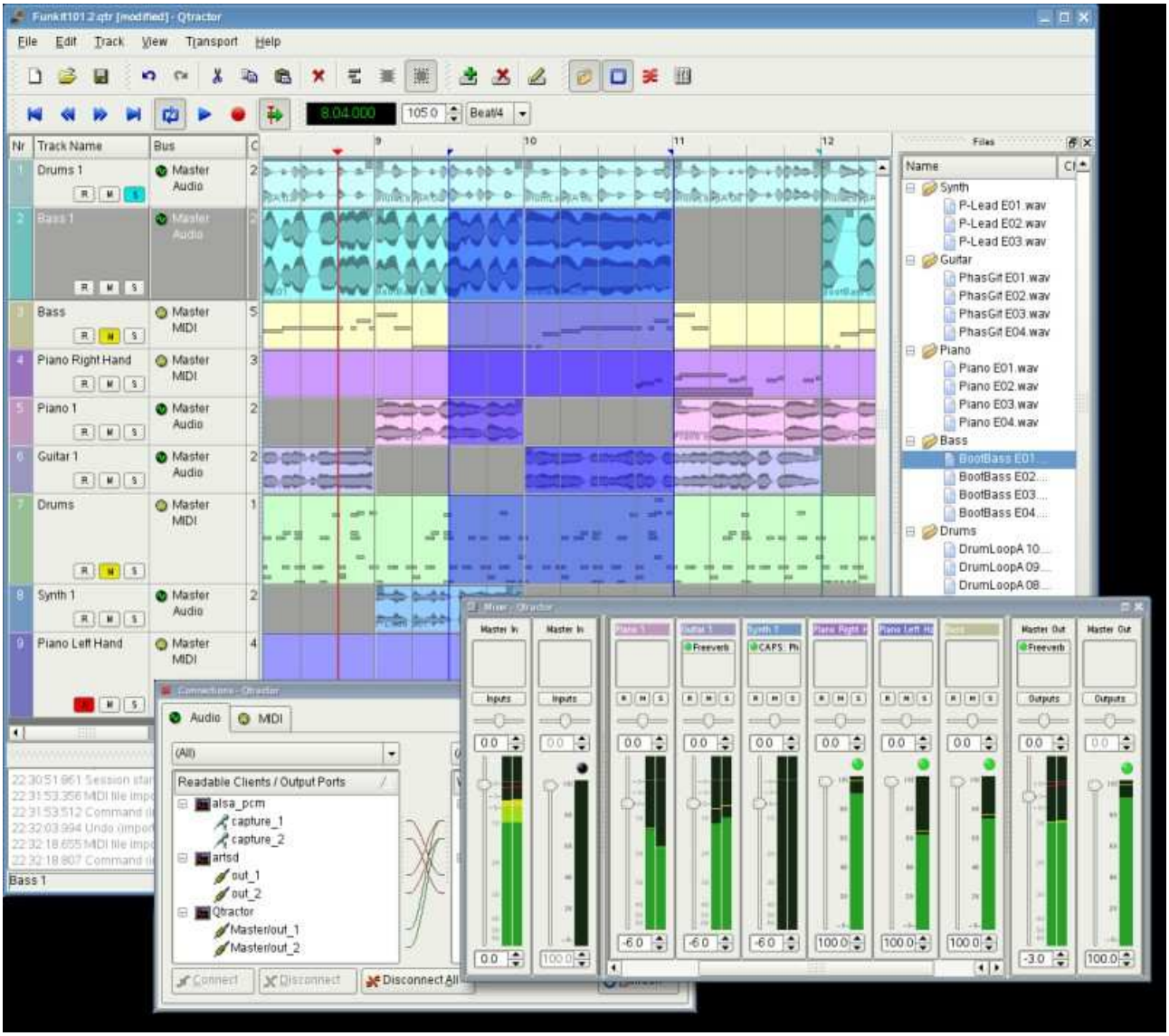


Figure 1: Actual GUI aspect showing the Tracks, Mixer, Connections and Files windows.

Dialog windows for editing *session* and *track* properties are also accessible in their proper context. Finally, session and application configuration options are assisted through respective customizing dialogs: *buses*, *instruments* and *options*.

4 Sessions

Qtractor sessions are defined as the complete and internally descriptive state of an arrangement made with the software. This state description is materialized in document form, as a XML encoded file. A session is therefore the computational description of all properties, variables, references and parameters of all audio and MIDI files and plug-ins that composes the working musical arrangement, put together while working with the software.

Effectively, sessions are formed by one or more tracks, which in turn includes their respective and fundamental elements, the clips. All possible and relevant information is stored in the session file, making it perfectly possible to restore the current working session at a later time. Sessions can thus be created as new, saved and loaded on demand, as found usual on document based GUI applications. Qtractor is a SDI application, only one session can be loaded for work at anyone time.

It is important to note that Qtractor sessions are locked hard to one, and only one, fixed audio signal sample-rate, exactly the one the JACK [3] server is running at the time the session is started. Any attempt to convert disparate sample-rate sessions is subject to a recommendation warning message. However, individual audio clip files are automatically converted on playback in real-time to the host sample-rate (via libsamplerate [8]).

Another restriction worth mentioning is that sessions have constant tempo (BPM) by current design decision, but can be otherwise changed at any time. However, it must be still regarded as a global property of the whole musical arrangement, that meaning there is no support for a tempo map, yet.

5 Files

Sound file selection is made available through a tabbed mini-organizer and convenient widget. Audio and MIDI file lists are kept separated on their respective tabs. Files can be explicitly added and grouped into a hierarchical tree list. Individual files can be explicitly drag-and-dropped from the desktop environment and within the provided tree list. This lists all the files which are referred in the working arrangement session. File items can be drag-and-dropped directly into the track window, thus creating new clips in the working arrangement.

Audio file format support is the same as the one provided by libsndfile (wav, aiff, flac, au, etc. [5]) and, optionally libvorbisfile (ogg [6]) and libmad (mp3 [7]). MIDI file support covers the usual SMF formats 0 and 1, through native, home-brew implementation.

6 Tracks and Clips

Clips are fully described as the elemental items of a session arrangement, being either integral or partial parts of existing audio and MIDI material, stored on external formated files.

As is, Qtractor is technically described as a non-destructive sequencer and arranger. It does not affect, alter or modify in any way, none of the audio or MIDI files that are loaded and represented as clips. Files resulting from capture and recording operations are the notable exceptions. Once created, all recorded files are thereafter loaded as usual. All editing operations are accomplished in exclusive parametric form, having no resemblance whatsoever on the file system.

Audio clips are representations of the whole or part of a single audio file. MIDI clips are representations of a sequence of events of one single MIDI channel, as extracted from a SMF format 0 file or of one single track, as from a SMF format 1 one, either in whole or in part.

Clip properties may include its label (name), start time (location), offset and length (in frames), fade-in and fade-out length (in frames), respectively from the start and end of the clip. Although fade-in and fade-outs are always displayed as straight lines, the actual audio volume (gain) and MIDI velocity effect can be opted to be of either linear, square or cubic characteristic.

Clips are placed on tracks, either by importing integral audio and MIDI files as new tracks or by drag-and-dropping files into the track-view arranger window. After being initially placed on their respective tracks, clips are subject targets to featured clip-region operations: drag-move, copy, cut, paste, delete, etc. Altering clip fade-in and fade-out is also carried out by dragging corresponding visual handles.

Most clip editing operations are accomplished through usual GUI interaction, by first selecting one or multiple clips and/or regions and applying the edit action upon the resulting selection. There are three selection modes available: clip, range and

rectangular modalities. In clip-mode, clips are selected as a whole with no sub-clip regions possible. In range-mode, clip regions are selected on all tracks between a given time interval or range. In rectangular-mode, only the regions that fall under a rectangular area are selected, that meaning for adjacent tracks and clips only.

Tracks may be armed for recording, making way for creating new audio and MIDI clip files with captured material. Tracks can also be muted and soloed on mix-down. Most editing operations should be possible while playback is rolling (but not completely safe though; there are many procedural helpers, but not completely assisted with lock-free primitives, yet).

7 Engines and Buses

Qtractor is a fairly massive multi-threaded application. For instance, each audio clip has a dedicated disk I/O executive thread, which synchronizes with the master engine and, for all purposes, to central JACK real-time audio processing cycle, through a lock-free ring-buffer. These audio file ring-buffers are recycled (filled/emptied) at one second threshold, and has a maximum streaming capacity of 4-5 seconds of audio sample data. Smaller clips are permanently cached in buffer.

Audio thread scheduling is mastered and mandated through the JACK callback API model. MIDI clip events are queued in anticipation through one MIDI output thread, which feeds a ALSA sequencer queue, synchronized on 1-second periods to the JACK process cycle. A single thread is responsible to listening (polling) on MIDI input and multiplexes all incoming events through record-armed MIDI tracks. Timestamping is done through ALSA sequencer facility.

Looping is assisted right through the audio file buffering layer, at the disk I/O thread context. The same consideration is adopted for MIDI output queuing. JACK transport support is not an option, as playback positioning is constantly kept in soft-chase fashion. Audio frame relocation is accounted from successive JACK client process cycles (i.e. buffer-period resolution).

JACK and ALSA sequencer ports are logically aggregated as buses with respect to the audio and MIDI signal routing paths, functioning as the fundamental device interfaces. Input buses, through exposing their respective input ports, are responsible inlets on capture and recording. Output buses are the main signal outlets and are responsible as playback and mix-down devices.

Figure 2 shows a conceptual block diagram that outlines, in very simplistic form, the logical audio and MIDI signal flow.

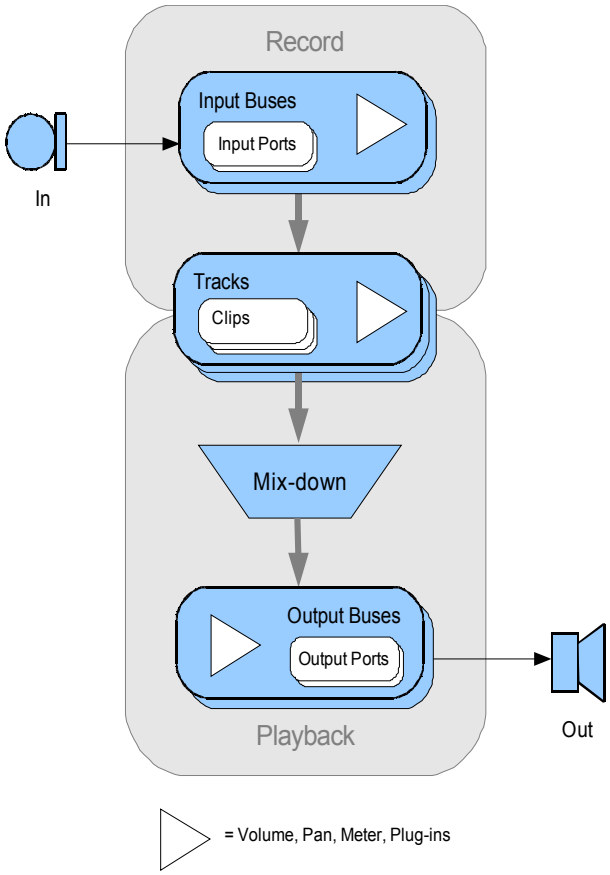


Figure 2: Conceptual Block Diagram

Buses are independently assigned to tracks. Each track is assigned to one input bus for recording, and to one output bus for playback and mix-down. The assigned output bus determines the number of channels the track supports. Clips bounded to disparate multichannel audio files, for which their number of channels do not match with proper bus/track's one, are automatically resolved on mix-down.

By default, “Master” buses are automatically created at session startup, being stereo for audio (2 channels ports, auto-connected) and single port for MIDI (16 addressable channels). Bus ports are accessible for arbitrary connection to and from external client applications or devices, through the connections window interface.

8 Track View

Tracks are arranged as a sequence of one or more overlapping clips of the same file type, either audio or MIDI. The tracks window is the main

application workspace, serving as a virtual canvas of a multi-track composition arranger. Most of the editing operations are made on this track list-view window.

The track list-view window has two panes, the left one displays the list of tracks with their respective properties and the center-right pane is the proper track-view canvas window where main multi-track composition and arranging activity is pictured and performed. Tracks are stacked on horizontal strips and clips are layered on a bi-dimensional grid, in time sequence for each track strip. Time is modeled on the horizontal axis and pictured by a bar-beat scale ruler at the top of the track-view.

Clips may be conveniently aligned to discrete time positions, depending on the current *snap* mode setting. When not set to "None", the snapping is always carried out to MIDI resolution, quantized to ticks per quarter note granularity.

Each track has its own user assignable colors for better visual identification. Audio clips are displayed with approximate waveform graphic, with peak and RMS signal envelopes, as read from the respective audio file segment. MIDI clips are shown as a *piano-roll* like graphic, with note events shown as small rectangles, depicting pitch, time and duration.

All session, track and clip editing operations are undo/redo-able. Discrete view zooming and track vertical resizing operations are also available.

9 Mixer

The mixer window serves for session control, monitoring, recording and assistance in mixdown operations. The mixer is divided in three panes: the left accomodates all input buses, the center with individual track strips and the right for the output buses. Each mixer strip offers a volume and pan control and monitors each one of the respective buses and tracks. Audio strips also offers the possibility to chain plug-in effects (LADSPA [9]).

Monitoring is presented in the form of peak level meters for audio and note event velocity for MIDI, both with fall-off eye-candy. MIDI mixer strips also feature an output event activity LED.

Audio volume is presented on a dBfs scale (IEC 268-10) and pan is applied in approximated equal-power effect (trigonometric weighting). For MIDI tracks, volume control is implemented through respective channel controller-7 and system-exclusive master volume for output buses. MIDI pan control is only available for track strips and is implemented through channel controller-10. MIDI input buses have volume and pan controls disabled.

10 Connections

The connections window serves the establishing of audio and MIDI port connections between the internal core layer input and output buses and the external devices or client applications. Incidentally the connections window can also be used to make connections between external client application ports, either JACK clients for audio or ALSA sequencer clients for MIDI. In fact, it almost completely replicates the very same functionality of QjackCtl [10]. All connections on the existing input and output buses are properly saved and restored upon session recall.

11 Audio Effects Plug-ins

LADSPA [9] plug-in support is available for all audio input and output buses and for all audio tracks. Plug-ins are aggregated seamlessly as one single instance on a multi-channel context and can be individually selected, activated and moved within the plug-in chain order. Individual plug-in control parameters can be modified in real-time through provided dialog windows and maintained as named presets for re-usability.

12 Instruments

As a special feature, Cakewalk [11] instrument definition files (.ins) are natively supported, thus offering a convenient MIDI bank-select/program-change mapping for existing MIDI instrument patch names, and easier, intelligible selection of MIDI track channels.

13 Future Thoughts

As of its current status, there are many and rather fundamental functionality still misssing that tear Qtractor apart from a finished product, let alone for the quest of its own goals. It's still a work in progress. In my own personal agenda priority, the following are the ones for taking care in times to come:

- MIDI clip editor.
- MIDI export.
- Punch-in/out recording.
- Time-stretching; pitch-shifting.
- Audio export.
- Tempo map.

14 Conclusion

As fundamental as is, Qtractor might be just some clone of earlier and existing software, being blatantly one of the Cakewalk's Pro Audio series. It is however more than that, when regarded from the free open-source software development point of view, much like some cauldron framework, user-oriented, programmable, pattern sequencer, eventually targeted as a potential toolbox and workbench for easy, direct, live music-making and experimentalism

15 Acknowledgements

I am grateful to the free software open-source community in general and to the Linux Audio developers and users in particular, who dedicated their valuable time to the development and support of free audio and MIDI software, being Qtractor just one humble manifestation of such class of human endeavor.

Qtractor logo/icon is an original work of Andy Fitzsimon, borrowed from the public domain openclipart.org gallery.

References

[1] Qtractor – An Audio/MIDI multi-track sequencer, http://qtractor.sourceforge.net/

[2] Qt4 - C++ class library and tools for cross-platform development and internationalization. http://www.trolltech.org/products/qt/

[3] JACK Audio Connection Kit, http://jackaudio.org/

[4] ALSA - Advanced Linux Sound Architecture, http://www.alsa-project.org/

[5] libsndfile - C library for reading and writing files containing sampled sound, http://www.mega-nerd.com/libsndfile/

[6] libvorbis - Ogg Vorbis audio compression, http://xiph.org/vorbis/

[7] libmad - High-quality MPEG audio decoder, http://www.underbit.com/products/mad/

[8] libsamplerate – The secret rabbit code, C library for audio sample rate conversion, http://www.mega-nerd.com/SRC/

[9] LADSPA - Linux Audio Developer's Simple Plugin API, http://www.ladspa.org/

[10] QjackCtl – JACK Qt GUI, http://qjackctl.sourceforge.net

[11] Cakewalk - Powerful and easy to use products for music creation and recording, http://www.cakewalk.com

[12] rncbc.org - My personal web presence, blog, forum and other mundane material, http://www.rncbc.org

JJack – Using the JACK Audio Connection Kit with Java

Jens GULDEN
Formal Models, Logic and Programming (FLP),
Technical University Berlin
jgulden@cs.tu-berlin.de

Abstract

The JACK Audio Connection Kit is one of the de-facto standards on Linux operating systems for low-latency audio routing between multiple audio processing applications. As sound processing in an interconnected setup is not originally supported by the Java language itself, it has proven to be useful to provide interoperability between JACK and Java via the JJack bridge API. This allows creating music applications for use in a professional audio environment with the Java programming language.

Keywords

JACK Audio Connection Kit, Java, low-latency, Linux audio

1 Introduction

This article introduces JJack ([1]), a JACK-to-Java bridge API, which consists of a native bridge library providing access to the JACK ([2]) audio processing system, and a Java-side model API which reflects features of the native JACK system into the object-oriented world of Java. With JJack, audio processing in an interconnected virtual studio environment can be achieved on Linux and MacOS with software written in Java.

First, the JACK Audio Connection Kit gets briefly introduced in section 2. Section 3 shows the main principles of creating JACK clients with Java using JJack. An object-oriented model of basic client interconnectivity is provided by the high-level API of JJack as discussed in section 4. Section 5 sketches possible further applications by pointing out the JavaBeans-compatibility of JJack clients. In section 6, sample JJack clients from the JJack distribution archive are presented. Section 7 gives a note on how new features since JDK 1.4 have helped to optimize JJack's performance, and section 8 provides a short comparison between the core features of Java's internal Java Sound API and the JJack bridge API. Finally, a real-world application using JJack, the music sequencer *Frinika*, is presented in section 9, and section 10 gives a short concluding summary.

2 The JACK Audio Connection Kit

JACK ([2]) is a low-latency audio server which provides interconnectivity between different audio applications, such as software-synthesizers, multi-track sequencers, effect-modules, mixers, recording applications etc. While physically being assigned to usually just a single audio device, on the software-side the JACK server provides a virtual studio environment for any number of different software applications, and allows to interconnect them with virtual cables.

A possible JACK system setup, with one JJack-based application acting as a JACK client, is schematically displayed in Fig. 1.

3 Processing Audio with JACK and Java

Each native software application that is capable of acting as a JACK client provides a `process()`-function. This function gets called by the JACK system regularly to pass in audio data as input, and probably receive audio data as output from the `process()`-method.

The very basic idea of how a JACK-client works in Java is the same: a method `process()` is invoked repeatedly, providing audio data as input for processing. This data can then be used by the `process()`-method to be analyzed or transformed in any way, and to finally generate output audio data. Clients can also choose to restrict themselves to only process audio input data without generating any output (e. g. recording the audio input), or to

ignore any audio input and exclusively generate output (e. g. sound-generators, synthesizers etc.).

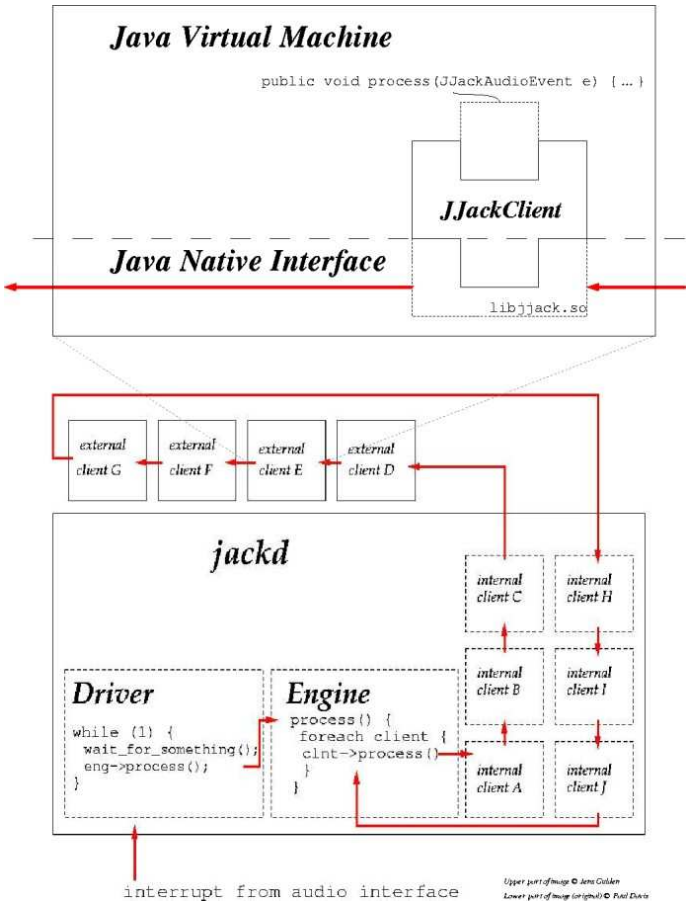


Fig. 1: JACK processing loop and JJack client

Each implementation of the interface `de.gulden.framework.jjack.JJackAudioProcessor` owns a method `process(JJackAudioEvent e)` which is responsible for processing audio data in this basic way. Example 1 gives an idea of how the `process()`-method can be implemented.

```
public void process(JJackAudioEvent e) {
    float v = getVolume(); // parameter from gui
    for (int i=0; i<e.countChannels(); i++) {
        FloatBuffer in = e.getInput(i);
        FloatBuffer out = e.getOutput(i);
        int cap = in.capacity();
        for (int j=0; j<cap; j++) {
            float a = in.get(j);
            a *= v;
            if (a>1.0) {
                a = 1.0f;
            } else if (a<-1.0) {
                a = -1.0f;
            }
            out.put(j, a);
        }
    }
}
```

Example 1: `process()`-method in Java

In the most simple scenario, a Java-written JACK client implements one basic `process()`-method, which will get called by a native bridge library with every callback from the JACK processing thread. The native bridge library appears as one native JACK client to the overall JACK system. (This situation had been depicted in Fig. 1.) Audio input data is provided via `JJackAudioEvent.getInput(channel)` as a Java-accessible `DirectFloatArray`. Audio output data is written to another `DirectFloatArray` retrieved by `JJackAudioEvent.getOutput(channel)`, which in turn will be passed as a memory-addressed native float-array back to the JACK server again.

Initializing JJack to be used in such a single-`process()`-method setup is done as demonstrated in example 2.

```
public class MyJJackClient implements JJackAudioProcessor {

    public static void main(String[] args) {

        // get JACK system's sample rate, initialize
        int sampleRate = JJackSystem.getSampleRate();
        System.out.print("Sample-rate: "+ sampleRate);

        // set single processing client
        MyJJackClient client = new MyJJackClient();
        JJackSystem.setClient(client);

        // ...
    }

    public void process(JJackAudioEvent e) {
        // ... (example 1) ...
    }

}
```

Example 2: using a single `process()`-method

4 JJack high-level API

Besides the 1:1 reflection of a native JACK client's `process()`-method as described in the previous section, JJack also contains a basic object-oriented framework for interconnecting multiple `JJackAudioProcessors` inside one running virtual machine, making them appear as one complex JACK client to the native JACK system. The high-level API is an add-on to JJack's basic functionality. It can be used to model complex audio processors which internally consist of multiple `JJackAudioProcessors` (or derived classes) in a Java-style, object-oriented manner on the Java side.

When using JJack's high-level API, the audio data is internally routed among the connected `JJackAudioProcessors` within one Java virtual machine. The overall combined setup of JJack-

processors appears as one single native JACK client to the overall native JACK system. This situation is shown in Fig. 3.

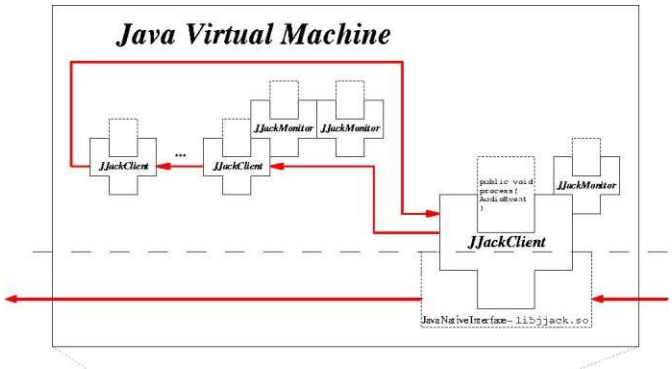


Fig. 3: Using the high-level API with multiple interconnected clients

Some core elements of the high-level API are displayed by the UML class-diagram in Fig. 4.

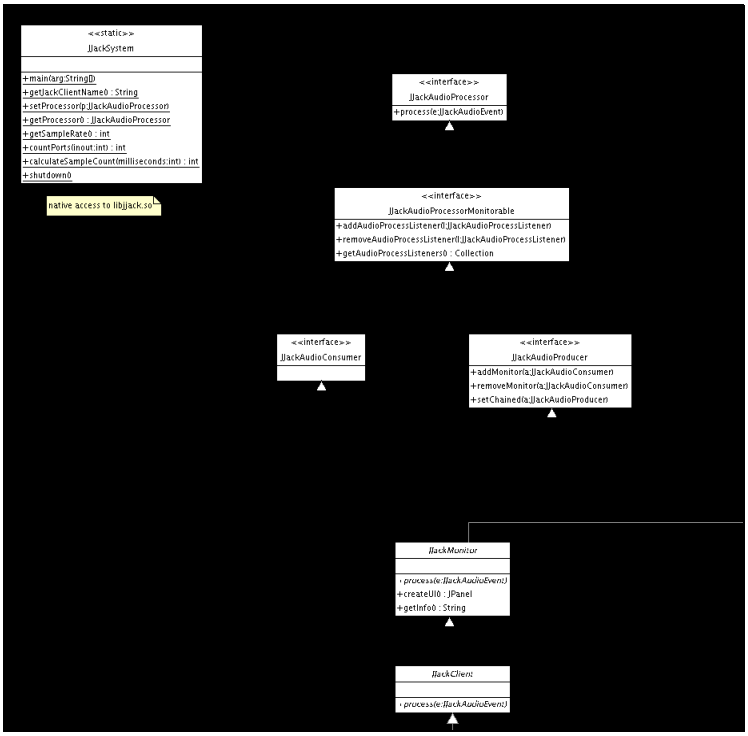


Fig. 4: JJack API UML class-diagram (partial)

5 JavaBeans compatibility

Interconnecting JJack clients is achieved by a JavaBeans-compatible event mechanism. As a consequence, it is possible to configure and interconnect multiple JJack clients inside a JavaBeans development-environment like Sun's *Bean Builder* ([4]). All clients combined inside one Java virtual machine appear as a single native client to the JACK system. Figure 5 demonstrates the use of the *Bean Builder* for creating setups of multiple JJack clients.

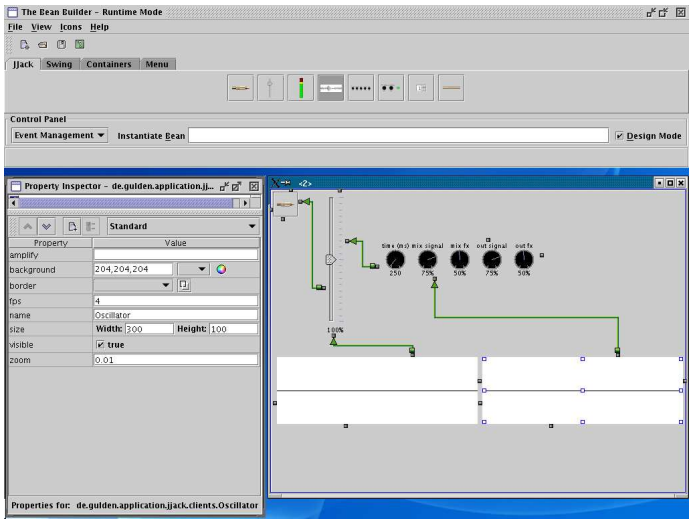


Fig. 5: JJack clients configured as JavaBeans in the BeanBuilder

6 Example clients

Some example JJack clients are included in the distribution archive, together with source-code. They can act as a starting point to learn how to use the JJack API. The examples are

GAIN:
class `de.gulden.application.jjack.clients.Gain`, a simple volume control (linear multiplication of the signal amplitude).

VU:
class `de.gulden.application.jjack.clients.VU`, a monitor client (input only) for displaying the average signal amplitude.

OSCILLATOR:
`de.gulden.application.jjack.clients.Oscillator`, a monitor client (input only) for displaying the signal as a waveform graph.

DELAY:
class `de.gulden.application.jjack.clients.Delay`, adds an echo effect to the audio signal.

GATE:
class `de.gulden.application.jjack.clients.Gate`, a noise gate that suppresses audio signal below a given threshold value. Only if the signal is loud enough, it will be passed through.

CHANNEL:
`de.gulden.application.jjack.clients.Channel`,

selects one channel from a multi-channel input and routes it to the mono output channel #0.

CABLE:
class `de.gulden.application.jjack.clients.Cable`, passes the audio signal through without any change. This is just a null-client for demonstration.

Fig. 6 shows a combination of example clients in operation.

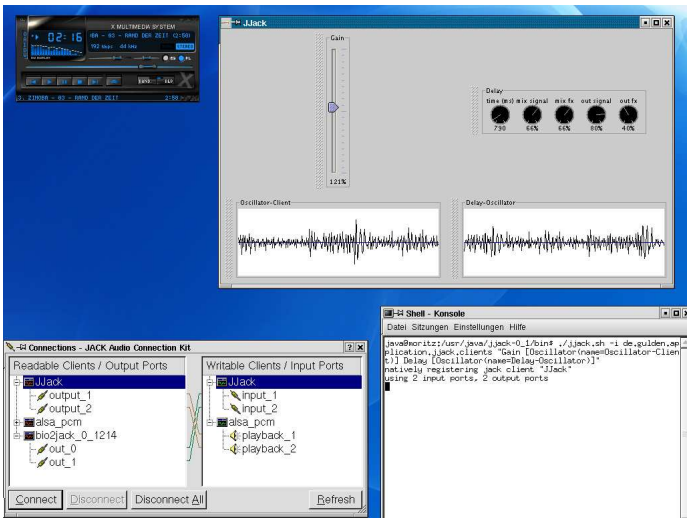


Fig. 6: Screenshot showing a combination of example clients in operation

The source-code implementing the DELAY client might help giving insight in how digital audio processing is done with Java and JJack. The source-code of the `process()`-method is listed in example 3.

7 Used features from JDK 1.4

Bridging between native JACK and Java can be achieved efficiently thanks to new "hardware-near" extensions to the Java standard API. Since JDK version 1.4, Java contains the package `java.nio.*` which allows the mapping of physical memory to Java-accessible arrays of choosable data-types ([4]). By using the interface `java.nio.FloatBuffer` and its implementation `DirectFloatBuffer`, no data-format conversion and even no copying of data is required to interface JACK's processing thread with code running inside the Java virtual machine.

This major advantage of JDK 1.4's features are the reason why JJack requires Java version 1.4 or above, versions up to 1.3 cannot be used with JJack.

```
/**
 * class:  de.gulden.application.jjack.clients.Delay
 *         (implements
 *         de.gulden.framework.jjack.JJackAudioProcessor)
 * method: void process(JJackAudioEvent e)
 *
 * Applies a classical digital delay, seperately
 * for each channel.
 */
public void process(JJackAudioEvent e) {
  int delaytime = getTime();
  float mixSignal = (float)getMixSignal() / 100;
  float mixFx = (float)getMixFx() / 100;
  float outSignal = (float)getOutSignal() / 100;
  float outFx = (float)getOutFx() / 100;
  int sampleRate = getSampleRate();
  int diff = delaytime * sampleRate / 1000 ;
  int channels = e.countChannels(); // number of channels
  if (ring == null) { // first call, init ringbuffers
    ring = new RingFloat[channels];
    for (int i = 0; i < channels; i++) {
      ring[i] = new RingFloat(diff);
    }
  }
  for (int i=0; i < channels; i++) {
    RingFloat r = ring[i];
    r.ensureCapacity(diff);
    FloatBuffer in = e.getInput(i);   // input buffer
    FloatBuffer out = e.getOutput(i); // output buffer
    int cap = in.capacity(); // number of samples available
    for (int j=0; j<cap; j++) {
      float signal = in.get(j); // read input signal
      float fx = r.get(diff);
      float mix = signal * mixSignal + fx * mixFx;
      float ou = signal * outSignal + fx * outFx;
      r.put(mix); // remember for delay
      out.put(j, ou); // write output signal
    }
  }
}
```

Example 3: `process()`-method of the Delay example client

8 Comparison between JJack and the Java Sound API

Since JDK version 1.3, the Java Sound API ([5]) is part of the Java language standard. As JJack and the Java Sound API differ in several aspects, it is important to be aware of the differences when deciding which audio processing approach is best suited for a software project's needs.

Table 1 gives an explanatory overview on the most relevant differences between JJack and the Java Sound API.

	JJack	JavaSound
interoper-ability	high, virtual cable-connectivity	low, device I/O oriented
timing latency[1]	JACK timing, potential low-latency below 5 ms on fast machines	device-dependent, varying among OSs and JDK versions (up to 200 ms latency JDK1.4/Linux, below 5 ms with JDK1.5 and higher)
realtime-capability	yes (application can run in a user-thread, JACK in a realtime thread)	no (unless the whole JVM runs in a realtime thread)
process architecture	pull-architecture (the JACK thread calls the `process()`-method)	push-architecture (application is responsible for delivering data on time)
internal data -format	32-bit float (less aliasing and higher performance expected, less conversions between interconnected applications)	16-bit integer (faster processing for simple clients, additional floating-point conversion required for complex tasks)
number of channels	any	according to hardware driver or software mixer
operating system	Linux or Mac (JACK required)	Any (JDK >= 1.3)

Table 1: Comparison JJack – Java Sound API

[1]Latency tests have been performed with a two-computers setup, using the benchmark application `de.gulden.framework.jjack.util.benchmark.AudioBenchmark` (available via JJack's concurrent versions system, CVS).

9 Real-world application

The music sequencer application *Frinika* ([7]) is entirely written in Java and serves as a real-world example for the use of JJack. On Linux systems, *Frinika* detects the available audio system, and if JACK can be accessed successfully, the application runs on top of JJack to handle its audio I/O.

Since version 0.3.0, *Frinika* supports multi-track audio recording, the accuracy of which is again based on JACK through JJack on Linux systems.

Fig. 7 shows a screenshot of the *Frinika* music sequencer in operation.

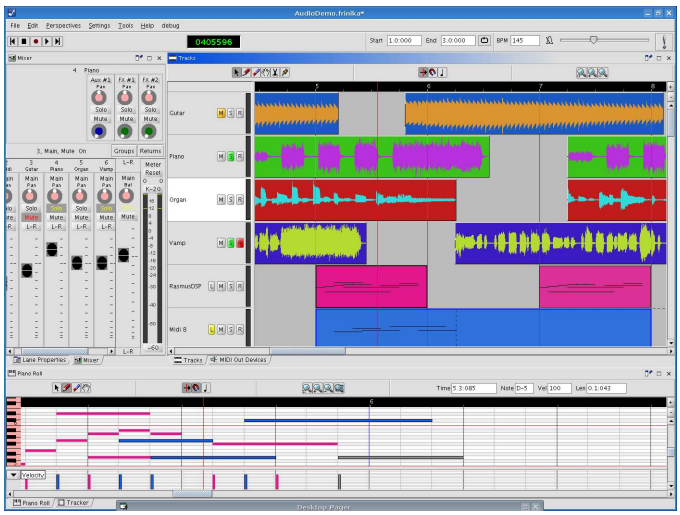


Fig. 7: Screenshot of the *Frinika* music sequencer

10 Conclusion

JJack has proven to be a serious alternative to the standard Java Sound API natively shipped with the Java programming language. Especially music applications for Linux and MacOS that are supposed to interoperate with other audio-processing software can benefit from interfacing with the JACK audio system instead of relying on the audio support internally provided by Java.

The overall architecture of a JACK-based audio application appears more elegant through the continuous use of floating-point arithmetic. It also is expected to be more stable with regard to timing issues, which is suggested by the pull-approach of the JACK processing architecture and the potential use of realtime-capabilities. Depending on the system configuration and driver support, using JJack may also offer lower latency times.

References

[1] Gulden, J., *JJack – JACK to Java bridge API*, software, http://jjack.berlios.de/, licensed under the GNU Lesser General Public License (LGPL)

[2] Davis, P. et al, *JACK – JACK Audio Connection Kit*, software, http://jackaudio.org/, licensed under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL)

[3] Sun Microsystems, *Bean Builder,* a visual programming environment demonstrating the assembly of applications by joining live instances of JavaBeans, software, https://bean-builder.dev.java.net/, licensed under the Berkeley Software Distribution (BSD) License

[4] Sun Microsystems, *New I/O APIs*, http://java.sun.com/j2se/1.4.2/docs/guide/nio/

[5] Sun Microsystems, *Java Sound API,* http://java.sun.com/products/java-media/sound/

[6] Schmeder, A., *PyJack,* JACK client for Python, http://a2hd.com/software, licensed under the GNU General Public License (GPL)

[7] Salomonsen, P. et al, *Frinika music workstation*, software, http://www.frinika.com/, licensed under the GNU General Public License (GPL)

[8] GNU Software Foundation, *GNU General Public License (GPL)*, legal license, http://www.gnu.org/licenses/gpl.txt

[9] GNU Software Foundation, *GNU Lesser General Public License (GPL)*, legal license, http://www.gnu.org/licenses/lgpl.txt

Figure 1 partially by Paul Davis

pnpd/nova, a new audio synthesis engine with a dataflow language

Tim BLECHMANN
Vienna, Austria
tim@klingt.org

Abstract

pnpd/nova is a new dataflow-based computer music system. Its syntax shares a common subset with max-like languages like Pd or Max/MSP, but introduces some new concepts to the dataflow language, most notably an extended and extendable message type system, data encapsulation and namespaces. Currently, it doesn't provide a graphic user interface, but contains a compiler for a text-based patcher language and a command-line interpreter, which can be controlled via OSC.
The audio synthesis engine is designed for supporting low latencies and is optimized for high performance.

Keywords

audio synthesis, computer music system, dataflow language

1 Introduction & Motivation

Max-like dataflow languages have been used in computer music systems since the 1980s. Currently Max/MSP[1] and Pd[2] are the most commonly used programs implementing the max paradigm[3].
Although pnpd/nova is heavily influenced by Max/MSP and Pd, it is not just a rewrite of one of these programs in C++. The dataflow language of pnpd/nova contains language features like namespaces or hierarchical object bindings, that work quite different in other max-like language and should make pnpd/nova much better suited for more complex applications.

2 Type System

2.1 Built-in Types

Like other max-like languages, pnpd/nova has a strong separation between signals and messages. The messaging happens synchronous with the audio signal, to be able to schedule events very tightly, unless it is explicitly detached to low-priority threads by the used objects. The pnpd/nova language contains the following atom types:

bang a simple function call, the atom representation of a bang is `#None`

float a double-precision floating point number

integer an exact integer number[3]

symbol a reference counted, hashed string, to be used as message selector

string a string class (based on the string class from stl)

atomlist a list, containing zero or more atoms

pointer pointer to a user-defined type

2.2 Extending Types

Using the `pointer` type, it is easy to extend the type system by adding custom message types from the C++ interface. Developers of externals can define their own custom message type by simply deriving their classes from the `CustomMessageType` class. All calls to pointer methods use the same inlet handler, the dispatching is done by using the runtime type information of C++. Custom messages are passed by reference. Currently the OSC implementation is based on this feature.

3 Patcher Language

3.1 Text-Based Patcher Syntax

The text-based syntax was designed to describe the patcher language in a human-readable[4] way. It can be used to write patches until there is a graphical patcher available. Listing 1 shows a simple 'hello world' patch.

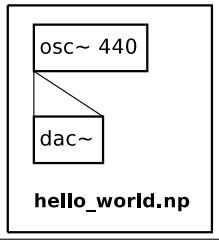
[1] `http://www.cycling74.com/products/maxmsp`
[2] `http://puredata.info/`
[3] based on the GNU Multi-Precision Library `http://www.swox.com/gmp/`
[4] in contrary to the internally used xml format

Listing 1 simple "hello world" patch

```
{
    signal = osc~<440>()
    dac~(signal[0], signal[0])
}
```

It creates a patch, containing a sine-wave oscillator with a frequency of 440 Hz with the symbolic name "signal", that's first outlet is connected to the first two inlets of the `dac~` class, that represents the audio output. The curly brackets `{}` define canvases, round brackets `object()` define object creations, angle brackets `object<args>()` define creation arguments and square brackets `[]` select an outlet of an object.
Connections can be defined implicitly with a comma-separated list in the round brackets as done in Listing 1 or explicitly. The construction `->` declares connections, if inlets or outlets are not specified explicitly, the first outlet is used for the connection. This is shown in Listing 2

Figure 1 hello world patch (block diagram)



Listing 2 hello "world patch" with explicit connections

```
{
    signal = osc~<440>()
    out = dac~()
    signal -> out
    signal -> out[1]
}
```

Both pieces of code represent the same block diagram, that is shown in Figure 1. The written patch files have to be converted to the internal xml file format, which is can be loaded into the interpreter.

3.2 Semantics

Data Encapsulation & Patch Lifetime

pnpd/nova patches are used as independent classes, that should be reusable in different environments. After a patch has been created and it is inserted into the dataflow interpreter, all objects execute their loadbang functions. It is not allowed to insert information into the patch, before this has been done, otherwise it could be used before its initalization is complete, what would obviously lead to severe problems. If there is the attempt of sending information to the patch, before the loadbangs has been executed (via inlets or message busses), the message is queued and run after the loadbangs have been executed. Before a patch is destroyed, endbang functions are executed so that cleanup handlers can be called. At that time, no information is allowed to get into the patch, so calls to inlet functions and message busses will be ignored.

Object Bindings

In Max/MSP and Pd languages, a global scope is used to control the access to objects by symbolic names. If local objects have to be used per-instance, the only workaroud would be to add a new unique symbol to the global scope[5]. pnpd/nova provides a mechanism to avoid that. Bindable objects are not directly bound to the symbols, but each patch contains a container for bindable objects. They can be declared explicitly on a certain scope, or implicitly on the topmost point in the hierarchy, where they are visible in every patch. A `declare` object can be used to declare global or local objects.

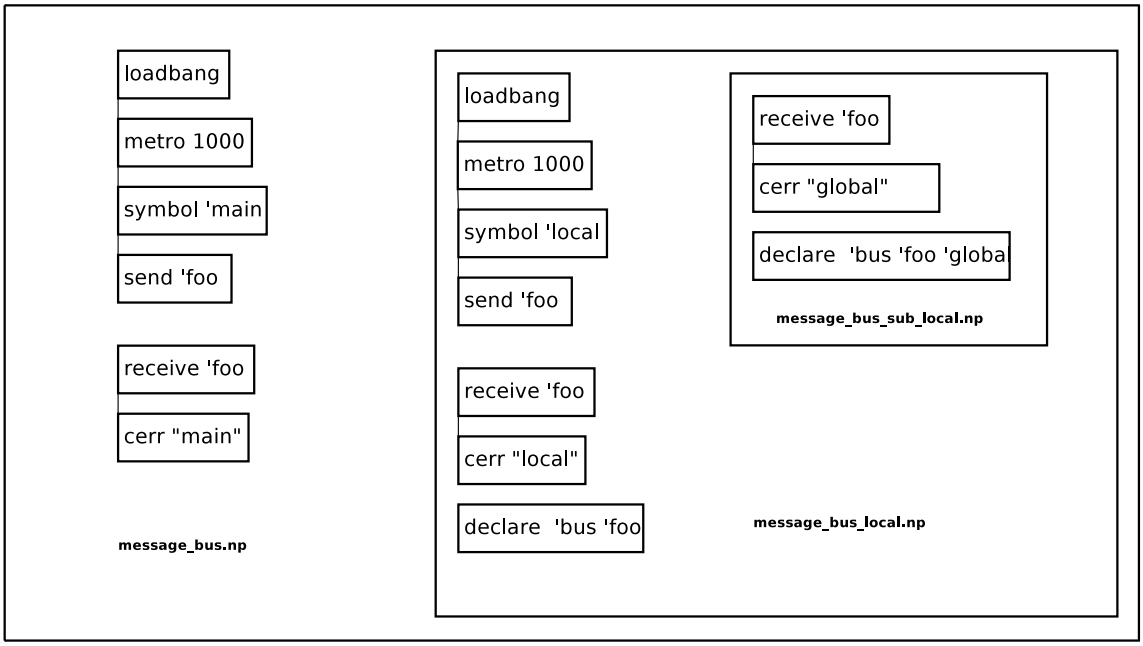
Listings 3, 4 and 5 demonstrate this. In the main patch 'message_bus' (Listing 3), a bus named 'foo' is implicitly constructed in the current scope. In the subpatch 'message_bus_local' (Listing 4) a local bus named 'foo' is explicitly declared, thus the `send` and `receive` objects of this patch are bound to the local bus. The patch 'message_bus_sub_local' (Listing 5), which is lower in the hierarchy than 'message_bus_local', declares a global bus, which is used in this scope. The global bus is located on the top of the hierarchy, where the implicitly declared bus from the top of the hierarchy is then getting rebound to. The block diagram representation is shown in Figure 2.

Namespaces

pnpd/nova features namespaces, which are inspired by the use of namespaces in python. If an object `foo.bar` is created, the interpreter searches for abstractions and externals in the

[5] in Pd this is done with $0, in Max/MSP with #0 prefixes

Figure 2 message_bus patch (block diagram)



Listing 3 message_bus.np

```
{
    metro = metro<1000>(loadbang())
    send<foo>(symbol<'main>(metro))
    cerr<"main">(receive<foo>())
    message_bus_local()
}
```

Listing 4 message_bus_local.np

```
{
    declare<'bus 'foo>()
    metro = metro<1000>(loadbang())
    send<foo>(symbol<'local>(metro))
    cerr<"local">(receive<foo>())
    message_bus_sub_global()
}
```

subfolder `foo` with the name `bar`, and for the external library with the name `foo`, containing the object `bar`. It will be searched relative to the path of the root patch and in user-defined search paths. If different objects are found in more than one place, loading the object fails, to avoid undefined behavior in the case of name clashes.
In future a `using` object will allow to search a certain namespace by default.

Listing 5 message_bus_sub_local.np

```
{
    declare<'bus 'foo 'global>()
    cerr<"sub_global">(receive<foo>())
}
```

4 Some Implementation Details

pnpd/nova is completely written in C++, with the exception of the parser for the text-based patcher, which is written in python. Portaudio[6] is used for audio i/o. Most of the internal thread communication is making use of lockfree C++ data structures. Most system operations are run in low-priority threads in order to make it possible to run pnpd/nova with as little as possible audio dropouts even during times of high cpu load.

4.1 Boost

pnpd/nova relies heavily on the boost C++ libraries[7], as they provide very powerful, portable and stable implementations for many aspects, the C++ standard doesn't deal with. Furthermore, several boost libraries are going to be part of the future C++ standard.
It simplified the memory-management of heap-allocated objects, which is now using smart pointers in certain areas, the spirit parser frame-

[6] `http://www.portaudio.com/`
[7] `http://boost.org/`

work made it much easier to write a decent parser for the string representation of atoms. The boost graph library turn out to be a great help for representing the dsp graph and creating the dsp chain. But also for network i/o (OSC via asio), filesystem traversal, the python bindings and several other parts it simplified the code a lot.

4.2 DSP

The implementation of the dsp core is similar to Pd[2], providing means to suspend parts of the dsp graph in order to save cpu power, or to reblock and overlap specific areas, as it is required for certain applications like windowed spectral processing. As in Pd, reblocking, overlapping and muting is directly bound to the canvases, where the dsp objects are located.
But unlike Pd, it is using nested dsp chains. If a canvas contains a `switch~` or `reblock~` object, a separate dsp chain is used for this and its child canvas. Changes to the dsp graph trigger a generation of a new dsp graph in a background thread. When the new dsp chains are ready, the root chain is exchanged with the obsolete chain. Since the sorting of the dsp graph is done asynchronously, it is possible to do changes to the patches or even load patches without audio dropouts.
The ugens are not necessarily bound to graphical objects. Dsp objects can either provide the dsp function as member function or allocate a special ugen class, depending on the dsp context or the state of the dsp graph. This way, objects can implement several ugens and allocate the most efficient one.

4.3 Performance Notes

pnpd/nova is optimized for high performance, making heavy use of the sse instruction set on modern cpus of the x86 and x86_64 architectures. Although recent compilers are able to vectorize certain code, it is usually not possible to generate optimal code, e.g. because of aliasing issues or the requirement to write algorithms differently[8]. In addition to that, it is using compile-time loop unrolling for performance-critical parts, implemented using C++ template metaprogramming techniques. The results of a general-purpose benchmark[9] against Pd-0.40 with oprofile showed 11537 samples (200000 CPU_CLK_UNHALT events per sample) compared to 27065 samples with Pd.

4.4 Portability

At the moment the only supported platform is linux. However, all dependencies are either platform independent C or C++ libraries or they support linux, osx and win32 and there are plans to provide binaries for other operating system than linux. The sse code is completely separated and plain C++ equivalents exist to assure portability to other architectures than x86. The lockfree algorithms are implemented using atomic_ops[10], which provides a wrapper for the assembler code of the used atomic primitives for several compilers and architectures.

4.5 Extending pnpd/nova

The public C++ api can be used to write externals in C++. Externals are shared libraries that are dynamically loaded at runtime. External developers simply have to derive their external classes from the `GObj` or `GObj_dsp` base classes. The C++ api should be reasonably stable, although the binary compatibility might not be guarrantied, because pnpd/nova is using lots of header implemented inline and template functions.
Beside the C++ api, it is possible to extend pnpd/nova with python. With the `py` class it is possible to run python functions and `pyx` can load python classes, which implement messaging externals.[11]

5 Todo List

pnpd/nova is already usable, but a lot of stuff needs to be done

- graphical user interface (possibly based on gtkmm/gnomecanvasmm or PyQt4)
- extend the object library, lots of objects are still missing
- better documentation of both the patcher language and the library
- testing, testing, testing ...

[8] more details can be found here [1]

[9] fm synth, delayline, filtered noise, sampling objects, signal i/o. testing system: pentium-m 750, 1024MB, linux 2.6.20-rt1

[10] `http://www.hpl.hp.com/research/linux/atomic_ops/`

[11] the interface is inspired by Thomas Grill's py/pyext objects for Pd and Max/MSP, `http://grrrr.org/ext/py/`

6 Conclusions

Although pnpd/nova is still in an early state of development, it is already quite usable. I have been using it in concerts since the late 2006. Lots of features are still missing, especially a gui client with a graphical patcher. However, it already prove to run very efficiently on modern hardware, supporting lowest latencies. We will see, if it is able to compete with Max/MSP or Pd, since both programs have a huge user base. Nevertheless, for users of these programs, switching should be very easy. For now, the most important thing beside manpower for writing a gui, is to find some users, who are willing to do beta testing.

7 Acknowledgements

I'd like to thank Miller Puckette for Pd and for making it open source, because i learned a lot when reading the Pd sources and it inspired pnpd/nova's design and dieb13 for hosting the project at klingt.org.

References

[1] T. Blechmann. Simd in dsp algorithmen. `https://tim.klingt.org/pnpd/Members/tim/iem.pdf`.

[2] M. Puckette. Pure data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[3] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

Developing LADSPA Plugins with Csound

Victor Lazzarini
National University of Ireland, Maynooth
Maynooth
Co. Kildare
Ireland
victor.lazzarini@nuim.ie

Rory Walsh
Dundalk Institute of Technology
Dundalk
Co.Louth
Ireland
rory.walsh@ear.ie

Abstract

Csound is one of the most powerful audio programming languages available to electroacoustic composers today. It origins can be traced directly back to Max Matthews in Bell Labs and since its inception it has grown to become one of the most extensive computer music toolkits in development. This paper will describe a new toolkit for the development of LADSPA plugins using the Csound audio programming language. The toolkit itself was developed using the new Csound API and the Linux Audio Developers Simple Plugin API. This text will explore the implementation of said toolkit and conclude with examples of the toolkit in use.

Keywords

Computer Music, Audio Plugins, Musical Signal Processing

1 Introduction

csLADSPA is a new toolkit for the development of LADSPA[1] plugins using the Csound audio programming language. csLADSPA provides musicians with no low-level programming experience with a simple albeit powerful toolkit for the development of audio plugins. The main goal of this project keeps in line with one of the main objectives of the LADSPA project i.e., to create a 'simple' architecture for the development of plugins. It was of the utmost importance to the author that the end-user need only a rudimentary knowledge of Csound in order to get started with csLADSPA. It is expected that a novice user will in time explore more of Csound to create complex plugins.

1.1 The Csound host API

An API (application programming interface) is an interface provided by a computer system, library or application, which provides users with a way of accessing functions and routines particular to the control program. Essentially APIs provide developers with a means of harnessing an existing applications functionality within a host application.

The Csound API[2] can be used to start any number of Csound instances through a series of different calling functions. The API also provides mechanisms for two way communication with an instance of Csound through the use of a 'named software bus'. In short, the Csound API makes it possible to harness all the power of Csound in ones own application.

1.2 LADSPA Plugins

The LADSPA framework came to fruition in early 2000 following the combined efforts of Paul Davis, Richard W.E. Furse, and Stefan Westerfield[3]. LADSPA provides developers with a simple way of developing audio plugins which can be loaded by a wide range of host applications. In terms of implementation, all LADSPA plugins must:

- Declare a plugin structure (for audio buffers, etc.).
- Instantiate the plug-in by calling a user defined function which returns a LADSPA_Handle data type.
- Register the plugin using a user-defined 'ladspa_descriptor()' function.
- Connect ports to data locations using a user-defined connect function.
- Process blocks of samples in a run function.
- Free memory

2 Inside csLADSPA

The csLADSPA library was written in C++[4]. It is programmed using the same basic structure as any LADSPA plugin. Calls are made to the Csound API at different stages in the execution of the plugin. The Plugin declares a data structure, which includes a pointer to an instance of the Csound class, an array to hold control values and an array to hold the names of the software bus channels which will be used for parameter control. The steps taken in the actual operation of the plugin are as follows:

- When the plugin is loaded, all Csound (.csd) files which reside in the plugin folder are parsed. This data is assigned to the various members of the LADSPA descriptor structure. This step generates a series of plugins based on the Csound source code files created by user.
- When a plugin is instantiated, csLADSPA creates a Csound instance and compiles the respective Csound source code.
- This is followed by the connection of ports to data locations and the assignment of control values to an array of floats. This can be assessed from inside the run function, which is called by the host application.
- When the host runs the plugin, blocks of samples are processed in a 'run' function. This accesses the Csound instance low-level IO buffers, through calls to Csound::Spin() and Csound::Spout(), in order to route the selected audio to it. Finally, inside a processing loop a call is made to Csound::PerformKsmps(), which does the actual signal processing.

A basic model of how the plugins work is shown in below (fig.1). The host application loads the csLADSPA plugin. When the user hits the process button the csLADSPA library will route the selected audio to an instance of Csound. Csound will then process this audio and return it to the csLADSPA plugin which will then pass that audio to the host application.

2.1 Getting started

In order to get started writing csLADSPA plugins the end-user must place the csLADSPA library and all csd plugin files in the folder pointed to by the LADSPA_PATH environment variable. LADSPA_PATH must be set in order for csLADSPA library to work. The csLADSPA library will automatically detect all Csound files and load them as separate plugins. In order to keep things simple, csLADSPA only works with the unified Csound file format.

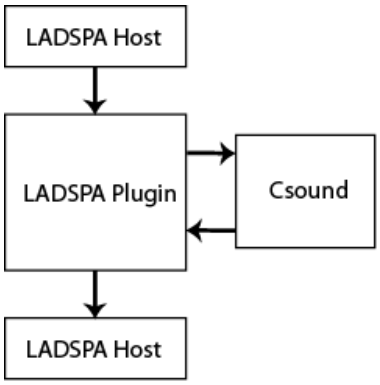


Figure 1. The csLADSPA model

2.2 csLADSPA tags

In order for csLADSPA to load the Csound files in the LADSPA_PATH the user must specify some basic information about the plugin. This is done by adding a section at the top of the Csound file whereby the user can specify things like the name of the plugin, the author, etc. It is in this section that the user can also specify the control ports they will need in order to interact with their Csound code when running the plugin through a host. Every csLADSPA plugin must specify the following:

Tags	Description
Name	The name of the plugin as it will appear in the host app-lication
Maker	Author of plugin
UniqueID	ID given to plugin, each plugin should use a unique ID.
Copyright	Copyright/Licence notice

If users wish to add controls to their plugins they can do so using the following tags:

Tags	Description
ControlPort	The name of the control as it appears when the plugin is ran and the name of the channel which Csound will retrieve the data on. The two names should be separated by a '|' symbol.
Range	Plugin max/min range. Again the two values are separated by a '|' symbol. If users wish to controls to respond logarithmically they can add a '&log' after they specify the range values.

Note that if a user uses the `ControlPort` tag they must *always* place a `Range` tag underneath it. Examples of how these tags are used can be seen in the next section.

3 Examples

In the following section three csLADSPA plugins will be presented. The first two plugins will illustrate the mechanisms for communication between the host and the csLADSPA plugin. The third and final plugin will illustrate a more complex process which makes use of some of the more advanced opcodes included with Csound5, i.e., the PVS[5] opcodes.

3.1 A basic gain plugin

Given that most plugin SDKs come with a simple 'gain' example we'll start here too. Here is the full code to a simple gain example.

```
<csLADSPA>
Name=Gain Plugin
Maker=John Doe
UniqueID=1049
Copyright=None
ControlPort=Gain|gain
Range=0|2
</csLADSPA>
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 1

instr 1
      kGain chnget "gain"
      ain in
      out ain*kGain
endin

</CsInstruments>
<CsScore>
i1 0 3600
</CsScore>
</CsoundSynthesizer>
```

As previously mentioned the means of communication between the plugin and the instance of Csound is provided by the named software bus, in this case the name given to the software channel is 'gain'. In Csound we can use the `chnget` opcode to retrieve data from a particular software bus. In the case above this data is used to multiply the output signal by a value between 0 and 2, as defined by the `Range` tag in the `<csLADSPA>` section of the above code.

3.2 A simple flanging plugin

Flanging is a commonly used digital audio effect. It's created by mixing a signal with a time-varying delayed version of itself. In Csound this can be done by using the vdelay opcode. To control the amount of delay one can use a low frequency oscillator or LFO. This plugin will need two control ports, one for the flange depth and one for the flange rate. Here is the full code for a simple 'flanger' plugin.

```
<csLADSPA>
Name=Flanger
Maker=John Doe
UniqueID=1054
Copyright=None
ControlPort=Flange Depth|depth
Range=0|1
ControlPort=Flange Rate|rate
Range=0|10
</csLADSPA>
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 1

instr 1
      kdeltime chnget "depth"
      krate chnget "rate"
      ain in
      a1 oscil kdeltime, 1, 1
      ar vdelay3 ain, a1+kdeltime, 1000
      out ar+ain
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
i1 0 3600
</CsScore>
</CsoundSynthesizer>
```

3.3 A spectral manipulation plugin

Csound5 comes with a host of new Phase Vocoder Streaming, PVS, opcodes. These opcodes provide users with a means of manipulating spectral components of a signal in realtime. In the following example the opcodes `pvsanal`, `pvsblur` and `pvsynth` are used to manipulate

the spectrum of the selected audio. The plugin averages the amp/freq time functions of each analysis channel for a specified time.

```
<csLADSPA>
Name=PVSBlur
Maker=John Doe
UniqueID=1056
Copyright=None
ControlPort=Max Delay|del
Range=0|1
ControlPort=Blur Time|blur
Range=0|10 &log
</csLADSPA>
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 1

instr 1
imaxdel chnget "del"
iblurtime chnget "blur"
asig  in
fsig  pvsanal   asig, 1024, 256, 1024, 1
ftps  pvsblur   fsig, 0.2, 0.2
atps  pvsynth   ftps
out atps
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
i1 0 3600
</CsScore>
</CsoundSynthesizer>
```

4 Conclusion

The current version of csLADSPA performs adequately and has been tested by students in the National University of Ireland Maynooth and at Dundalk Institute of Technology, Ireland. The system has been used both as a creative tool and as a pedagogical utility used in the teaching of DSP techniques. It has been fully tested in real-time using Jack-Rack[6] and in non-realtime mode using Audacity[7]. With regards to future developments the authors are currently working on a better error-checking system and multichannel support is also being investigated.

csLADSPA is Free Software, available for download from www.ear.ie/csLADSPA.htm

References

[1] http://www.ladspa.org.

[2] John ffitch. 2004. On The Design of Csound5. Proceeedings of the 3rd Linux Audio Developers Conferencence. ZKM, Karlsruhe, Germany.

[3] Dave Phillips, Linux Audio Plug-Ins: A Look Into LADSPA: http://www.linuxdevcenter.com/pub/a/linux/2001/02/02/ladspa.html

[4] Bjarne Stroustrop. 1991. The C++ Programming Language, second edition. Addison-Wesley, New York.

[5] Victor Lazzarini, Joseph Timoney and Thomas Lysaght. 2006. Streaming Frequency-Domain DAFX in Csound 5. Proc. of the 9th Int. Conf. on Digital Audio Effects (DAFX) 2006, Montreal, Canada. pp.275-278.

[6] http://jack-rack.sourceforge.net/

[7] http://audacity.sourceforge.net/

A Tetrahedral Microphone Processor for Ambisonic Recording

Fons ADRIAENSEN
Laboratorio di Acustica ed Elettroacustica, Parma, Italy
fons@kokkinizita.net

Abstract

This paper introduces a Linux audio application that provides an integrated solution for making full 3-D Ambisonics recordings by using a tetrahedral microphone. Apart from the basic A to B format conversion it performs a number of auxiliary functions such as LF filtering, metering and monitoring, turning it into a complete Ambisonics recording processor. It also allows for calibration of an individual microphone unit based on measured impulse responses. A new JACK backend required to make use of a particular four-channel audio interface optimised for Ambisonic recording is also introduced.

Keywords

Ambisonics, tetrahedral microphone, recording.

1 Introduction

The standard first-order Ambisonics B-format consist of four signals named W, X, Y and Z. In acoustic field theory terms, W represents the *pressure* signal at a given point in space, and X, Y, Z the three components of the *velocity vector* at the same point, projected onto orthogonal axes. Conventionally X points forward, Y left, and Z up.

These four signals also correspond to the outputs of four real microphones - an omnidirectional one for W, and three figure-of-eight ones for X, Y and Z - provided one can find a way to put these four microphones at exactly the same point in space.

For horizontal-only surround recordings Z is not used, and it is possible to mount the three required mics close together in a vertical line so they are effectively coincident for sounds arriving from horizontal directions. But for full 3-D such a *direct B-format* setup is no longer practical.

One solution, already developed by Michael Gerzon e.a. in the early years of Ambisonics [Gerzon, 1975], is to use a *tetrahedral microphone*. This contains four cardioid or near-cardioid capsules mounted very close together at the vertices of a regular tetrahedron and pointing outwards, as shown schematically in fig. 1. Figure 2 shows an example of how this may be realized in practice.

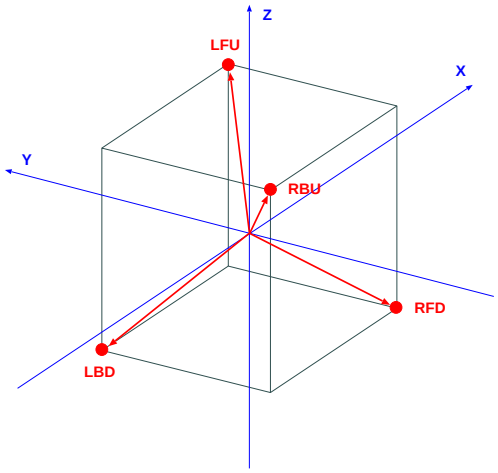


Figure 1: Tetrahedral mic geometry.

Given the four A-format microphone signals LFU, RFD, RBU and LBD[1], we can find the B-format signals from

$$\begin{aligned} W' &= LFU + RFD + RBU + LBD \\ X' &= LFU + RFD - RBU - LBD \\ Y' &= LFU - RFD - RBU + LBD \\ Z' &= LFU - RFD + RBU - LBD \end{aligned}$$

To find the correct W, X, Y, Z, we also need to filter the outputs of this *A-B matrix* and apply some gain factors. Two types of filter are required, one for the zero order component W, and a second one for each first order one, X, Y, Z. These filters will be discussed in more detail in section 3.

The first tetrahedral mics were manufactured by Calrec Ltd. in the UK, and their technology

[1] LFU means *left-front-up*, etc.

Figure 2: A tetrahedral mic (ST250).

was later acquired by Soundfield Ltd.[2] These are quite expensive microphones, not only because of their high quality, but also because they need a hardware processing unit to perform the A to B-format conversion and filtering mentioned above, and in particular because for best results each microphone needs to be calibrated individually and provided with a matched A-B matrix and/or filters.

The Danish microphone manufacturers DPA[3] were the first to offer a tetrahedral mic without a processing unit. It was a very high quality mic, but it is unfortunately no longer available. Soundfield Ltd. also produce such an A-format microphone, the SPS200-A.

Recently, Core Sound[4] has announced the relatively inexpensive *Tetramic*. It should be available when this paper is presented, and its price is expected to be below 1000 USD. It comes without a controller, and relies on a software solution for A to B-format conversion.

Given such an application, this mic provides an affordable solution for Ambisonic recording. While it was this announcement that triggered the development of the *Tetraproc* software presented in this paper, it should be pointed out that this software can be used with any tetrahedral microphone. It is published under the GPL license, and in no way, either technically or commercially, linked to products of Core Sound or any other manufacturer.

2 Tetraproc processor architecture

Apart from the basic A to B format conversion, Tetraproc also provides some convenient monitoring functionality. Ambisonic microphones are often used for live recording, and in these circumstances one wants a system that is easy to set up and use, and that allows for verification of the recording chain. In practice the only other software needed should be the JACK server and a recording application such as Ardour.

Figure 3 (next page) shows the processing chain implemented in Tetraproc. The high pass filters, the mute and invert switches, and the monitoring functions are controlled by the graphical user interface. All other modules are set up using a separate configuration window, and these settings are saved into configuration files. These config files are also generated by a separate calibration program discussed in section 4.

2.1 A to B format conversion

Going from the A-format microphone inputs to the B-format output, the following processing steps are performed:

- **High pass filtering.** This has an adjustable cutoff frequency and a slope of 24 dB/oct. This is really an essential feature. Figure-of-eight microphones having a good low frequency response are also excellent detectors of earthquakes, passing underground trains, waggling mic stands, slamming doors and air currents. These can result in rather large amplitude signals, and cutting off low frequencies is the only way to get rid of them. Ideally this should be done before AD-conversion, but not all audio interfaces provide such filters.
- **Mute switches.** For testing connections and verifying correct operation of the microphone it is convenient to be able to listen to selected inputs, hence the mute switches which are provided on the GUI.
- **Low frequency parametric filtering.** This is provided to adjust the frequency response of the microphones in this region. The parameters provided are centre frequency, bandwidth and gain. The same filtering is applied to all four channels.
- **A-format FIR filters.** These are implemented using fast FFT-based convolution and can be used to correct the frequency and phase response of the four microphones using filters calculated from measured impulse responses. This will be mainly im-

[2] <http://www.soundfield.com>
[3] <http://www.dpamicrophones.com>
[4] <http://www.core-sound.com>

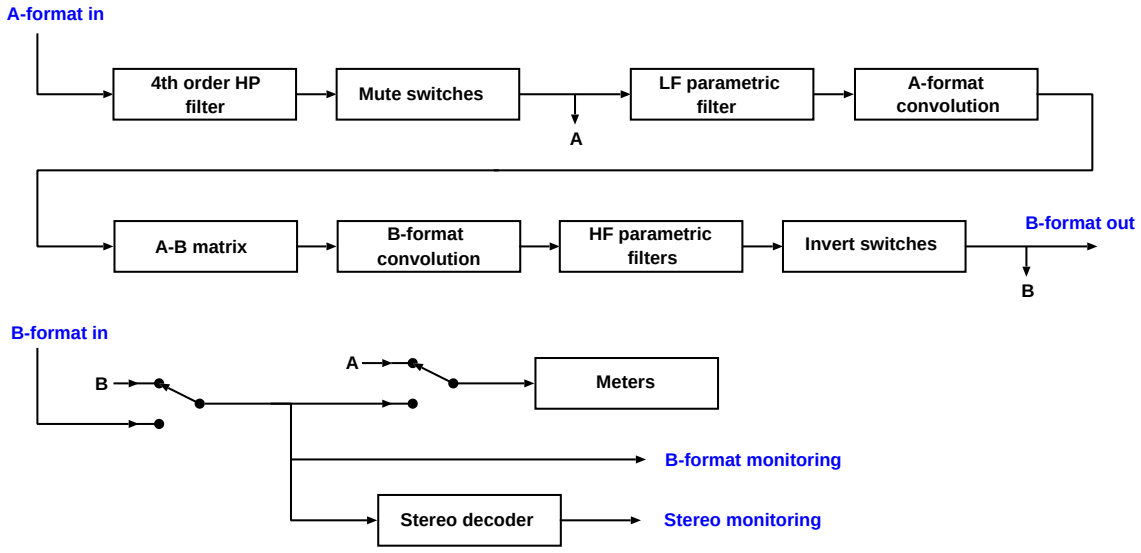


Figure 3: The Tetraproc processing chain

portant in the medium and high frequency regions.

- **A-B matrixing.** This performs the A-B transformation already described in the previous section. The actual matrix coefficients are modified to compensate for small gain and directivity mismatches between the four microphones. They are calculated by the calibration program described in section 4.

- **B-format FIR filters.** Again using fast convolution, these may be inserted to implement Angelo Farina's method (see next section) for obtaining the post-matrix filtering. They can be used together with the parametric sections that follow.

- **HF parametric filtering.** Two sections are provided in each channel to realize the required post-matrix filters. Parameters for these filters will be preset to sensible defaults and can be tweaked for optimum performance during the calibration of a tetrahedral microphone.

- **X, Y, Z inversion switches.** In some cases it is required to invert some of the first order signals, for example when the microphone is used upside down, hanging from its cable.

The output at this point is the B-format signal that will be recorded.

2.2 Monitoring functions

Monitoring can be switched between the B-format signal being recorded, or one being played back. A virtual stereo microphone with adjustable azimuth, elevation, microphone angle and directivity is provided for stereo monitoring. This module also provides a volume control and optional low-frequency crosstalk for headphone listening.

Four bargraph meters are provided on the GUI. These show either the A-format signals, or the B-format signal being monitored.

2.3 DSP implementation issues

None of the processing steps above present any real difficulty, but some attention to detail is required in order to obtain the highest quality.

The high pass filtering is implemented using a filter architecture optimised for low frequency filters described in [Adriaensen, 2006]. It is used to avoid problems with filter coefficient and signal quantisation which may arise with some standard digital filter structures.

When both the A-format and B-format FIR filters are enabled, they are combined together with the A-B matrix into a single four by four convolution process. This doubles the number of convolutions, but is both more efficient and more accurate than using eight separate ones.

The parametric filter sections use the Mitra-Regalia architecture.

3 Choice of the A-B matrix filters

During the study phase for the Tetraproc software it became clear that the choice of the post A-B matrix equalisation filters for a tetrahedral mic is sort of a black art. Theoretical analysis is possible, but due to conflicting requirements it does not lead to an obvious best solution. The problem gets more complicated again when the non-ideal characteristics of real microphones, in particular at high frequencies, and diffraction effects are taken into account — assuming the details of these are known at all.

The filters are necessary because at higher frequencies the size of the microphone array becomes comparable to the wavelength, and the microphones can no longer be considered to be coincident. The result is that while a B-format microphone has very good polar patterns at low and medium frequencies (better than most real omni or gradient mics), these will degrade at higher frequencies and break down above about 10 kHz (as they do with real microphones, except some of the very best). The frequency response of the B-format signals at high frequencies depends on the direction of the sound, and a compromise has to be found.

A second reason why these filters are necessary is to ensure that the B-format signals remain exactly in phase over the entire useful frequency range — this phase relationship is an essential feature of the Ambisonic system.

Considering just the theoretical response assuming perfect microphone capsules, there is already the choice between equalising for flat free field response in some preferred directions (e.g. the cardinal axes), or for flat diffuse field response, considering signals arriving from all directions.

In his famous paper [Gerzon, 1975] which seems to be one of the few original and authoritative publications on the subject,[5] Michael Gerzon notes that since the effective polar patterns become quite complex at high frequencies, it would be best to equalise for flat diffuse field response, and also shows plots of the corrections that would be required to do this. Gerzon provides design parameters for some analog filters, but these do not match the diffuse field curves. The filters actually used in one of the products based on his work [Calrec Ltd., 1984] are again different.

The theoretical curves depend on the radius of the array, and even more strongly on the directivity of the microphone capsules used. The latter will have some nominal value but it will in practice not be constant over the full frequency range. The actual values at high frequencies will be unknown in most cases.

In the light of all this, it seems unwise to include only fixed post-matrix filters in an application such as Tetraproc. It is for this reason that the two sections of parametric filtering are provided. The software package will contain a number of parameter presets for these filters, corresponding to the theoretical curves for a range of array diameters and directivities, and in many cases one of these may prove to be satisfactory. But for best results the filter settings should be derived from a calibration procedure, as outlined in the next section.

While the parametric filter approach seems to be the one preferred by some specialists in this field [Lee, 2006], a rather different one was suggested by Angelo Farina. He proposes to use Kirkeby-inverted measured impulse responses not only for equalising the individual microphone capsules, but also for the the post-matrix filters [Farina, 2006]. The Tetraproc software also permits the use of this method by providing the post-matrix convolution step.

4 The calibration procedure

While it is possible to use *Tetraproc* with a default configuration for a given microphone type (the available models are slightly different in terms of geometry and polar patterns), best results will be obtained only if the A-B conversion process is calibrated for each particular microphone. This is because each mic capsule will have its own small deviations from nominal sensitivity, directivity and frequency response. For a normal mic these don't matter much, but they become significant when the signals from a number of mics recording the same sound are combined in a way that relies on cancellation, as is the case for the first order outputs of the A-B matrix.

The complete calibration procedure can be divided into three parts.

- The most important part is the compensation for any mismatching in sensitivity and directivity of the four capsules that make up a tetrahedral microphone. Any errors here will result in defective polar patterns

[5] The PDF file of this paper as available from the AES is a scanned version of the original typewritten document and some parts of it are difficult to read. A typeset version is available from this author on request.

of the virtual B-format microphones over the entire frequency range. The compensation is done by adjusting the coefficients of the A-B matrix.

- A second aspect is the adjustment of the post-matrix equalisation. This EQ compensates for the fact that at medium and high frequencies the four capsules are no longer coincident, as discussed in the previous section. In particular at higher frequencies (above about 5 kHz), this involves many unknown factors, and it will always be a compromise between conflicting requirements.
- Finally, the low frequency response of the microphone can be adjusted. Almost all directional mics will show some sort of high-pass response in this region. For some applications, e.g. orchestral recordings, this should be equalised. In practice this will vary little between mics of the same type, so default equalisation parameters will do in many cases.

A separate calibration application has been developed to assist the user with this rather complicated process. This program requires a number of measurements as its input. For the simplest case, when only the sensitivity and directivity compensation is performed, this involves eight recordings of a test signal reproduced by a loudspeaker, made at intervals of 45 degrees in the horizontal plane. This is a relatively simple operation that can be performed by all users. Alternatively, it is possible to use eight impulse responses measured using a program such as *Aliki*. This procedure also requires a separate omnidirectional measurement microphone, to compensate for the frequency response of the speaker used to reproduce the sweep signals. If these IR are available it is also possible to adjust the post-matrix equalisers, or to use the B-format convolution. Using the A-format convolution to equalise each capsule separately requires another set of measured impulse responses. For adjusting the low frequency parametric equaliser, four recordings of a special test signal reproduced using a small point source speaker are required.

Some parts of the application program work fully automatically, and others require interaction with the user who is required to interpret some results presented in a graphical format. It is well beyond the scope of this paper to explain the processing performed by the calibration program in any detail. A complete description together with instructions on how to perform the required measurements will be available in the manual for this application.

5 Supporting the 4Mic interface

Together with the *Tetramic*, Core Sound also introduces an audio interface optimised for use with this microphone. The *4Mic* interface offers four phantom powered inputs, precisely tracking gain controls, and AD conversion up to 24 bit and 192 kHz. Output is via one or two SPDIF streams. The unit can be configured to multiplex the four channels onto one stereo SPDIF stream at the double sample rate. The purpose of this is to enable the use of existing portable 2-channel recorders to capture the four A-format signals. The resulting stereo WAVEX file can then be converted to the required 4-channel one by just modifying its header.

Since there seem to be few audio interfaces offering two SPDIF inputs, it seems attractive to use the multiplexed output format also when recording via *Tetraproc*. This raises the question of where to implement the demultiplexing. The following have been considered:

- **In the drivers.** This would require changes to the source code of all drivers supporting and SPDIF interface, and is therefore not a viable option.
- **In libalsa.** It would probably take a long time to get this accepted into the ALSA source tree, assuming it would be accepted at all.[6]
- **In a libalsa plug-in.** This seems to be impossible as the plug-in interface does not allow for different input and output sample rates.
- **In an application.** It would be possible to include the demultiplexing in e.g. Tetraproc, but this would mean that the entire JACK graph has to work at the double sample rate, and the demultiplexed channels would need to be up-sampled, resulting in a useless doubling of the recorded file sizes.

[6] The ALSA developers do not seem to be interested in the problem. The author posted several requests for information on the ALSA developers mailing list and did not even receive a single reply.

- **In a JACK backend.** This is the solution that was finally chosen.

The *fourmic* backend supplied with *Tetraproc* opens the ALSA driver at twice the sample rate and period size as seen by the JACK engine. It demultiplexes a selected stereo pair to four capture ports, and optionally also up-samples up to four playback ports so they can be used for monitoring.

There is one problem with this demultiplexing. The first samples delivered by the ALSA drivers seem to correspond to a random offset into an SPDIF frame. So there is a one in two chance that this is an odd offset, resulting in the channels being swapped and two of them having a one sample delay w.r.t. the others. The ALSA API does not seem to provide any means to find out the current position in an SPDIF frame, so the only solution for this at the time of writing is to check the channel assignments and restart JACK if they are wrong.

6 Acknowledgements

The design of this software would not have been possible without the generous help of the small community of Ambisonic experts.

I would like to thank Richard Lee for sharing his ideas on tetrahedral microphone alignment and equalisation. Also many thanks to the members of the Sursound mailing list, in particular Aaron Heller, Angelo Farina, Dave Malham, David McGriffy and Eric Benjamin, and to Len Moskowitz of Core Sound.

References

Fons Adriaensen. 2006. Near field filters for higher order Ambisonics. Available from <http://www.kokkinizita.net/linuxaudio>.

Calrec Ltd. 1984. Technical manual for the Mk4 soundfield microphone.

Angelo Farina. 2006. A-format to B-format conversion. <http://pcfarina.eng.unipr.it/Public/B-format/A2B-conversion/A2B.htm>.

Michael Gerzon. 1975. The design of precisely coincident microphone arrays for stereo and surround sound. *50th Audio Engineering Society Conference*, (AES preprint 8083L20).

Richard Lee. 2006. Sound field alignment and EQ. Private communication.

Audio Metering and Linux

Andrés CABRERA
Departamento de Música, Pontificia Universidad Javeriana de Colombia
Carrera 7 No. 40 - 62
Bogotá,
Colombia
andres@geminiflux.com

Abstract

This documents presents an overview of current audio level and loudness measuring techniques and concepts, in the context of musical and post-production environments. It proposes a metering application, which is currently in early development, which should address metering needs currently unsatisfied in Linux.

Keywords

Audio level metering, Peak Meter, RMS Meter, PPM Meter,Equal loudness

1 Introduction

Sound is energy propagating as compression and rarefaction through a medium, but when captured, or artificially generated it is represented as a variation in voltage (or a representation of this variation as digital samples, grooves on a record or variations in tape magnetization). The ear is a sound pressure sensor which together with the brain produce the experience of hearing. There is a technical and practical need to quantify sound pressure levels and perceived loudness, which is done with sound level and loudness meters. A particular setting for the usage of audio metering presents itself in the production of music and sound for audiovisual productions.

1.1 Decibels

The ear senses sound pressure non-linearly, so the unit developed to quantify audio level, called decibel (dB), is logarithmic, and has been defined as:

$$L_1 = 10log_{10}\frac{W_1}{W_2} \quad (1)$$

Where the value in decibels of power W_1 is L_1. Decibels are always expressed in relation to a reference power W_2. This definition can be used to calculate decibels when the measurement is in acoustic power or intensity, or electric power. This is not practical, since most often we measure air pressure, voltages or current. From their relation to power, we can express equation (1) as:

$$\begin{aligned} L_p &= 10log_{10}\frac{p_1^2}{p_2^2} \\ &= 20log_{10}\frac{p_1}{p_2} \end{aligned} \quad (2)$$

The standard reference pressure for sound is 20 μPa. If this reference level is used to measure sound pressure level (SPL) in the air, the result is expressed as dB_{SPL}. Decibels in reference to voltage level are also indicated with subscripts, like dB_u for a reference level of 0.775 $Volts$.
Equation (2) can be used to calculate decibels in full scale (dB_{FS}) for PCM digital systems, where the absolute maximum sample value corresponds to $0dB_{FS}$, and amplitudes are expressed as negative numbers below it. When signed integer samples (8-bit *short*, 16-bit *int* or 24-bit *long* types) are used, the following formula can be used to calculate full-scale decibels:

$$\begin{aligned} L_{FS} &= 20\frac{lnA_{max}}{ln10} - 20\frac{ln \mid A_i \mid}{ln10} \\ &= 20\frac{ln(A_{max}/ \mid A_i \mid)}{ln10} \quad \mid A_i \mid > 0 \\ &= -\infty \quad A_i = 0 \end{aligned} \quad (3)$$

Where the value in full-scale decibels L_{FS} for absolute amplitude A_i is calculated as the difference with the decibel value for the greatest possible sample value A_{max} in a given bit precision. The reference value is the minimum sample value available (i.e. 1) and natural logarithm is used since it is often more practical in computer systems to use them to take advantage of the *exp()* function from the standard *math.h* library, when calculating amplitude from dB_{FS} values.
The value of A_{max} is:

$$A_{max} = 2^{n-1} \quad (4)$$

Where n is the integer sample bit depth, and 1 is subtracted from the exponent because samples values oscillate around a center, so the total number of values must be divided by two to represent positive and negative displacement (when using signed data types).
When using floating point samples, there is no set reference level, but when using the usual range from 1 to -1, the following can be used to calculate dB_{FS}:

$$\begin{aligned} L_{FS} &= 20\frac{ln(1/\mid A_f \mid)}{ln10} \\ L_{FS} &= -20\frac{ln\mid A_f \mid}{ln10} \quad \mid A_f \mid > 0 \\ &= -\infty \quad A_f = 0 \end{aligned} \tag{5}$$

Note that the absolute value of the amplitude must be used in both cases.

1.2 Root Mean Square

The previous method of calculating decibels yields "instantaneous" values reflecting a temporary state that doesn't represent actual energy in an oscillation. For this reason, sound volume (and voltage average) is sometimes calculated using Root Mean Square (RMS), which presents an average that better describes an oscillating signal's level. The RMS value is always calculated for a certain period of time between T_1 and T_2 and is defined as:

$$f_{rms} = \sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2} [f(t)]^2\, dt} \tag{6}$$

For discrete values, like digital PCM samples, a signal's RMS value for a group of N samples is:

$$x_{rms} = \sqrt{\frac{1}{N}\sum_{k=1}^{n} x_k^2} \tag{7}$$

1.3 Loudness

There is a fundamental difference between perceived loudness and nominal audio levels. Audio with a high level might sound softer than one with lower levels. There are several reasons for this.
It is well known that the human ear perceives loudness as a function of amplitude and frequency. The recent ISO 226:2003 standard [1] presents a revision of the well known Robinson-Dadson (in turn based on the Fletcher-Munson) curves. It defines a set of equal loudness contour lines on a frequency vs. sound-pressure level graph, relating sound pressure level and frequency to subjective human perception of loudness. The curves show that the human ear is most sensitive to frequencies between 3000 and 5000 Hz, and that at higher sound pressures, the difference in sensitivity is reduced, making the curves flatter.

It has also been shown that the ear perceives short transients softer than a longer equivalent sound [2]. Short transients might report a high level on a peak meter, but not be perceived as loud by a listener. This effect is significant for sounds shorter than 100ms, and becomes more evident for shorter sounds.

Another important concept is long-term loudness . This is an average loudness over a longer period of time (usually a complete program) which quantifies perceived loudness, to be able to compare and match with other material [3].

2 Uses of Level Metering

There are three uses for level metering:

- To keep levels within equipment limits
- To assist subjective judgement of levels
- To comply with delivery or industry standards

Controlling signal levels is important in all aspects of audio production and delivery, from recording to mastering. Audio equipment, whether analogue or digital, has a threshold above which, audio isn't accurately represented. If this threshold is exceeded, distortion occurs. This is sometimes used as an effect, but is undesirable in most cases. If a signal level is too low it might be degraded by a device's own noise floor. Metering guarantees a clean signal path, helping the musician or engineer stay within equipment limits. For this application, the best suited meters are ones that can show potential problems clearly. Meters with a fast response, that can clearly show peaks, are the most suited.

Meters can be helpful as an objective means of measuring audio, when deciding relative mix levels, or to help achieve consistent loudness between different sources. Even though the ear should be the final judge, meters can help a tired ear, or an operator in an unusual or inadequate environment to better judge audio levels.

When delivering content for broadcast or mastering, it is ideal and sometimes compulsory to comply with certain standards, which

vary greatly depending on the situation and destination. Adequate metering will help audio material pass quality control standards and delivery requirements. Adequate metering together with proper speaker calibration will assist in producing adequately loud masters, while respecting standard program loudness. Film sound has had a "standard" calibration setting of $85dB_{SPL}(C)$ for $0VU$[4]. This, though not really a standard, is a respected practice -mostly-throughout the industry. This technique is explored for other settings in Bob Katz's K-system [5], which is a combination of metering/calibration guidelines.

3 Techniques for measuring audio

Throughout the history of audio technology, different techniques have been used to measure audio levels. Most of these techniques are deeply related to the medium they measure, and vast differences can be seen for instance in meters targeted at analogue and digital systems. Several techniques have been developed to imitate subjective perception of loudness.

3.1 VU

Originally developed in 1939, the VU (Volume unit) meter is the oldest type of metering still in usage, and consists of power measurements with time averaging (sometimes referred to as the meter's ballistics). This type of meter is calibrated so that 0 Volume Units represent 1 milliwatt of sine-wave power at 1000 cycles per second (a 1000 Hz tone at $+4dB_u$) [6]. A VU meter has rise and fall times of 300 ms, sometimes called the time integration constant. This means that the VU should take 300ms to reach 99% of the target voltage value. The VU meter is not particularly well suited to detect problematic peaks, or to measure music loudness. Still, it can be useful, as many analogue consoles (particularly low and mid-range studio consoles) provide little headroom above 0VU and produce audible distortion even for slight peaks above 0VU. Although today there are better systems to measure speech loudness (See section 3.4) and detect problematic peaks, VU meters are still ubiquitous, specially in analogue equipment, and their measurements are still used for delivery standards in certain cases.

3.2 PPM

To address some of the short-comings of VU meters, Peak Programme Meters (PPM) were developed and standardized. These meters, sometimes called quasi-peak meters, are also time averaged RMS meters, but have a much faster attack than VU meters, therefore showing potential peak problems more accurately. They don't have instantaneous response, but they were designed according to the ear's ability to detect short distortion. There are several types of PPM meters[1] [7] :

Nordic These meters have an attack time of $5ms$ (for 77.77% target level) and a decay time of 1.5 seconds for $20dB$. The scale shows values from -36 to +9.

BBC/EBU These meters have an attack time of $10 \pm 2ms$ (for 77.77% target level) and a decay time of 2.8 ± 0.3 seconds in fast mode and 3.8 ± 0.5 for 24 dB. The difference between the BBC and EBU standard resides only in the scale used. The BBC uses a scale from 1 to 7 where 4 equals 0dB_u, 6 being considered the maximum acceptable peak, and EBU uses a scale from -12 to +12.

PPM meters usually report around 4dB higher than VU meters. These meters are useful to make sure equipment doesn't generate any audible distortion, but still have a slow decay to approximate perceived loudness. They are widely used (specially in Europe) for broadcast delivery standards. PPM meters are technical tools and don't seek to measure loudness. VU meters are typically better loudness meters.

3.3 Sample Peak Meters

Digital audio equipment and software typically implement sample peak meters which show the maximum sample received. This type of meter is simple to implement as you only need to check if the new sample is greater than the current meter position and if not, divide the meter value by some factor to generate a smooth movement. This "fall-off" improves the readability of the meter (otherwise very short peaks might be too short for the eye), while making sure all the sample peaks are reported.

These meters can miss inter-sample peaks that may appear on the digital-to-analog conversion. Interpolated upsampling can increase the precision of this meters [8].

[1] DIN 45406 specifies another type of fast PPM metering.

3.4 Equivalent Continuous Sound Level Measurement

Some studies [9] have identified Equivalent Continuous Sound Level measurements (implemented mostly on sound level meters rather than audio equipment) as the most accurate method to determine perceived loudness for long term measurements[2]. Equivalent Continuous Sound Level has been defined as [10]:

$$L_{A_{eq}T} = 20log_{10}\left[\frac{\sqrt{(1/T)\int_{t-T}^{t} p_A^2(\xi)d\xi}}{p_0}\right] \quad (8)$$

Equal Loudness is the value in decibels of the RMS value for a time period T of an A-weighted (see section 4) audio signal with pressure p, referenced to pressure p_0 [3]. This can be expressed for N number of samples in a digital system as:

$$L_{A_{eq}T} = 20log_{10}\left[\frac{\sqrt{(1/N)\sum_{k=1}^{n} x_k^2}}{p_0}\right] \quad (9)$$

$L_{A_{eq}T}$ measurement with a few additions has been adopted by Dolby for their flagship broadcast meter, the LM100 [11]. This device is slowly gaining ground (at least in the USA) and becoming a standard for audio delivery requirements [4].

3.5 Other loudness calculation techniques

Different manufacturers and standards institutes have developed other loudness measurement techniques. An interesting example is the Zwicker Loudness model (DIN 45631/ISO 532B), which performs separate measurements for different spectral bands. Also noteworthy are the LARM and HEIMDAL algorithms from TC electronics [12], which performed exceptionally well in two separate studies. Neither of these has achieved widespread usage in broadcast or music production.

4 Weighting

To compensate for the variation in detected loudness with respect to frequency and pressure (See section 1.3), several "weighting" networks have been standardized. Weighting networks are filter networks that approximate ear response characteristics in a simple and efficient way. There are several filter networks in use and study today in audio loudness metering:

A, B and C Weighting These weightings, designed to be used for low, medium and high pressure levels respectively, implement low pass and high pass filters, leaving a flat middle section [7]. They are frequently found on SPL meters and A-weighting is the basis for Equal Loudness level calculations ($L_{A_{eq}T}$).

M Weighting This weighing, detailed in ITU-R 468 (previously CCIR 468), has been dubbed "M" for Movie, since it has been used to compare loudness between different sections of film, or with movie trailers.

RLB and R2LB Weighting The Revised Low-frequency B-weightings [13] have the closest approximation to subjective loudness among the weightings according to some studies [12]. This weighting has been proposed in ITU-R BS.1770 as the basis of an equal loudness measurement ($L_{eq}(RLB)$).

There's still no concensus on the most appropriate method or weighting to evaluate loudness. Some techniques appear better than other for certain material or listening conditions, and experimental data is sometimes conflicting.

5 Software Metering

Metering, though traditionally implemented in hardware devices, has seen PC software counterparts. Apart from meters available within audio applications, there are many software packages dedicated to level and loudness metering, however few of them are open source, even though some are freeware. Worthy of notice are:

SpectraFoo This is one of the most complete metering tools available. It is a standalone application or TDM/MAS plugin for ProTools TDM and Digital Performer. Apart

[2] The study has been contested by both TC Electronics and Dolby Labs according to the article "Real-time loudness control for broadcast" by Thomas Lund, found at: http://www.broadcastpapers.com/whitepapers/Realtime-Loudness-Control-For-Broadcast.cfm?objid=32&pid=35&fromCategory=26.

[3] ξ is a dummy variable of time integration

[4] $L_{A_{eq}T}$ measurements are suggested as the way to set the *dialnorm* parameter in ATSC Digital Television Standard A/53 Revision E. The DVB Project has not set any audio standards yet, but also uses AC-3 for audio encoding.

from a complete array of metering tools, with logging, it includes spectral, phase and other types of analysis. It can work in real time or from a file. Only for Mac OS [14].

Signal Tools A ProTools plugin capable of long-term $L_{A_{eq}T}$ measurements on the TDM version [15].

Pinguin Audio Meter Pro Only for Windows, provides K-system and PPM metering [16].

6 Metering on Linux

Audio software typically provides some form of sample peak metering. Most Linux programs follow this practice. Multitrack environments like Ardour and Rosegarden, and audio editors like Rezound and Audacity provide the usual peak meter strips for each channel. Other software like Pure Data, Csound or Ecasound can provide text information about sample peaks. Currently the most advanced option for audio level metering on Linux is Steve Harris' Meterbridge package [17]. This package contains sample peak, PPM and VU metering (apart from other stereo metering tools). It is a simple but effective and visually appealing package. However, it is designed to be used as a real-time visual aid only.

6.1 What's missing

The available metering options are probably adequate for most music production projects, but for serious post production work, particularly with the advent of digital television standards, and for audio quality control and analysis it becomes important to have other standard compliant metering options like $L_{A_{eq}T}$ and $L_{eq}(RLB)$, and also options for logging and for calculating long-term loudness.

Logging in this context can take two main forms:

- Graphical or text log of the measured levels for regular time periods
- Histogram of density of occurrence of levels

Implementing loudness measuring algorithms as a library might prove useful for other programs like media players, to help achieve good quality automatic loudness matching. Other operating systems like Windows Vista have implemented similar schemes [18], but no technical details are available, though it is likely, since there is no mention of patented technology, that some known algorithm is used.

7 PostQC

A graphical metering tool called PostQC is currently under development by the author, which will support many of the standard metering options mentioned in this article, and will implement needed logging and long-term measurements. It is still in early development, although some features are already working. Figure 1 shows a screenshot of the current state of PostQC. It can be seen that sample peak and $L_{A_{eq}T}$ metering has been implemented and logging can be shown in a somewhat primitive histogram form. Level threshold overshoot is logged showing the channel in which the overshoot occurred, the duration and the amount of overshoot. Most of the work has been done and tested for file input, but the jack real-time portion is almost ready, as all the jack engine is done, and all that needs to be done is the jack callback function, which is a slight variation to the file block process function. PostQC has been programmed in C++ using QT 3.3 and currently depends on libsndfile 1.0.17 and jack. It is to be publicly available soon under the GPL.

Some of the goals of the project include:

- Real-time (Jack) and Offline Audio File metering
- Support for many types of standard level and loudness metering
- Real-time graphical meters
- Upsampled sample peak metering
- Written report of measured data
- Easy usage and useful in many contexts
- Lashified
- Emulation of LM100 dialogue detection by using a clean dialogue/narration signal as gate trigger to turn on and off long-term averaging of loudness.
- Level threshold overshoot report with occurrence time
- Level histograms
- Graphical and text level and loudness logs
- Make sure all meters adhere to standards

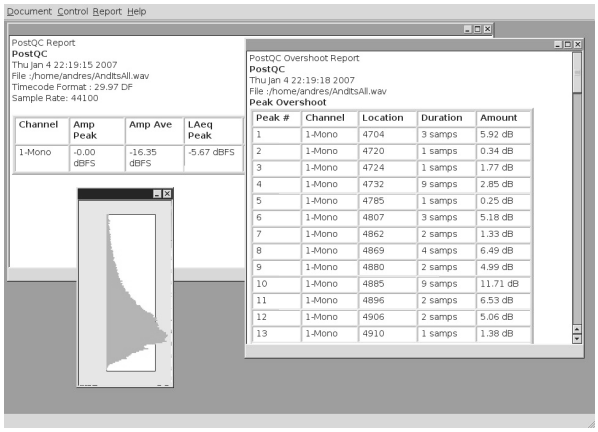


Figure 1: Screenshot of current development version of PostQC

7.1 Other relevant audio measurements

Other aspects that might be useful to determine audio quality apart from metering, that could be included, include real bit depth metering, phase correlation and distortion (clipping) detection.

8 Conclusions

Level metering is an important part of the technical and subjective production of audio material. Linux is not far behind from other platforms in this respect, but an additional tool like the one proposed will certainly make using Linux for metering very appealing.

References

[1] International Organization for Standardization. *ISO/IEC 226:2003: Acoustics - Normal equal-loudness-level contours*, 2003.

[2] J. Alton Everest. *The Master Handbook of Acoustics, 3rd. Edition*. TAB Books, 1994.

[3] Earl Vickers. Automatic long-term loudness and dynamics matching. In *Proc. of the AES 111th Convention*, 2001.

[4] Ioan Allen. Are movies too loud? www.dolby.com/assets/pdf/tech_library/54_Moviestooloud.pdf.

[5] Bob Katz. Integrated approach to metering, monitoring, and leveling practices, part 1: Two-channel metering. *Journal of AES*, 48(9):800–809, 2000.

[6] International Electrotechnical Commission. *IEC 60268-17: Sound system equipment. Part 17: Standard volume indicators*, 1990.

[7] International Electrotechnical Commission. *IEC 60268-10: Sound system equipment - Part 10: Peak programme level meters*, 1991.

[8] Audio Engineering Society. *Standards project report - Considerations for accurate peak metering of digital audio signals. AES-R7-2006*, 2006.

[9] International Telecommunications Union. *Audio metering characteristics suitable for use in digital sound production.Recommendation SG06.2. SRG-3 Status Report (2). Document 6P/145-E*, 2002.

[10] International Electrotechnical Commission. *IEC 60268-17: Electroacoustics - Sound Level Meters - Part 1: Specifications*, 2003.

[11] Dolby laboratories, inc. lm100 broadcast level meter. www.dolby.com/professional/pro_audio_engineering/lm100_01.html.

[12] E. Skovenborg and S. H. Nielsen. Evaluation of different loudness models with music and speech material. In *Proc. of the AES 117th Convention*, 2004.

[13] G.A. Soulodre and S.G. Norcross. Objective measures of loudness. In *Proc. of the AES 115th Convention*, 2003.

[14] Metric Halo. Spectrafoo complete. www.mhlabs.com/metric_halo/products/foo/.

[15] Digidesign. Signal tools. www.digidesign.com.

[16] Pinguin audio meter - pro. http://www.masterpinguin.de/.

[17] Steve Harris. Meterbridge. www.plugin.co.uk/meterbridge.

[18] Nick White. Audio innovations in windows vista. windowsvistablog.com/blogs/windowsvista/articles/450038.aspx.

Renewed architecture of the *sWONDER* software for Wave Field Synthesis on large scale systems

Marije A.J. Baalman and **Torben Hohn** and **Simon Schampijer** and **Thilo Koch**
Institute for Audio Communication (Sekr. EN8)
Einsteinufer 17
10587 Berlin
Germany
baalman@kgw.tu-berlin.de and torbenh@gmx.de and simon@schampijer.de and tiko@admin-box.com

Abstract

For large Wave Field Synthesis (WFS) systems multiple computers are needed for rendering to manage the necessary amount of audio channels. To make this possible with the *sWONDER* software, the software was completely restructured and divided into several separate programs which can run on multiple computers, communicating with each other via OpenSoundControl. This paper describes the new structure of the program, as well as several implementation details of the scheduling unit and audio rendering unit.

Keywords

Auralisation, Wave Field Synthesis, Convolution

1 Introduction

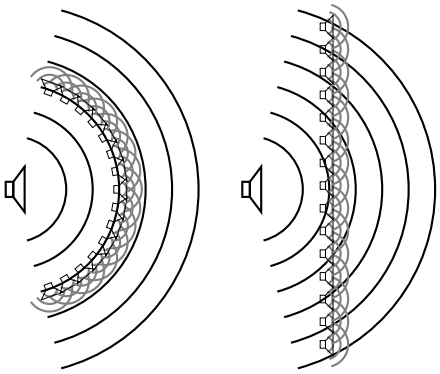
Wave Field Synthesis (WFS) is a method for sound spatialisation. Its main advantage is that it has no sweet spot, but instead a large listening area, making the technology attractive for concert situations.

The main principle of WFS is illustrated in figure 1. A wave field can be synthesized by a superposition of wave fields caused by a lot of small secondary sources, provided you calculate the right delays and amplitude factors for the source signal for each secondary source.

For large WFS systems the calculation of the audio signals for each loudspeaker cannot be done on just one computer, due to limitations of the CPU-power and hardware considerations, such as the number of output channels. Thus, a cluster of computers is necessary, and there is a need to synchronise these calculations. The previous versions of *sWONDER* [1; 2] were monolithic programs, which did not provide this option. This paper describes a new structure for the *sWONDER* program, which enables the software to control large scale WFS systems.

2 Hardware setup

In 2006/2007, the TU Berlin launched a project to equip one of the lecture halls with a large



(a) The Huygens' Principle (b) Wave Field Synthesis

Figure 1: From the Huygen's Principle to Wave Field Synthesis

WFS system[3; 4], of in total 840 loudspeaker channels, both for sound reinforcement during the regular lectures, as well as to have a large scale WFS system for both scientific and artistic research purposes. The loudspeakers are built into loudspeaker panels[5], each providing 8 audio channels, which are fed with an ADAT signal. Each panel additionally has 2 larger speakers which emit the low-pass filtered sum of the 4 channels above it.

To drive these speakers a cluster of 15 Linux computers is used. Each computer computes the loudspeaker signals for 56 loudspeaker channels. Each computer is equipped with an RME HDSP MADI[6] sound card. Each MADI output is connected to an MADI to ADAT bridge (RME ADI648[6]), which is mounted inside the wall, so that the ADAT cables can be kept short (up to 10 meters). The input to the system is multiplexed to each MADI sound card with the use of MADI bridges (RME MADI Bridge[6]).

The cluster has two networks, one for the OSC[7] communication, and one for data-transfer. Separating these network functions,

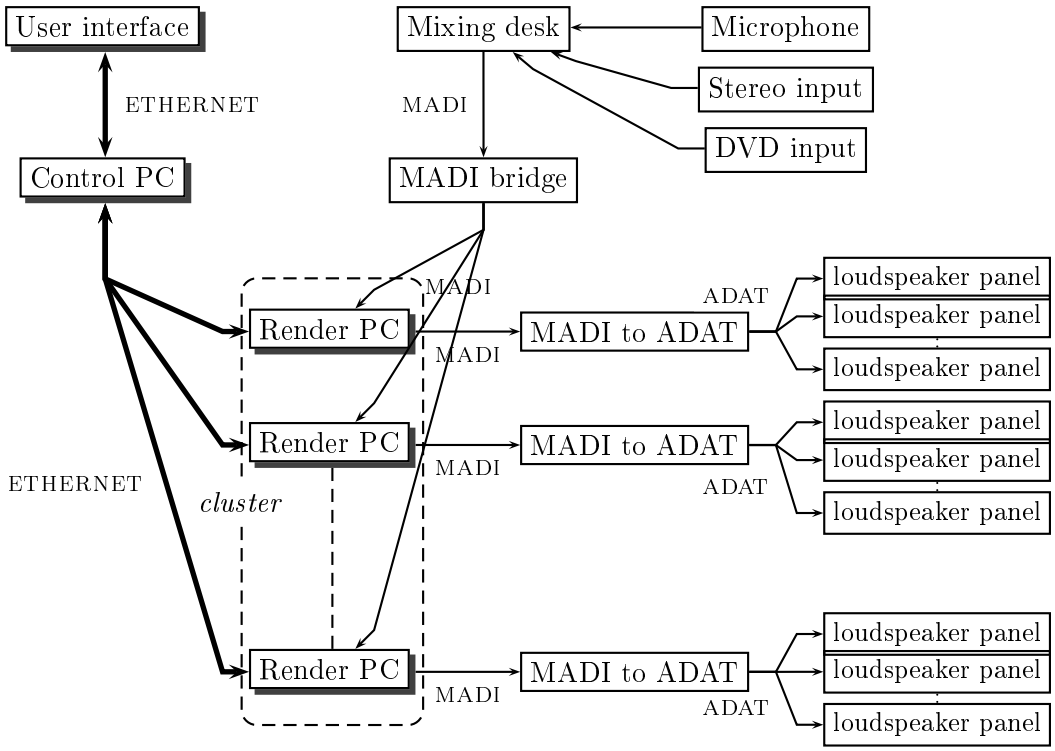


Figure 2: Schematic overview of the hardware setup for the WFS system in the lecture hall of the TU Berlin.

ensures that the OSC communication is fast. The master machine (Control PC) acts as a bridge to the outside world and is the only computer that is connected to an external network.

A general overview of the hardware setup is given in figure 2.

The *sWONDER* software was adapted to control this system, in such a way, that it can also be used by similar but not necessarily identical systems.

3 Software architecture

The software is divided in several parts:

- a graphical user interface,
- a score player/recorder,
- a control unit,
- a real-time render unit,
- an offline render unit
- and a common library for general functions.

Communication between the different parts of the program is based on the OSC protocol[7]. Figure 3 gives an overview of the program parts and their communication.

3.1 Graphical User Interface

The graphical user interface (GUI) provides dialogs for loudspeaker array configuration, grid point configuration (possible source positions and their characteristics), composition and a real time control interface. In the real time control interface, it is possible to move sources around with the mouse, as well as to store different scenes, between which can be switched. The GUI is currently still in development, and will be based on the current GUI [1; 2]. It will be ported to *Qt4* [8], its usability will be improved, and we are working on ways to visualise the timeline of two-dimensional movement.

3.2 Score player/recorder

The system can take any kind of audio input, so that the user can use the audio player (s)he prefers to play the audio. The score player/recorder is used to synchronise with an audio player and record and playback source movements. Synchronisation is based on MTC (Midi Time Code), as this is a clock format which many DAW's support.

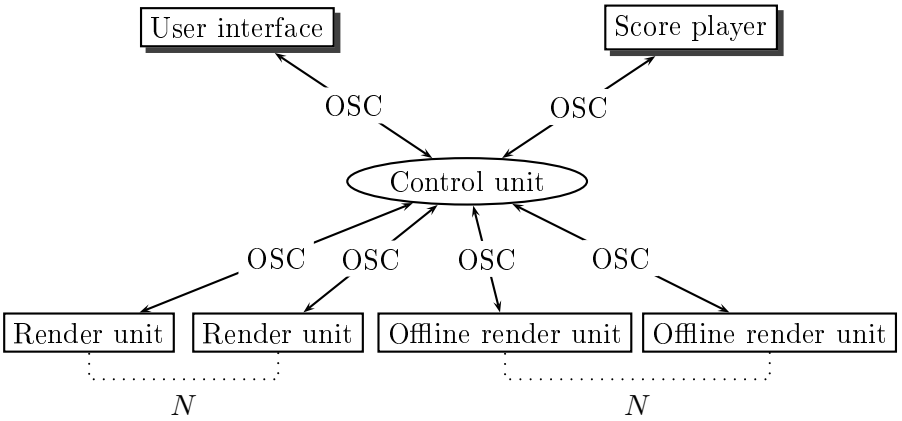


Figure 3: Schematic overview of the different parts of the software *sWONDER*. The control unit can communicate with an arbitrary number (N) of realtime and offline renderers.

3.3 Offline renderer

For room simulations or for complex sound sources [9], the calculations for the impulse responses for each speaker can take quite long, and cannot be performed in realtime. For this purpose, there will be an offline render unit, which takes care of all these calculations, utilising the benefit from parallel execution on a cluster.

3.4 Control unit

The control unit acts as a bridge between the user interface and the audio renderers; it also communicates with the score player/recorder. Though the *sWONDER* suite of programs will also supply a graphical user interface, any other program that can send (and receive) OSC can be used to control the system. The user interface only needs to communicate with the control unit, and does not need to know anything about the audio rendering details; the control unit takes care of that.

3.5 Rendering engine

The real-time render unit is responsible for the actual audio signal processing. It has several ways to deal with the audio streams: playback of direct sound, utilising weighted delay lines, convolution of the input sound for early reflections, and convolution of the input sound for reverb followed by a weighted delay lines to create plane waves with the reverb tail. Schematically this is shown in figure 4.

The rendering engine consists of two parts: *twonder* for the delay line implementation, and *fwonder* for the convolution. Both programs are controlled by OSC; audio input and output has JACK as the audio backend.

4 Direct sound

4.1 Delay lines

The direct sound of a WFS synthesized source, consists of the delayed and attenuated source signal. This delay and attenuation is unique for each speaker. The direct sound of the source is rendered in the time-domain by the *twonder* part of the program.

To initialise the delay lines, the length of the delay lines need to be determined. The length is related to the largest distance a source will have to a speaker. Also, it needs to be decided how far in front of the speakers we want to move a source, as this determines the needed delay offset. If no focused sources are needed, we can set the delay offset to a smaller number, thus introducing less latency in the system. These options can be set per source.

4.2 Moving sources

When a source moves, the delay time will change continuously, as well as the volume factor. In *twonder* the delay time for the start and the end of the block is calculated (thus these are a kind of anchor points), and the samples inside the block are resampled. This is clarified in figure 5. If the delay time is 20 samples at the start of the block, and 30 samples at the end of the block, we need to output 10 samples less than the actual block size N. Thus, we need to resample $N - 10$ to N samples. Because of the CPU restraints (we need to do this for a lot of delaylines in realtime), we need an efficient resampling algorithm. We chose linear interpolated resampling. The implementation is a modified version of Bresenham's line drawing

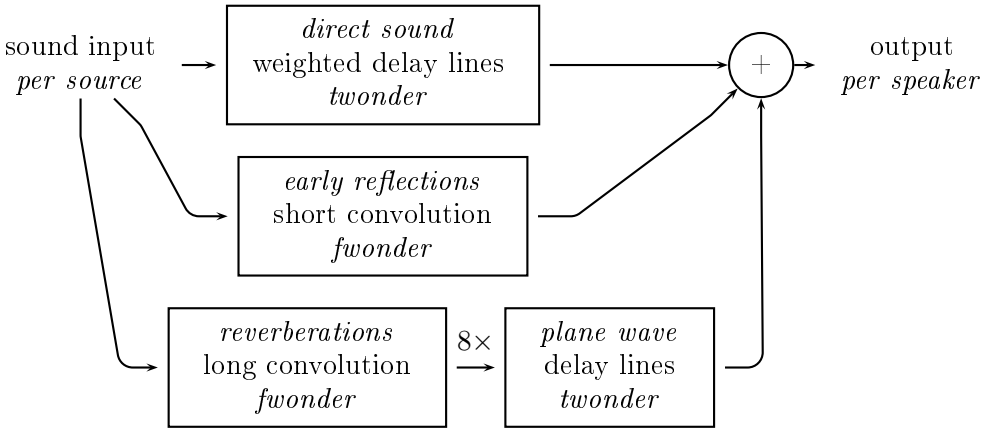


Figure 4: An overview of the audio signal processing by the real-time render unit

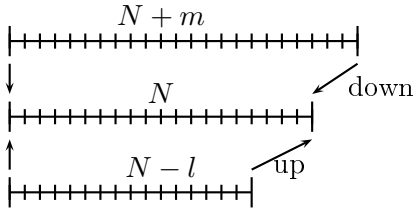


Figure 5: Illustration of the resampling problem: if the delay time gets longer within a certain block, we need to output more samples than we have available in our buffer. Thus, we need to upsample the avaible samples. If the delay time gets shorter, we need to output less samples, than we have available in our buffer and we need to downsample them.

algorithm [10], which eliminates the need to cast a float to an integer in the innner loop.

Moving a source in this way, creates a Doppler effect, which will be audible if the movement is very fast. In some cases it is not desired to hear a Doppler effect, so another option for movement is provided, which we have called a *fade jump*. Using this option, the illusion of movement is created by fading the source out on one position, while fading it in on the next position. The update frequency for this can be set by the user.

4.3 Plane waves

Plane waves are achieved by just varying the delay times for each speaker, based on the angle the wave front makes with the speaker array. A delay offset is created by giving the plane wave a point of origin in space, in addition to its direction. This approach also makes it possible to switch from a point source to a plane wave and vice versa.

Plane waves can be used to simulate sources that are very far away and only have a direction, or to simulate reflections, as will described in the next section.

5 Room simulation

Room simulation is achieved by adding reflections to the direct sound. This can be achieved in several ways: (1) inclusion of a first reflection in the delay line, (2) doing a short convolution for early reflections for each speaker with offline calculated impulse responses (IRs) and (3) doing a convolution with a longer impulse response, the result of which will be played back using plane waves.

The first option is in development. In this option also a filter on the reflected sound can be included, provided the filter can be created with ca. 8 FIR taps.

The second and third option are possible already, though the offline renderer to calculate the early reflection impulse responses is not ready yet. Alternately, other methods could be used to calculate the early reflection IRs, such as an old version of *sWONDER*, or using an approach based on measurements such as described in [11; 12; 13]

Ad 2: The impulse responses are unique to each source position and speaker. Thus for each speaker a convolution needs to be made. This option is CPU-intensive, and requires all of the impulse responses to be loaded into memory. In [14] research is presented from which can be concluded how closely gridpoints need to be spaced to ensure perceptual consistency of the wave field, for a specific setup (depending on the dimensions of the virtual room, as well as the size

of the desired listening area).

Ad 3: research at the TU Delft has proved that using 8 plane waves (at 45 degrees interval directions) is sufficient to create a realistic reverberation[15].

5.1 Convolution

The *fwonder* program implements a fast convolution from multiple inputs to (even more) multiple outputs. It uses the same complex multiplication method as BruteFIR [16]. Instead of extending BruteFIR we rewrote a convolution engine from scratch, because this was considered faster than extending BruteFIR, due to the lack of transparency and documentation of the BruteFIR code. The other available solutions were not written in C++ or tied to SuperCollider [17; 18], which would have slowed down development also. So we decided to reimplement the algorithm, while learning from the others.

5.2 IR caching

When a source is moving, we need to change the impulse responses being used. Because the set of impulse responses does not fit into memory, a cache structure needs to manage the loading of the impulse responses from disk.

This problem is solved as follows: when the position of a source changes the UI sends the absolute position in meter to the control unit. The control unit sends the new position to *twonder*, and simultaneously calculates the corresponding (closest) grid position for which an early reflection impulse response is available, and sends this information to the render unit. The render unit then switches the impulse responses used in the convolution to the new ones. Crossfading is used to reduce the artefacts of this process.

Because the loading of new impulse responses is a task that takes some time to complete, it should happen before an event actually occurs if possible. In real-time mode we do not know in advance what parameters of which source will change next. As a solution the grid of points for which IRs are calculated is divided in anchor points and normal points. Anchor points are points whose IRs are always stored in memory. When a source moves to a new location, first the IR of the anchor point is used, and then the surrounding points are loaded into memory, so that changes to locations nearby can be made in real-time (see figure 6). When there is a score, we do know which IRs are needed in the future, and we can determine the needed IRs in time, as shown in figure 7.

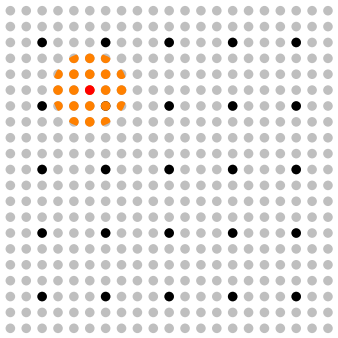


Figure 6: Loading grid point impulse responses into cache. The black points are the anchor points and correspond to impulse responses that are always loaded in memory. The red (darkest grey) point indicates the grid point used for the current position, the orange (grey) points the one for which the IRs are currently loaded in memory. The light grey points are the available points.

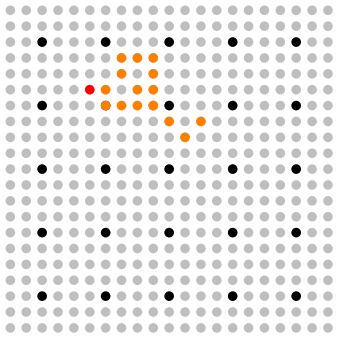


Figure 7: Loading grid point impulse responses into cache while playing a score. As we know the future locations of the source, we can preload the IRs that correspond precisely to the sound path.

The control unit takes care of this scheduling of loading and unloading of IRs and sends commands to the render units to perform this (i.e. the render unit is 'stupid' and just follows the orders of the control unit).

5.3 Calculating the IRs

The IRs as described above, will need to be calculated beforehand with the offline renderer. This is handled as follows: in the UI the user

defines a grid of points and virtual room dimensions. Then he sends a message to the control unit to start the calculation. The control unit then communicates with all the offline renderers that are running, to perform this task, and sends a message back to the UI when the task is completed. Then the calculated IRs can be used in realtime.

6 Time and synchronisation

There are several concepts of time within the system: the user interface can send messages, which have to be executed *now* and have a certain *duration*; or it can send messages which have to be executed at a certain time from *now* and have a certain *duration*.

The score player/recorder has to deal with both MTC and synchronise itself to that clock, as well as communicate to the control unit, just like other user interfaces.

All communication from the user interface to the control unit about time, is in seconds. As the renderers need to be synchronised with sample accuracy, the control unit translates the time in seconds to frame time. The audio clock is used as the time reference. This clock is reliable, has got the desired granularity and is present on each render unit and the control unit. The audio devices in the units are fed with a MADI signal including a word clock signal. Because the audio links are digital, a sawtooth generated at the control node, will be sufficient to extract the initial synchronisation position from the audio signal. When initial synchronisation is done, sync will be maintained by the word clock sync.

This leads to a system with one central clock and avoids the need for clock skew compensation which is needed when having multiple clocks.

As an example we consider the task of changing the position of a source. This information is sent from the UI to the control unit where a timestamp for this event is generated. Since the control unit has the information about the actual time in samples the messages will be stamped with this time reference and send to the render unit.

Both the control unit and the render unit can deal with interpolation over time, i.e. it is possible to send the control unit a message to move a source from one position to another with a certain duration of the movement. The control unit will pass on this duration to *twonder*, which then interpolates the movement and calculates the positions (and thus the delays) at the end of each block, and creates the movement. The control unit will also calculate the intermediate positions on the grid, and ensure that the IRs for the intermediate points are preloaded by *fwonder* and the IRs needed for the current position are switched to in time.

7 File formats

For configuration of the system and creating a project with the system, several files are needed to store the relevant data.

It was chosen to use XML for the format for storing this data, as it is easily extendible in case of need.

There are files for:

Configuration This contains the data about the rendering units: the network setup and the speaker setup.

Project This contains the general settings for a project, such as how many sources are used and the characteristics of each sources. It can also contain a score, and settings for different scenes (static constellations of sources, between which the user can switch).

Grid This contains the information about the grid points used for early reflection calculation, as well as information about the impulse responses (path and format in which they are stored).

As a basis for the project file format we used the XML-format for 3D audio as described in [19]. Currently we are undertaking efforts to start a discussion with other institutes that work on Wave Field Synthesis to agree upon a common XML-format to be able to exchange content.

8 Working OSC commands

In table 1 an overview is given of the currently working OSC commands.

8.1 Project

To be able to store scenes, you need to create a new project with the command: `/WONDER/project/create`, with one *string* as argument: the project name.

You can save the project with the command: `/WONDER/project/save`, and later load it again with the command `/WONDER/project/load`.

command	types	arguments
/WONDER/project/create	s	projectname
/WONDER/project/load	s	projectname
/WONDER/project/save	s	projectname
/WONDER/scene/add	i	scene no.
/WONDER/scene/select	iff	scene no., time, duration
/WONDER/scene/remove	i	scene no.
/WONDER/scene/set	i	scene no.
/WONDER/source/position	iffff	srcid, pos x, pos y, pos z, time, duration
/WONDER/source/angle	ifff	srcid, angle, time, duration
/WONDER/source/type	iiff	srcid, type, angle, time

Table 1: Working OSC commands

8.2 Scenes

You can create a snapshot of the current source positions (called a "scene") and store them in the project, using the command `/WONDER/scene/add` with an integer as argument for the slot number under which you want to store the scene.

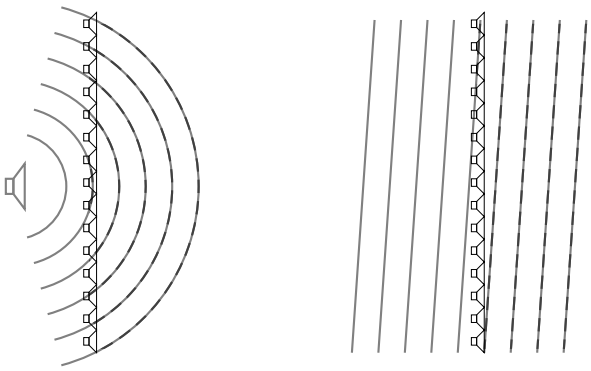
Later you can recall the scene with the command `/WONDER/scene/select`, with as arguments the scene number, the time at which the change to the scene should start, and the duration in which it should fade to the new scene.

With `/WONDER/scene/remove` a scene is deleted (and thus the slot is freed again). With `/WONDER/scene/set` you can overwrite an existing scene. Note the subtle difference between adding a scene and setting a scene: adding creates a new scene and stores the current source positions to it. It gives an error back when the scene number already exists. "Set" stores the current source positions to an existing scene and gives an error back if the scene slot does not exist.

8.3 Source control

There are two types of sources: point source (see fig. 8a) and plane wave (see fig. 8b).

With the command: `/WONDER/source/type` you can set the type for one source. Plane wave is "0", point source is "1". The angle argument is the start angle for the plane wave. Whenever the type is changed you should also send a `/WONDER/source/position` command, to set the position of the source. In the case of a point source, this will be the actual position of the source. In the case of a plane wave, this is a reference point for the calculation; it should be chosen to be a position somewhere behind the array in the direction where the plane wave is coming from. This point determines the basic latency of the plane wave.



(a) Point source (b) Plane wave

Figure 8: Source types

`/WONDER/source/position` takes as arguments the source id, the x and y position (in meters), the z position (which should be 1.0 for now), the time at which the change should start (in seconds from "now"), and the duration for the change to take place (also in seconds).

`/WONDER/source/angle` takes as arguments the source id, the angle, the time at which the change should start, and the duration for the change to take place.

9 Conclusions

We have presented the new architecture of the *sWONDER* software, with a focus on the central control unit and the audio rendering unit. The user interface of *sWONDER*, a score player and offline render unit are in development, to provide a full suite of open source tools for doing WFS.

Parts of the software may also be useful for other purposes, such as the OSC-controllable de-

laylines and convolution engine.

We plan to extend the software with options for other spatial reproduction techniques, such as binaural headphone reproduction and ambisonics.

10 Acknowledgements

Our thanks go to the Bauabteilung of the TU Berlin for funding, and especially to Christoph Moldrzyk for initiating the project.

The software is released under the GPL license at `http://swonder.sourceforge.net`.

References

[1] M.A.J. Baalman. Application of wave field synthesis in electronic music and sound installations. In *2nd International Linux Audio Conference, 29 april - 2 Mai 2004, ZKM, Karlsruhe*, 2004.

[2] M.A.J. Baalman. Updates of the wonder software interface for using wave field synthesis. In *3rd International Linux Audio Conference, April 21-24, 2005, ZKM, Karlsruhe*, 2005.

[3] C. Moldrzyk, A. Goertz, M. Makarski, W. Ahnert, S. Feistel, and S. Weinzierl. Wellenfeldsynthese für einen großen hörsaal. In *DAGA 2007, Stuttgart, Germany*, 2007.

[4] T. Behrens, W. Ahnert, and C. Moldrzyk. Raumakustische konzeption von wiedergaberäumen für wellenfeldsynthese am beispiel eines hörsaals der tu berlin. In *DAGA 2007, Stuttgart, Germany*, 2007.

[5] A. Goertz, M. Makarski, C. Moldrzyk, and S. Weinzierl. Entwicklung eines achtkanaligen lautsprechermoduls für die wellenfeldsynthese. In *DAGA 2007, Stuttgart, Germany*, 2007.

[6] Rme - intelligent audio solutions. `http://www.rme-audio.com/`.

[7] M. Wright, A. Freed, and A. Momeni. Opensoundcontrol: State of the art 2003. In *2003 International Conference on New Interfaces for Musical Expression, McGill University, Montreal, Canada 22-24 May 2003, Proceedings*, pages 153–160, 2003.

[8] Trolltech. Qt library. `http://www.trolltech.com/products/qt/index.html`, 1996-2005.

[9] M.A.J. Baalman. swonder3dq: Auralisation of 3d objects with wave field synthesis. In *4th International Linux Audio Conference, April 27-30, 2006, ZKM, Karlsruhe*, 2006.

[10] Wikipedia. Bresenhams line algorithm. `http://en.wikipedia.org/wiki/Bresenham's_line_algorithm`, 2007.

[11] Frank Melchior and Diemer de Vries. Detection and visualization of early reflections for wave field synthesis sound design applications. In *Tonmeistertagung 2006, Leipzip, Germany*, November 16-19 2006.

[12] Diemer de Vries, Jan Langhammer, and Frank Melchior. A new approach for direct interaction with graphical representations of room impulse responses for the use in wave field synthesis reproduction. In *120th AES Convention, Paris, Preprint 6657*, May 2006.

[13] Edo Hulsebos and Diemer de Vries. Spatial decomposition and data reduction of sound fields measured using microphone array technology. In *17th ICA, Rome, 2001*, 2001.

[14] Hiske Helleman. Sensitivity of the human auditory system to spatial variations in single early reflections. Master's thesis, Delft University of Technology, The Netherlands, 2003.

[15] J.-J.Sonke and D. de Vries. Generation of diffuse reverberation by plane wave synthesis. In *102nd AES Convention, March 1997, Preprint 4455*, 1997.

[16] A. Torger. Brutefir. `http://www.ludd.luth.se/~torger/brutefir.html`, 2001-2005.

[17] J. McCartney. Supercollider. `http://www.audiosynth.com`.

[18] Stefan Kersten. A fast convolution engine for the virtual electronic poem project. Master's thesis, Technische Universität Berlin, 2006.

[19] Guillaume Potard. *3D-Audio Object Oriented Coding*. PhD thesis, University of Wollongong, Australia, September 2006.

Offener Schaltkreis. An Interactive Sound Installation

Christoph HAAG
Martin RUMORI
Franziska WINDISCH
Ludwig ZELLER
Klanglabor, Academy of Media Arts Cologne (KHM)
Peter-Welter-Platz 2
50676 Cologne,
Germany,
osk@khm-lists.rumori.de

Abstract

Offener Schaltkreis (Open Circuit) [1] is an interactive sound installation developed by students at the Academy of Media Arts, Cologne. It mainly focuses on openness, which applies to all facets such as optical appearance, the interface given to the user, the technical tools being used and the collaborative style in which the installation has been developed. In this paper, we will discuss the aesthetical and technical issues of our sound installation *Offener Schaltkreis*.

Keywords

sound installation, interaction, interface design, tangible, *pd*

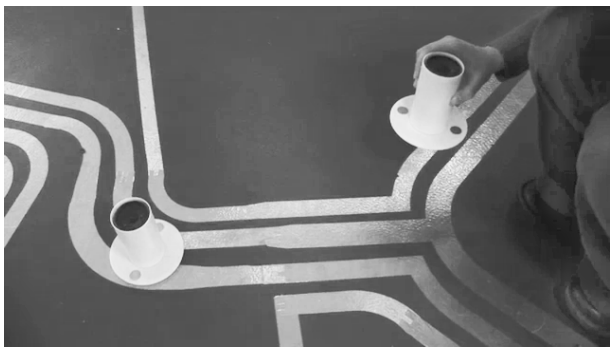


Figure 1: user interacting with *Offener Schaltkreis*

1 Introduction

"At the Academy of Media Arts, art, technology, and science work together for mutual enhancement. Different ways of thinking meet: theory encounters practical design, technological programmes and artistic imagination combine."[1]

The interactive sound installation *Offener Schaltkreis* is actually a result of such a meeting of two different programmes within the KHM: *Hybrid Space* and *OSFA*.

[1]from the information flyer KHM 2006/2007

The *Hybrid Space* was founded by Prof. Frans Vogelaar within the media design department. "A new interdisciplinary field of design, researching the transformations of architectural, urban/regional space of the emerging 'information age', explores the dynamic interaction of architecture/urbanism and the space of mass media and communication networks. It develops scenarios for the interplay of public urban and public media space." [2]

The *OSFA* series of workshops introduced by Martin Rumori is connected to Klanglabor within the department of arts and media studies. *OSFA* translates to "open source for arts" and emphasizes on all aspects for using open source technology for artistic purposes.

Offener Schaltkreis is experienced by putting freely placeable speaker-cylinders on a labyrinth created out of open copper tracks. Since these tracks carry electrical audio signals, corresponding sound layers become audible. This simple principle of operation is derived from another project, created at about the same time at KHM: *Talking Cities Radio* [3]. While *Talking Cities Radio* is an interface for the audible content of an exhibition, *Offener Schaltkreis* aims at being a sound installation whose aesthetical criteria were entirely developed collaboratively from scratch.

2 Aesthetical Approach

2.1 Model of a city

According to Foucault [4], the city is considered as a heterotopia: an agglomeration of diverse spaces, not least on an aural level. The acoustic impressions differ from place to place, just like inhabitants and surroundings are different. Every neighbourhood has its characteristics and on the way through a city, not only the visual, but also the acoustic environment is

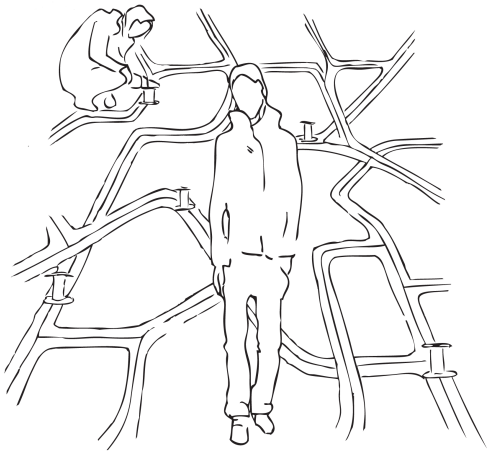


Figure 2: design sketch for *Offener Schaltkreis*

constantly changing. The experience of moving through an urban environment gets downscaled and transferred into a room. Through putting the speaker-cylinder on the tracks a concrete place of sound is created, which uses the copper as a source for its emission and therefore for its sensual manifestation. But this concrete place of sensual manifestation is not connected to a special place on the map. It is not static but in a permanent shift, just like its audio source. Free positioning of the speaker-cylinder allows the visitor to create her own soundscapes. Soundscapes, which open themselves only through time and movement in space.

2.2 Transporting audio

The copper tracks on the ground resemble a map, recalling the transportation networks of a city. This fact made it evident from the beginning, to work with sounds and noises of a city, including their transformations and synthetic imitations. For every copper track an individual piece of sound has been created from collected sound material.

The installation is constantly playing. For every track a virtual read head loops in various speeds through a given sound piece. Every track includes four parts, which create, while playing all tracks together, changing auditory scenes.

2.3 The visitor as an author

A silent labyrinth created out of open copper trails on the floor carries the electrical signals of a multichannel sound repository. By putting freely placeable speaker-cylinders on them, the carried sound layer becomes audible. *Offener Schaltkreis* reacts depending on the manner in which the speaker-cylinders are used: if nothing is moved, the sounds stay calm and soft, but if cylinders are repositioned, the currently played sound material is modulated by increasing speed, pitch and velocity.

If nothing happens during a few minutes, the installation starts cooling down, back to the quietest, lowest level, where it remains sleeping: just like every acoustic space sleeps while nobody is there. As soon as a single cylinder is moved during this state, the sounds of all speakers suddenly jump to a higher level of activity. Thus, they give the impression of a social structure: a sensitive, pulsating, constantly shifting body, built with sound.

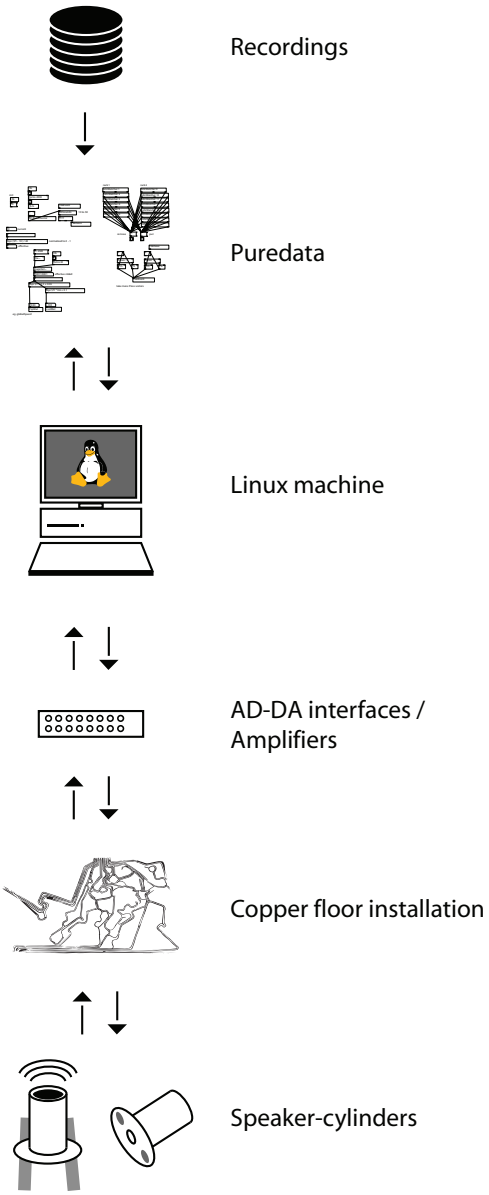


Figure 3: technical flow chart

3 Technical Description

A Linux PC with a multichannel audio interface drives the openly installed copper tracks (mass and signals). The freely movable speaker-cylinders also have open copper contacts at the bottom so that placing them on the tracks connects the amplified audio signals to the speakers.

3.1 Software

The audio recordings described in chapter 2 are played back by a *pd* patch [5]. The patch is freely available at [1]. It reads the materials in progressing loops, playing repeatingly with forward-moving loop markers. That kind of "macro-granular synthesis/sequencing" creates a diffuse sound, that morphs between different areas of the recordings, presenting ever-changing sound facets.

The interactive mapping of the visitor's activity to sound shaping parameters like volume, "loop" length (or "macro grain length") and thus pitch and speed is done directly in each track's player instance.

The more the system is stimulated (up to a certain point), the less the actions of the visitors are taken into account. This kind of damping makes it hard to reach the defined maximum value, allowing for soft boundaries of the dynamic range. The lower end of the dynamic range represents the abovementioned "sleeping". In this state, the least activity on the speaker leads to a sudden high increase.

In order to accomplish the interactive mapping, it is necessary to track the visitors' actions. Therefore discrete "put" and "remove" events are generated based on electrical measurements on the copper tracks.

3.2 Hardware

Besides the actual audio content signals, all tracks carry an inaudible 20 kHz sine signal at constant amplitude. This is used as a reference signal for counting the amount of speakers that are placed properly on the copper tracks. The sum of the audio and the reference signals is fed back into the corresponding `adc~` inputs of *pd* where the 20 kHz sine is isolated.

When no speaker is placed on a specific track, the feedback signal of that track will be at $-\infty$ *db* in amplitude, since the electrical circuit is simply not closed. In the arrangement, we avoided adjacent tracks of the same kind, thus making sure that masses and signals are always alternating. As a result, a proper speaker connection is guaranteed at any arbitrary position of the installation.

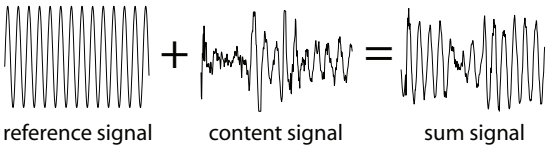


Figure 4: signal composition

With the first speaker placed, the amplitude of the isolated feedback signal jumps to a certain value well above zero. Due to electrical laws, this level U_{sens} converges logarithmically towards the amplitude of the fed-in signal when more speakers are added to the same track, because all speakers will be situated in a parallel connection ($U_{sens} = 1 - \frac{1}{1+n}$).[2] This exponential graph can be transformed in order to get the discrete integer value of placed speakers.

The tracking of the visitors' activity is accomplished by interpreting this number on a per track basis. The change of this quantity can be interpreted either as addition ("put") or removal of speakers. As described above, this information is then used as a parameter for the sound generation within *pd*.

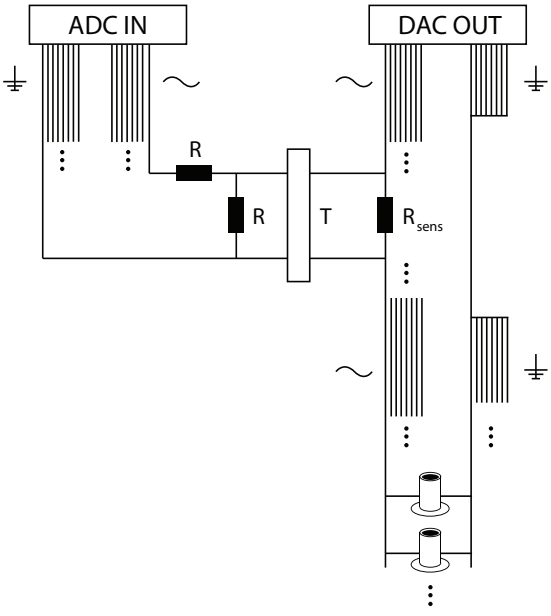


Figure 5: circuit diagram

[2] The more speakers are placed in that parallel connection, the more the equivalent resistance of the floor installation is decreased resulting in an increase of U_{sens} on that voltage divider.

4 Open Source and Open Circuit

Since this work originated out of the seminar series *OSFA workshop series*, the use of open sourced software was indeed one of the few fixed preconditions for developing this project.

The installation tries to empower the visitors or listeners to act on their own. We try to give them the freedom to use the installation in any way they wish. This attempt finds its counterpart in the use of free software, within the empowerment of using and modifying software in any way we choose.

Of course, also the economic aspect of using open source software is important to its use in the context of media art. Free software sometimes simply also means: "free as in beer".

Open Source is generally available on more platforms than proprietary software, since everybody can hack a port to her favorite system, thus complying more easily to her predefined constants.

Works that are developed in an academic context, like the *Offener Schaltkreis*, are also often intended to be presented as a publication that documents more than just the result – but also the path that was taken during the design process. Obviously, presenting code as open source is an almost obligatory requirement for this way of highlighting processes besides the result itself. The open source community provides proven systems for licensing and sharing the author's work.

5 Conclusions

Several artistic and technical aspects of the interactive sound installation *Offener Schaltkreis* have been presented. The initially mentioned openness as a main focus of *Offener Schaltkreis* had the effect of a constantly changing experience of artistic creation. While watching the visitors of the installation it turned out, that further changes to the user interface towards an even more direct feedback might be desirable.

6 Acknowledgements

Our thanks go to Martin Nawrath, technical staff of the KHM, Prof. Anthony Moore, Head of Klanglabor at KHM, Tobias Beck and Michael Thies and the entire *Talking Cities Radio* team.

References

[1] Christoph Haag, Martin Rumori, Franziska Windisch, and Ludwig Zeller. Homepage of Offener Schaltkreis. *http://osk.openkhm.de*, 2006.

[2] Frans Vogelaar et al. Wiki of the Hybrid Space Project. *http://hybridspace.openkhm.de*, 2006.

[3] Frans Vogelaar, Ina Krebs, Christoph Haag, Andreas Muxel, Therese Schuleit, and Isabelle Niehssen. Homepage of Talking Citites Radio. *http://hybridspace.openkhm.de/2006/talkingcities/*, 2006.

[4] Michel Foucault. Of Other Spaces, Heterotopias. *http://www.foucault.info/documents/heteroTopia/foucault.heteroTopia.en.html*, 1967.

[5] Miller Puckette. About Pure Data. The Pure Data Portal. *http://puredata.org*, 2006.

Visual prototyping of audio applications

David Garcia and **Pau Arumi**
IUA, Universitat Pompeu Fabra
Psg. de Circumval.lacio, 8. 08003 Barcelona, Spain
{dgarcia,parumi}@iua.upf.edu

Xavier Amatrinain
CREATE, Univ. of California Santa Barbara
Santa Barbara, CA, USA
xavier@create.ucsb.edu

Abstract

This article describes an architecture that enables visual prototyping of real-time audio applications and plugins. Visual prototyping means that a developer can build a working application, including user interface and processing core, just by assembling elements together and changing their properties in a visual way. The article addresses the problem of having to bind interactive user interface to a real-time processing core, when both are defined dynamically with an extensible set of components, allowing bidirectional communication of arbitrary types of data and still fulfiling real-time requirements of audio applications. It also introduces some design patterns that have enabled its implementation.

Keywords

Audio applications, Visual prototyping, interfaces, GUI, Frameworks

1 Introduction

Having a proper development environment is something that may increase development productivity. Development frameworks offer system models that enables system development dealing with concepts of the target domain. Eventually, they provide visual building tools which also boost the development productivity. In the audio and music domain, the approach of modeling systems using visual data-flow tools has been widely and successfully used in system such as PD [1], Marsyas [2], Open Sound World [3] and CLAM [4]. But, such environments are used to build just processing algorithms, not full applications ready for the public. A full application would need further development work addressing the user interface and the application work-flow.

User interface design is supported by existing toolboxes and visual prototyping tools. They provide a similar flexibility than the one data-flow tools provide to build the processing core. Examples of such environments which are freely available are Qt Designer [5], Fltk Fluid [6] or Gtk's Glade [7]. But such tools just solve the composition of graphical components into a layout and limited reactivity. They still do not address a lot of low level programming that is needed to solve the typical problems that an audio application has. Those problems are mostly related to the communication between the processing core and the user interface.

This article describes an architecture that addresses this gap and enables fully visual building of real-time audio processing applications by combining visual data-flow tools and visual GUI design tools.

Section 2 describes the target applications to be built with the architecture. Section 3 describes the development work flow that the architecture offers and the main tools involved. Section 4 goes deeper on the key part of the architecture, the run-time engine, and describes how it solves different problems it faces. Section 5 describes some audio related design patters that enable the actual implementation of the architecture. Those patterns are part of a bigger pattern catalog which is briefly described on section 6. And, finally, section 7 explains the current status and achievements and the future lines both for the architecture implementation and the pattern catalog.

2 Target applications

The set of applications the architecture is able to visually build includes real-time audio processing applications, which have a relatively simple application logic. That is synthesizers, real-time music analyzers (figure 1) and audio effects and plugins (figure 2).

So application logic should support just starting and stopping the processing algorithm, configuring it, connecting it to the system streams (audio from devices, audio servers, plugin hosts, MIDI, files, OSC...), visualizing the inner processing data and controlling some algorithm parameters while running.

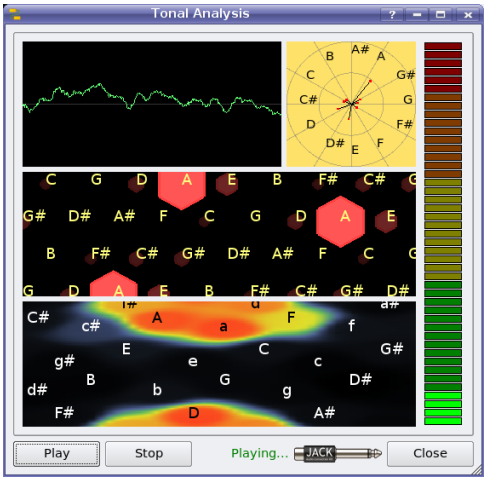


Figure 1: A sample of audio analysis application: Tonal analysis with chord extraction

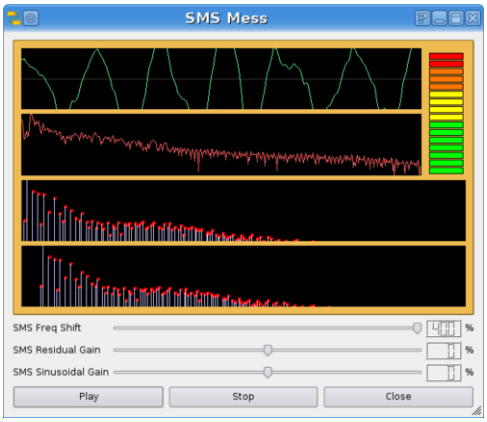


Figure 2: A sample of audio effect application: JACK enabled SMS transposition

Given those limitations, the defined architecture does not claim to visually build every kind of audio application. For example, audio authoring tools, which have a more complex application logic, would be out of the scope, although the architecture would help to build important parts of such applications.

The architecture will provide the following features:

- Communication of any kind of data and control objects between GUI and processing core (not just audio buffers)
- The prototype could be embedded in a wider application with a minimal effort
- Plugin extensibility for processing units, for graphical elements which provide data visualization and control sending, and for system connectivity backends (JACK, ALSA, PORTAUDIO, LADSPA, VST, AudioUnit...)

3 The big picture

The proposed architecture (figure 3) has three main components: A visual tool to define the audio processing core, a visual tool to define the user interface and a third element, the run-time engine, that dynamically builds definitions coming from both tools, relates them and manages the application logic. We implemented this architecture using some existing tools. We are using CLAM NetworkEditor as the audio processing visual builder, and Trolltech's Qt[1] Designer as the user interface definition tool. Both Qt Designer and CLAM NetworkEditor provide similar capabilities in each domain, user interface and audio processing, which can be exploited by the run-time engine.

Qt Designer can be used to define user interfaces by combining several widgets. The set of widget is not limited; developers may define new ones that can be added to the visual tool as plugins. Figure 4 shows a Qt Designer session designing the interface for an audio application, which uses some audio related widgets provided by CLAM as a widgets plugin. Note that other audio related widgets are available on the left panel list.

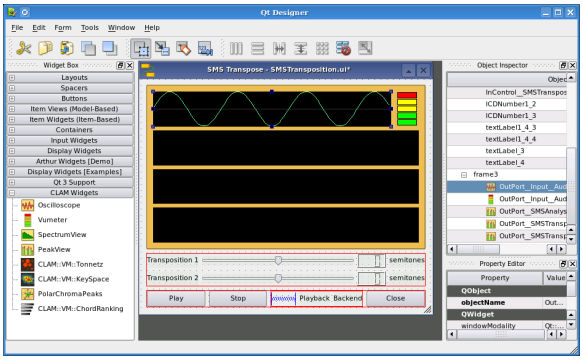


Figure 4: Qt Designer tool editing the interface of an audio application.

Interface definitions are stored as XML files with the ".ui" extension. Ui files can be rendered as source code or directly loaded by the application at run-time. Applications may, also, discover the structure of a run-time instatiated user interface by using introspection capabilities.

[1] We are using Qt version 4.2

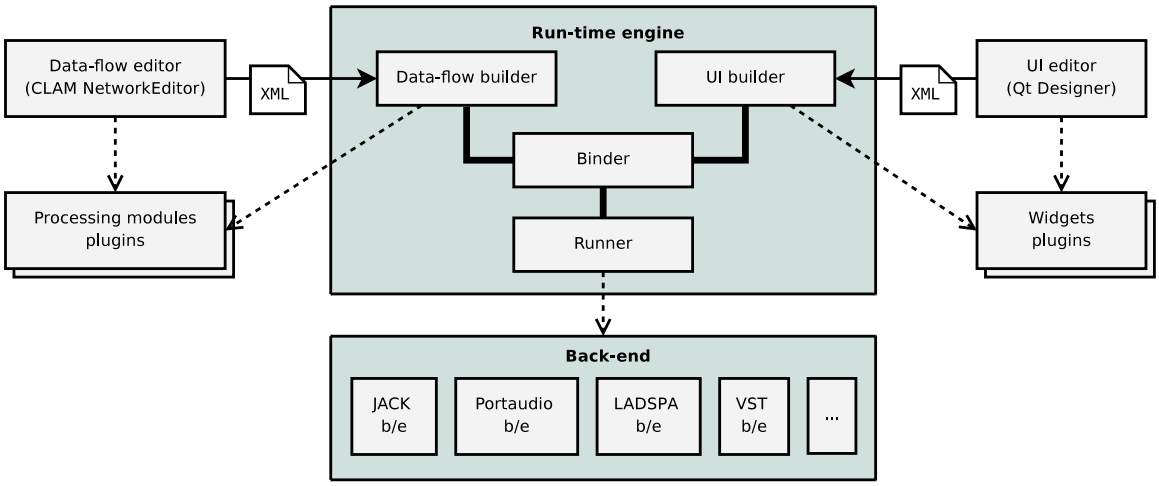


Figure 3: Visual prototyping architecture

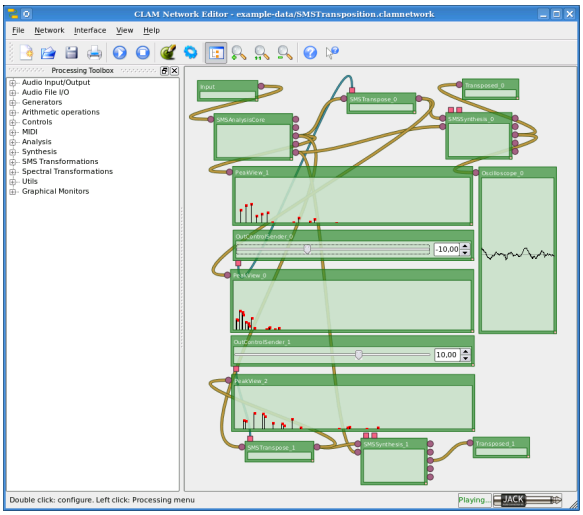


Figure 5: The processing core of an application built with the CLAM Network Editor

Analogously, CLAM Network Editor allows to visually combine several processing modules into a processing network definition. The set of processing modules in the CLAM framework is also extensible with plugin libraries. Processing network definitions can be stored as XML files that can be loaded later by applications in run-time. And, finally the CLAM framework also provides introspection so a loader application may discover the structure of a run-time loaded network.

4 Run-time engine

Just by having a data flow visual tool and a visual interface designer, we should still do some programming to glue it all and launch the application. The purpose of the run-time engine, which is called Prototyper in our implementation, is to figure out which should be that code and supply it. Next, we enumerate the problems that the run-time engine faces and how it solves them.

4.1 Dynamic building

Both component structures, the audio processing network and the user interface, have to be built up dynamically in run-time from an XML definition. CLAM and Qt frameworks supports dynamic building mostly because they provide object factories. Object factories provide objects given a type identifier. Object factories are a very well known general design pattern and it is covered better by E. Gamma et al. [8] and implementation details are further covered by Alexandrescu [9].

Because we want interface and processing components to be expandable, the factories should be able to incorporate new objects defined by plugin libraries. To enable the creation of a certain type of object, the class provider must register it on the factory at plugin initialization.

4.2 Relating objects

The run-time engine must relate components of both structures. For example, the spectrum view on the SMS Transposition application (second panel on figure 2) needs to periodically access spectrum data flowing by a given port of the processing network. The run-time engine firstly has to identify which components, according to the developer's intent. Then guess

whether the connection is able to be done: spectrum data can not be viewed by an spectral peaks view. And then, perform the connection, all that without the run-time engine knowing nothing about spectra and spectral peaks.

The proposed architecture uses properties such the component name to relate components on each side. Then components are located by using introspection capabilities on each side framework.

Once located, the run-time engine must assure that the components are compatible and connect them. The run-time engine is not aware of the types of data that connected objects will handle, we deal that by applying the Typed Connections design pattern explained in section 5.1. In short, this design pattern allows to establish a type dependant connection construct between two components without the connector maker knowing the types and still be type safe.

4.3 Thread safe communication in real-time

One of the main issues that typically need extra effort while programming is multi-threading. In real-time audio applications based on a data-flow graph, the processing core is executed in a high priority thread while the rest of the application is executed in a normal priority one following the Out-of-band and In-band partition pattern [10]. Being in different threads, safe communication is needed, but traditional mechanisms for concurrent access are blocking and the processing thread can not be blocked. Thus, new solutions, as the one proposed by the Port Monitor pattern in section 5.2, are needed.

4.4 System back-end

Target applications, being real-time processing, have smaller application logic than others but it still has application logic to define. Most of the application logic is coupled to the sink and sources for audio data and control events. Audio sources and sinks depend on the context of the application: JACK, ALSA, ASIO, DirectSound, LADSPA... So the way of dealing with threading, callbacks, and assigning input and outputs is different in each case. The architectural solution for that has been providing backend plugins to deal with this issues.

This also transcends to the user interface as sometimes the application may let the user to choose the concrete audio sink or source, and even choose the audio backend.

5 Enabling design patterns

Probably the two most complex technical problems involved in implementing this visual prototyping architecture are the following: Managing connections when port types are not limited but extensible; and monitoring data being processed in the high-priority thread from a UI thread. Interestingly, similar problems to these are often found in other contexts. However, their solutions are very similar, so they can be generalized and formalized as a design pattern.

A software pattern is a proved solution to a recurring problem. It pays special attention to the context in which is applicable, to the competing "forces" it needs to balance, and the teaching component on the implications of its application. Patterns provide a convenient way to formalize and reuse design experience. However, neither data-flow systems nor other audio audio-related areas have yet received many attention on domain specific patterns.

The next sections describes Port Monitor and Typed Connections two patterns for those mentioned problems.

5.1 PATTERN: Typed Connections

Context

Most simple audio applications have a single type of token: the sample or the sample buffer. But more elaborated processing applications must manage some other kinds of tokens such as spectra, spectral peaks, MFCC's, MIDI... You may not even want to limit the supported types. The same applies to events channels, we could limit them to floating point types but we may use structured events controls like the ones OSC [11] allows.

Heterogeneous data could be handled in a generic way (common abstract class, void pointers...) but this adds a dynamic type handling overhead to modules. Module programmers should have to deal with this complexity and this is not desirable. It is better to directly provide them the proper token type. Besides that, coupling the communication channel between modules with the actual token type is good because this eases the channel internal buffers management.

But using typed connections may imply that the entity that handles the connections should deal with all the possible types. This could imply, at least, that the connection entity would have a maintainability problem. And it could even be unfeasible to manage when the set of

those token types is not known at compilation time, but at run-time, for example, when we use plugins.

Problem

Connectable entities communicate typed tokens but token types are not limited. Thus, how can a connection maker do typed connections without knowing the types?

Forces

- Process needs to be very efficient and avoid dynamic type checking and handling.
- Connections are done in run-time by the user, so they can mismatch the token type.
- Dynamic type handling is a complex and error prone programming task, thus, placing it on the connection infrastructure is preferable than placing it on concrete modules implementation.
- Token buffering among modules can be implemented in a wiser way by knowing the concrete token type rather than just knowing an abstract base class.
- The set of token types evolves and grows.
- A connection maker coupled to the evolving set of types is a maintenance workhorse.
- A type could be added in run time.

Solution

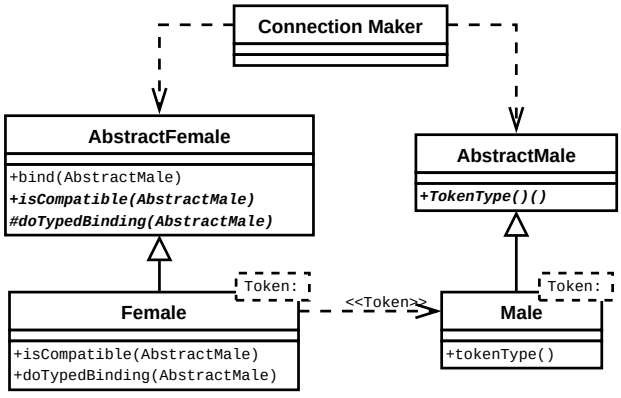


Figure 6: Class diagram of a cannonical solution of Typed Connections

Split complementary ports interfaces into an abstract level, which is independent of the token-type, and a derived level that is coupled to the token-type. The class diagram of this solution is shown in figure 6.

Let the connection maker set the connections throught the generic interface, while the connected entities use the token-type coupled interface to communicate each other. Access to typed tokens from the concrete module implementations using the typed interface.

Use run-time type checks when modules get connected (*binding time*) to get sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely just on compile-time type checks.

To do that, the generic connection method on the abstract interface (`bind`) should delegate the dynamic type checking to abstract methods (`isCompatible`, `typeId`) implemented on token-type coupled classes.

Consequences

By applying the solution, the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is assured by checking the dynamic type on binding time and relying on compile time type checks during processing time. So this is both efficient and safe.

Because both sides on the connection know the token type, buffering structures can deal with tokens in a wiser way when doing allocations, initializations, copies, etc.

Concrete modules just access to the static typed tokens. So, no dynamic type handling is needed.

Besides the static type, connection checking gives the ability to do extra checks on the connecting entities such as semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

Related Patterns

This pattern enriches Multi-rate Stream Ports and Event Ports, and can be also useful for the binding of the visualization and the Port Monitor.

The proposed implementation of Typed Connections uses the Template Method [8] to call the concrete binding method from the generic interface.

Examples

OSW [3] uses Typed Connections to allow incorporating custom data types.

The CLAM framework uses this pattern notably on several pluggable pairs such as in and out ports and in and out controls, which are, in addition, examples of the Multi-rate Stream Ports and Event Ports patterns.

But the Typed connection pattern in CLAM is not limited to port like pairs. For example,

CLAM implements sound descriptors extractor modules which have ports directly connected to a descriptor container which stores them. The extractor and the container are type coupled but the connections are done as described in a configuration file, so handling generic typed connections is needed.

The Music Annotator [12] is a recent application which provides another example of non-port-like use of Typed Connections. Most of its views are type coupled and they are mostly plugins. Data to be visualized is read from an storage like the one before. A design based on the Typed Connection pattern is used in order to know which data on the schema is available to be viewed with each vista so that users can attach any view to any type compatible attribute on the storage.

5.2 PATTERN: Port Monitors

Context

Some audio applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization needs just to be fluid, the processing has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread has real-time requirements and is a high priority scheduled thread. But because the non real-time monitoring should access to the processing thread tokens some concurrency handling is needed and this often implies locking.

Problem

We need to graphically monitor tokens being processed. How to do it without locking the real-time processing while keeping the visualization fluid?

Forces

- The processing has real-time requirements (ie. audio)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- Just the processing is not filling all the computation time

Solution

The solution is to encapsulate concurrency in a special kind of process module, the *Port monitor*, that is connected to the monitored out-port. *Port monitors* offers the visualization thread an special interface to access tokens in a thread safe way.

In order to manage the concurrency avoiding the processing to stall, the *Port monitor* uses two alternated buffers to copy tokens. In a given time, one of them is the writing one and the other is the reading one. The *Port monitor* state includes a flag that indicates which buffer is the writing one. The *Port monitor* execution starts by switching the writing buffer and copying the current token there. Any access from the visualization thread locks the buffer switching flag. Port execution uses a *try lock* to switch the buffer, so, the process thread is not being blocked, it is just writing on the same buffer while the visualization holds the lock.

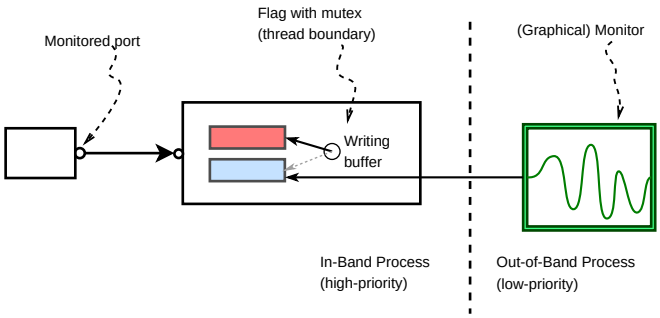


Figure 7: A port monitor with its switching two buffers

Consequences

Applying this pattern we minimize the blocking effect of concurrent access on two fronts. On one side, the processing thread never blocks. On the other, the blocking time of the visualization thread is very reduced, due that it only lasts a single flag switching.

Any way, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be reading always the same buffer. That may happen when every time the processing thread tries to switch the buffers, the visualization is blocking. This effect is not that critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

Another issue with this pattern is how to monitor not a single token but a window of tokens. For example, if we want to visualize a sonogram (a color map representing spectra along the time) where each token is a single spectrum. The simplest solution, without any modification on the previous monitor is to do the buffering on the visualizer and pick samples

at monitoring time. This implies that some tokens will be skipped on the visualization, but, for some uses, this is a valid solution.

Related Patterns

Port Monitor is a refinement of Out-of-band and In-band Partition pattern [10]. Data flowing out of a port belongs to the In-band partition, while the monitoring entity (for example a graphical widget) is located in the out-of-band partition.

It is very similar to the Ordered Locking real-time pattern [13]. Ordered Locking ensures that deadlock can not occur, preventing circular waiting. The main difference is in their purpose: *Port Monitor* allows communicate two band partitions with different requirements.

Examples

The CLAM Network Editor [14] is a visual builder for CLAM that uses Port Monitor to visualize stream data in patch boxes. The same approach is used for the companion utility, the Prototyper, which dynamically binds defined networks with a QT designer interface.

The Music Annotator also uses the concurrency handling aspect of Port Monitor although it is not based on modules and ports but in sliding window storage.

6 Growing a data-flow pattern language for audio

Typed Connections and Port Monitor patterns are part of a broader catalog [15], with 10 patterns that address recurrent problems in data-flow audio systems and which is expected to grow with new patterns.

Some patterns of this catalog are very high-level, like Semantic Ports and Driver Ports, while other are much focused on implementation issues, like Phantom Buffer). Although the catalog is not domain complete, it could be considered a *pattern language* because each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one, to further refine the solution. These relations form a hierarchical structure drawn in figure 8. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

This pattern catalog shows how to approach the development of a complete data-flow system in an evolutionary fashion without the need to do *big up-front design*. The patterns at the top of the hierarchy suggest that you start with high

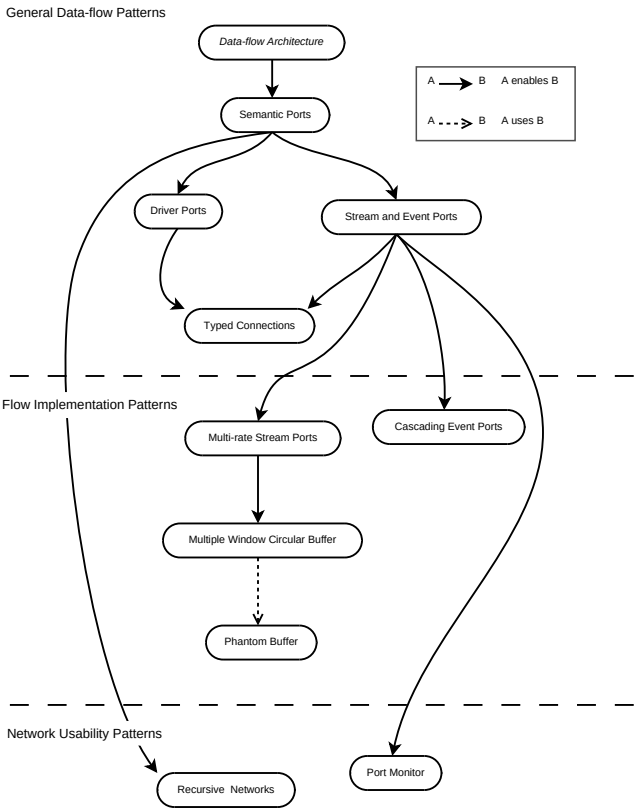


Figure 8: The audio data-flow pattern language. High-level patterns are on the top.

level decisions, driven by questions like: "do all ports drive the module execution or not?" and "do you have to deal only with stream flow or also with event flow?" It might also happen that at some point you will need different token types. Then you'll have to decide "does ports need to be strongly typed while connectable by the user?", or "does the stream ports needs to consume and produce different block sizes?", and so on. On each decision that introduces more features and complexity you face a recurrent problem that is addressed by one pattern in the language.

The main limitation of this pattern language is that do not cover all the typical problems in its domain. Some features of existing data-flow systems proposes recurring problems that should be incorporate into the language. Those include: communication of modules running in different processes, module firing scheduling, integration of heterogeneous (push based vs pull based) systems, ports hand-shacking for meta-data propagation, etc.

The architecture for visual prototyping pre-

sented in this article also contains many recurring problems that could be generalized as patterns. However, our opinion is that more application examples are needed for those patterns to mature.

7 Conclusions

The presented architecture has been already implemented as free software within the CLAM framework and it is available for download as source code or as binary for several platforms. Several already built applications are provided such the ones shown on the screen captures in this article.

Using this architecture, one can build audio applications in few minutes, and developers may concentrate on the development of novel components. Still some work is needed on the implementation of connection classes and back-ends so that they could be provided also as plug-ins.

This article presents some ideas that we hope could help on improving the audio software development by offering an architecture that enables visual development of full applications, a ready to use implementation of such architecture within the CLAM framework, and two related design patterns for a further reuse of design experience.

8 Acknowledgements

Authors of this paper would like to thank Dietmar Schuetz for being our mentor in pattern writing, encouraging us to improve the patterns again and again. We are very grateful to Ralph Johnson, a member of the Gang of Four, who provided insightful feedback and courage to keep our effort on data-flow patterns, during the PLoP workshops. We also thank contributions from past developers and signal-processing experts at the MTG lab. Josep Blat, from the UPF, have provided great support for the CLAM project. This work has been funded by UPF scholarships and by a grant from the STSI division of the Catalan Government. We are also indebted with the Trolltech crew and the Linux Audio Community for giving us the chance to build upon their work.

References

[1] M. Puckette, "Pure Data: Another Integrated Computer Music Environment," in *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, 1996, pp. 37–41.

[2] G. Tzanetakis and P. Cook, *Audio Information Retrieval using Marsyas*. Kluewe Academic Publisher, 2002.

[3] A. Chaudhary, A. Freed, and M. Wright, "An Open Architecture for Real-Time Audio Processing Software," in *Proceedings of the Audio Engineering Society 107th Convention*, 1999.

[4] Clam website. [Online]. Available: http://www.iua.upf.es/mtg/clam

[5] J. Blanchette and M. Summerfield, *C++ GUI Programming with QT 3*. Pearson Education, 2004.

[6] (2006, Dec.) The fast light toolkit (fltk) homepage. [Online]. Available: http://www.fltk.org

[7] Glade home page. [Online]. Available: http://glade.gnome.org

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] A. Alexandrescu, *Modern C++ Design*. Addison-Wesley, Pearson Education, 2001.

[10] D. A. Manolescu, "A Dataflow Pattern Language," in *Proceedings of the 4th Pattern Languages of Programming Conference*, 1997.

[11] M. Wright, "Implementation and Performance Issues with Open Sound Control," in *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Association, 1998.

[12] X. Amatriain, J. Massaguer, D. Garcia, and I. Mosquera, "The clam annotator: A cross-platform audio descriptors editing tool," in *Proceedings of 6th International Conference on Music Information Retrieval*, London, UK, 2005.

[13] B. P. Doublass, *Real-Time Design Patterns*. Addison-Wesley, 2003.

[14] X. Amatriain and A. P., "Developing cross-platform audio and music applications with the clam framework," in *Proceedings of the 2005 International Computer Music Conferenc (ICMC'05)*, 2005, in press.

[15] P. Arumí, D. Garcia, and X. Amatrian, "A data-flow pattern catalog for sound and music computing," in *Pattern Language of Programming PLoP 2006*, Oct. 2006.

Model-Driven Software Development with SuperCollider and the UML

Jens GULDEN
Formal Models, Logic and Programming (FLP),
Technical University Berlin
jgulden@cs.tu-berlin.de

Abstract

The SuperCollider programming language is widely perceived as a script-like interface to the corresponding SuperCollider audio server. However, the language has grown to a serious object-oriented programming language by now, and hence allows applying visual modeling techniques for model-driven development of object-oriented software systems. This article shows how diagram-based visual modeling with the Unified Modeling Language (UML) can be used for creating SuperCollider software.

Keywords

Modeling, Model-Driven, Object-Oriented, SuperCollider, Unified Modeling Language (UML)

1 Introduction

While the SuperCollider programming language ([1]) is most commonly used just as a scripting language for passing commands to a SuperCollider audio server, it also provides all basic language constructs for developing (small or medium-sized) object-oriented software systems. Such language constructs are e. g. the declaration of *classes*, *instantiation* of objects, *polymorphism* (method-overwriting), *class*- and *instance-variables*, object-*references* etc., which are all available in SuperCollider.

Because of its support for object-orientation, visual modeling techniques can be used together with the SuperCollider language, such as *class-diagrams* of the Unified Modeling Language (UML) which provide a visual description of the declarations that make up an object-oriented software system.

The present article describes how the UML can be applied for model-driven development of SuperCollider software. The upcoming chapter two introduces UML class-diagrams and how they are used for modeling object-oriented software systems. The third chapter shows how code in the SuperCollider language is integrated to implement a model's declarations, and in chapter four runnable code is generated from a model. Chapter five then demonstrates how a specific UML-tool is configured to generate executable code in the SuperCollider language. The sixth chapter summarizes the relevant mappings between visual class-diagram elements of the UML on the one hand, and language constructs of the SuperCollider language on the other hand. In chapter seven, an example is presented of how model-driven development with the UML has been applied to create a SuperCollider implementation of the XML Document Object Model (DOM). Chapter eight finally gives information on how to download the presented work, chapter nine sketches possible future plans, and the closing tenth chapter contains a short conclusion.

2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML, [2]) is not a programming language. The word “language“ denotes a set of visual elements used in diagrams that describe the architecture of object-oriented software systems.

UML *class-diagrams* express which classes are part of the software system, which operations and variable-members they have, and how the classes are related to each other. Class-diagrams thus provide an overview on the declarative structure of

an object-oriented software system. Figure 1 shows a small example class-diagram.

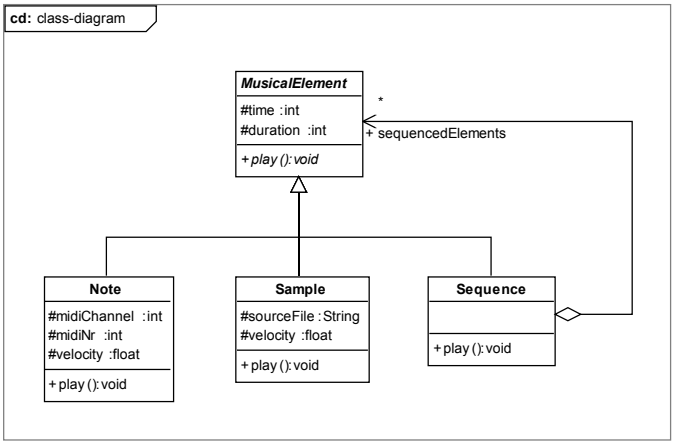


Fig. 1: Class-diagram example

Class-diagrams describe all declarations in an object-oriented software system, but they do not contain any programming language code (editing code is discussed in the following section).

When developing software, the UML does not replace a programming language like the SuperCollider language. Instead, it embeds the overall structure of the language into a visual diagram.

Consequently, any UML class-diagram can be transformed into program code of an object-oriented programming language. The above example diagram could e. g. result in program code as shown in example 1.

```
/*
 * SuperCollider3 source file "MusicalElement.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class MusicalElement ---
//
MusicalElement {

    // --- attributes

    var time; // type  int
    var duration; // type  int



    // --- play() : void ---
    //
    play {
        "ABSTRACT".die; // simulate abstract method
    }


} // end MusicalElement


/*
 * SuperCollider3 source file "Note.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Note ---
//
Note : MusicalElement {

    // --- attributes

    var midiChannel; // type  int
    var midiNr; // type  int
    var velocity; // type  float


    // --- play() : void ---
    //
    play {
        // ... do something to play the midi note ...
    } // end play


} // end Note


/*
 * SuperCollider3 source file "Sample.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Sample ---
//
Sample : MusicalElement {

    // --- attributes

    var sourceFile; // type  String
    var velocity; // type  float


    // --- play() : void ---
    //
    play {
        // ... do something to play the sample ...
    } // end play


} // end Sample


/*
 * SuperCollider3 source file "Sequence.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Sequence ---
//
Sequence : MusicalElement {

    // --- relationships

    var <>sequencedElements; // 0..*-relation to type
MusicalElement



    // --- play() : void ---
    //
    play {
        sequencedElements.do( { // play elements in sequence
            arg element;
            waitUntilTime(element.time); // ...pseudo-code...
            element.play();
        } );
    } // end play


} // end Sequence
```

Example 1: Generated code from the example diagram

UML class-diagrams are also called *models* of the software. Graphical models can help to express the core ideas behind a software system on a conceptually higher level than it is possible with raw program code. This is why this kind of visual software development is called *model-driven development*.

There are several advantages which motivate the use of model-driven development. An individual programmer can benefit from better navigable code and more efficient access to elements of the software system, compared to plain files. Also,

tasks that otherwise would require stereotype typing work can be automated using visual editing, e. g. the creation of new classes or the declaration of relationships among classes.

When communicating about software in a group of people, visual diagrams help to enrich the semantics of the formalized software system and allow to express complex domains of discourse using geometric means and spatial constellations. UML class-diagrams are thus especially helpful for sharing ideas about the architecture of a software system among several people.

3 SuperCollider method bodies in the UML-model

As the UML is not a programming language itself, but provides graphical elements for declaring the structure of object-oriented software, the actual implementation of methods is still done by conventional programming using a traditional programming language.[1] Thus, the method bodies declared in the visual diagram are to be 'filled out' by textual programming. Some UML-tools provide their own source-code editor that allows editing a method-body inside the modeling application.[2]

Fig. 2 shows a screenshot of how this can look like within a specific UML-tool. The source-code editor at the bottom contains the code of the method highlighted in the diagram above.

4 Generating executable SuperCollider code

In order to get executable software from the model, a complete set of source-files is generated from it. These files contain traditional source-code in the *target programming language* and can then be compiled or interpreted to finally make up an executable program. It depends on the UML-tool which target programming languages are available to generate code for. In an ideal case, a UML-tool can be configured freely to generate code for any object-oriented language. This is the case with the tool used for demonstration in this article, *Poseidon for UML* ([4]). This tool uses editable code-generation templates for configuring the target programming language, which allows generating code not only for well-known standard object-oriented languages such as Java or C++, but also for SuperCollider.

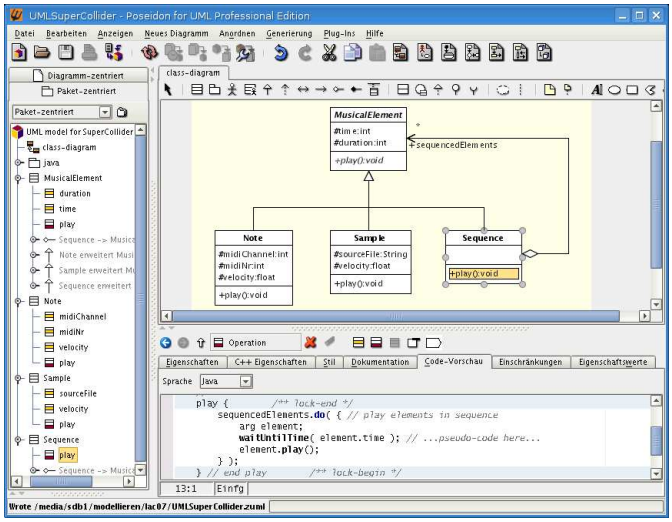


Fig. 2: Editing the source code of a method-body

5 Configuring a UML-tool to generate SuperCollider code

There is no standard on how UML-tools are to be configured for a new target programming language. As a consequence, this section is restricted to the use of the UML-tool Poseidon for UML ([4]), which is used for demonstration throughout this article.

In Poseidon, the code generation process is controlled by file templates into which variable code fragments from the model are inserted while evaluating the template. By evaluating properties from the model via variables and simple case-distinctions (*if*-branches, etc.), the desired SuperCollider code is generated. Example 2 shows a sample of the code-generation template for the SuperCollider language, as it has been used to configure the tool Poseidon.

[1] There are approaches which also allow to derive dynamic behaviour of software systems, thus the code of method-bodies, from graphical diagram modeling. This is, however, beyond the scope of this article.

[2] An alternative approach is using external code-editors (usually within integrated development environments) for code-editing, and letting the modeling-tool take care for synchronizing between the model and the code. This approach is called “round-trip-engineering”, while the process described in this article (editing code inside the modelig-tool and then generating source-files) is called “forward-engineering”.

```
## --- Render a single attribute. ---
#macro (renderOneAttribute $preparedAttr)
...
#set ($SCvisibility = "")
#if ($static.indexOf("static")!=-1)
#set ($SCvar = "classvar")
#else
#set ($SCvar = "var")
#end
#if ($visibility.indexOf("public")!=-1)
#set ($SCvisibility = "<>")
#end
    ${SCvar} ${SCvisibility}${name}${initialValueExpr};
      // type #stripPkg($preparedAttr.getTypeAsString())
#end


## --- Render attributes from associations/relationships. ---
#macro (renderAttributesForAssociationEnds $prepAssocEnds)

#foreach ($preparedAssocEnd in $prepAssocEnds)
...
#set ($SCvisibility = "")
#if ($visibility.indexOf("public")!=-1)
#set ($SCvisibility = "<>")
#end
    ${SCvar} ${SCvisibility}${name}${SCinitialValueExpr};
      //${typeCommentMulti}relation to type #stripPkg(
$preparedAssocEnd.getTypeAsString() )

#end
#end
#end
```

Example 2: Part of the code-generation template for SuperCollider, as used with *Poseidon for UML*

Although the use of templates of this kind is specific to Poseidon for UML, other UML-tools can also be configured for non-standard target programming languages. The principles demonstrated in this article are portable to other modeling-tools.

6 Mapping between UML and SuperCollider

This section summarizes the mappings between UML model elements and the corresponding SuperCollider language constructs. These mappings have already been implicitly encoded in the configuration templates discussed in the previous section.

Table 1 lists the mappings between the UML model elements and the SuperCollider language constructs as they are used in the code-generation process. This list can be used as a basis for configuring further UML-tools to use SuperCollider as the target programming language.

UML	SuperCollider
Class	Class
Package	None, UML-packages are ignored for SuperCollider code-generation. They can however be useful inside the UML model to organize classes.[3]
Attribute	(Instance-) Variable
Static Attribute (underlinedIdentifier)	Class-Variable
Method ('operation')	Method ('message'/'function')
Static Method (underlinedIdentifier)	Class-Method
Abstract Method (*italicIdentifier*)	Method which always throws a runtime error (thus must be over-written to be used)[4]
0..1/1..1-Relationships	Variable, reference to single instance
0..*/1..*-Relationships	Variable, list containing references to multiple instances
Types of attributes, function-arguments and return-values.	None, but comments. The SuperCollider language is untyped, but including type declarations in the UML model will generate comments in the source code which increases readability of the code.
Public (+) visibility of attributes	Variable, with getter and setter declaration (“<>”).

[3]This implies that class names must be chosen to be unique all over the whole set of classes available for SuperCollider. A possible convention is to use a package-like prefix, e. g. `DOMNode`, `DOMElement`, `DOMText` etc.

[4]This is currently implemented by calling `"ABSTRACT".die` but might more elegantly be done using `this.subclassResponsibility()`.

Package (~), Protected (#), Private (-) visibility of attributes	None, normal variable is used. Besides getter/setter access, visbility is not reflected in the SuperCollider language.
Public (+), Package (~), Protected (#), Private (-) visibility of methods	None. Method visibility is not reflected in the SuperCollider language.

Table 1: Mapping between UML model elements and SuperCollider language constructs

Distinguishing between different levels of visibility is not reflected explicitly in the SuperCollider language, however, it may be useful to use visibility-modifiers in the model in order to express additional semantics about the declared elements (e. g., use 'private' visibility (-) for members that are only used internally by the same class, and 'protected' visibility (#) for methods that are intended to be used by subclasses only).

7 Example: XML-DOM implementation for SuperCollider

The Document Object Model (DOM, [5]) is a standardized set of interfaces for representing XML documents in an object-oriented structure, making them handable for processing with an object-oriented programming language. [6] is an implementation of the DOM API version 1.0 for SuperCollider. The library has been completely developed using a model-driven development environment as described above, and thus serves as a real-world example for model-driven, UML-based software development with SuperCollider. The library has already proven its usefulness in music-related software-projects such as the emulation of the signal-noise of the ENIAC computer (ENIAC NOMOI, [7]), or creating a prototype implementation for an XML-format describing parts, positions and time in audio waveforms ([8]).

The DOM API version 1.0 is an early revision of the DOM standard, however already useful for most common XML-related tasks. It defines a minimum set of 10 interfaces which have been modeled using the class-diagram shown in Fig. 3.

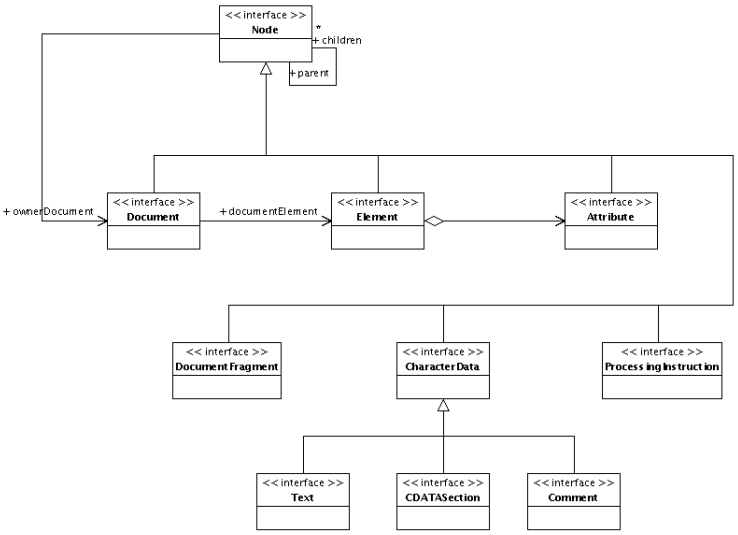


Fig. 3: Class-diagram of the XML DOM API implementation

The corresponding classes for implementing these interfaces in the SuperCollider language have also been developed entirely using a UML class-diagram, see appendix A.

The source code shown in the following example 3 gives an insight into the code as generated from the XML DOM API model.

```
// --- class DOMNode ---
//
// Attributes of the node are stored via Dictionary-entries.
//
DOMNode {

    // --- attributes

    classvar <>node_ELEMENT = 1; // type  int
    classvar <>node_ATTRIBUTE = 2; // type  int
    classvar <>node_TEXT = 3; // type  int
    classvar <>node_CDATA_SECTION = 4; // type  int
    (...)
    var nodeValue; // type  String
    var nodeName; // type  String
    var nodeType; // type  int
    (...)
    var attributes = nil; // type  Dictionary

    // --- relationships

    var ownerDocument; // 0..1-relation to type  DOMDocument
    var parent; // 0..1-relation to type  DOMNode
    var children; // 0..*-relation to type  DOMNode

    // --- getNodeName() : String ---
    //
    getNodeName {
        ^nodeName;
    } // end getNodeName

    // --- setNodeName(name) : void ---
    //
    setNodeName { arg name; // type String
        nodeName = name;
    } // end setNodeName
    (...)
```

Example 3: SuperCollider code generated from the XML DOM API model, class `DOMNode.sc`

An additional example directly related to musical applications is shown in appendix B. The diagram contains an early sketch of a synthesizer class library, as it could be developed with SuperCollider.

8 Downloads

All technical details about the presented approach can be found at `http://swiki.hfbk-hamburg.de:8888/MusicTechnology/751`.

The UML integration is done on top of the UML modeling tool *Poseidon for UML* ([4]). The code-generation templates for SuperCollider can be downloaded at `http://swiki.hfbk-hamburg.de:8888/MusicTechnology/uploads/751/sc-templates.zip`. They are freely available and licensed under the GPL ([9]).

Poseidon is a commercial product. A free-of-charge "Community Edition" of Poseidon was available for non-commercial use until version 4.x. Since version 5.x, only an evaluation version limited to use in time is available free of charge any longer.

The XML DOM API example is available both as model-file for Poseidon and as generated SuperCollider source code at `http://swiki.hfbk-hamburg.de:8888/MusicTechnology/747`. The XML DOM API implementation is also free and licensed under the GPL.

9 Future Plans

As the product Poseidon for UML is no longer available as a free-of-charge Community Edition, it has become necessary to migrate to a free UML modeling tool. One possible candidate to migrate to might be Fujaba ([10]).

10 Conclusion

Applying visual modeling techniques is not restricted to wide-spread mainstream programming languages. Since the SuperCollider programming language is equipped with all necessary constructs for object-oriented software development, it becomes possible to adopt existing UML-tools to SuperCollider as their target programming language. The article has shown that visual modeling of software is no longer an exotic issue of theory and science only, but has reached a degree of practical usefulness that allows its application also in non-mainstream development contexts.

The SuperCollider language, on the other hand, has proven to be mature enough for being involved in a state-of-the-art model-driven development process.

References

[1] Gulden, J., *Developing with [SuperCollider and] the Unified Modeling Language (UML),* software, `http://swiki.hfbk-hamburg.de:8888/MusicTechnology/751`

[2] McCartney, J. et al, *SuperCollider3 - A real time audio synthesis programming language*, software, `http://supercollider.sf.net/`

[3] Booch, G., Jacobson, I., Rumbaugh, J., *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading (Mass.), 1999

[4] Gentleware AG, *Poseidon for UML*, software, `http://www.gentleware.com/products.html`

[5] World Wide Web Consortium (W3C), *Document Object Model (DOM),* `http://www.w3.org/DOM/`

[6] Gulden, J., *XML parsing and formatting [for SuperCollider], s*oftware, `http://swiki.hfbk-hamburg.de:8888/MusicTechnology/747`

[7] Carlé, M. et al, *ENIAC NOMOI,* media project, `http://www.medienwissenschaft.hu-berlin.de/~mc/ENIAC_NOMOI_eng.php`

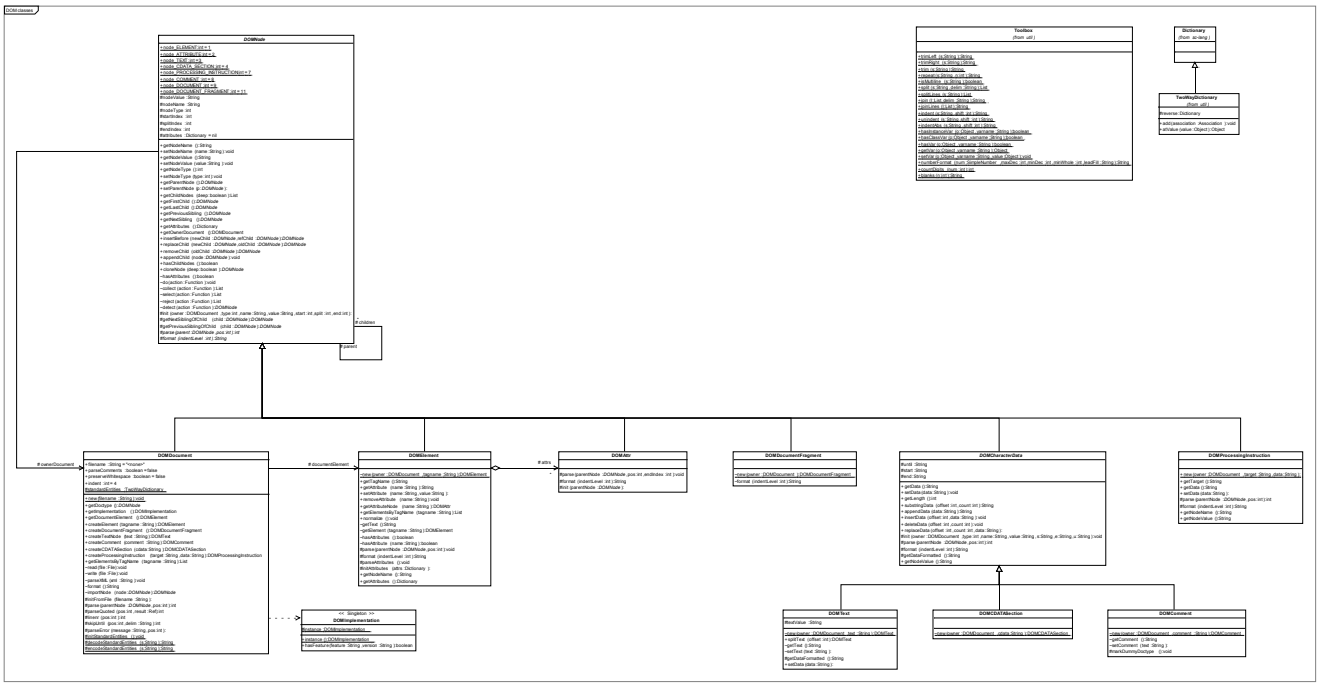
[8] Gulden, J., Rutz, H., *Proposal for an XML format representing Time, Positions and Parts of Audio Waveforms,* this conference

[9] GNU Software Foundation, *GNU General Public License (GPL)*, legal license, `http://www.gnu.org/licenses/gpl.txt`

[10] University of Paderborn, Software Engineering Group, *Fujaba Tool Suite*, software, `http://www.fujaba.de/`

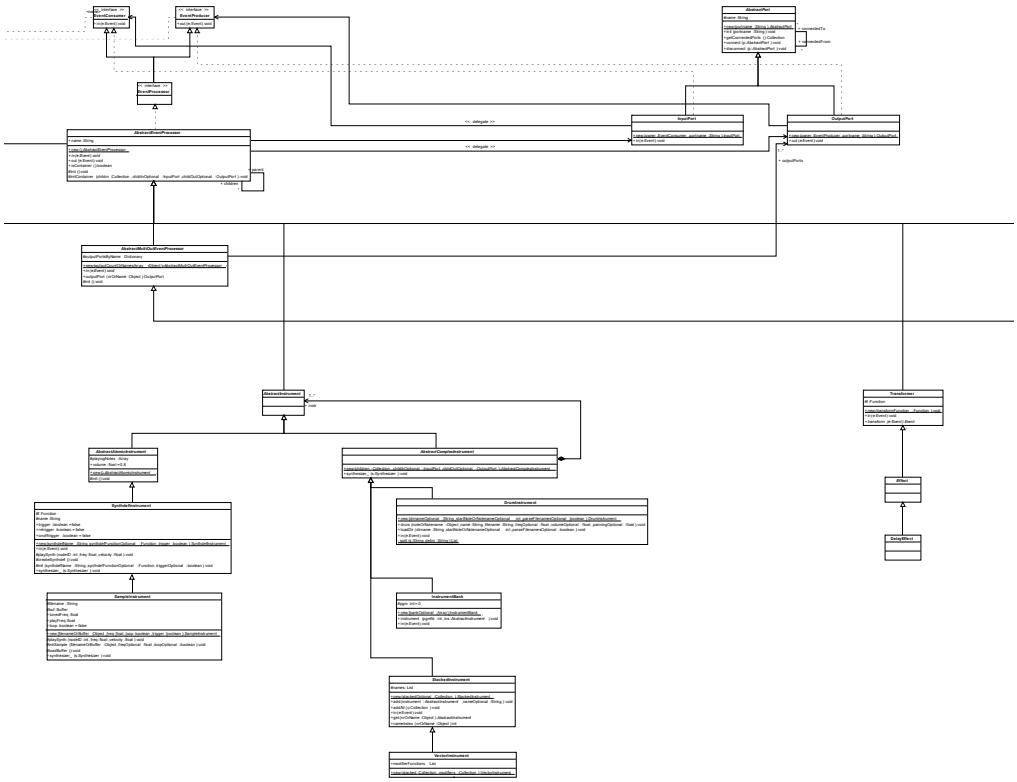
Appendix A

Full-scale UML class-diagram of the XML DOM API implementation (see section 7).



Appendix B

Early sketch of a synthesizer class library for SuperCollider (partial, for demonstration only).



pure:dyne

Aymeric Mansoux and **Antonios Galanopoulos** and **Chun Lee**
Goto10
12, rue Charles Gide
86 000 Poitiers
France
contact@goto10.org

Abstract

pure:dyne is a live GNU/Linux distribution optimized for the purpose of real-time audio and visual performance. As its name suggests, pure:dyne is built upon the dyne:II platform and originally optimized for PureData. However, pure:dyne now also contains several other interesting and useful creative software, and is becoming evermore practical to be used as a complete GNU/Linux distribution for both media art and daily tasks. This paper therefore aims to introduce and discuss several aspects surrounding pure:dyne thus encouraging the usage and feedback of this project.

Keywords

goto10 dyne live-distribution PureData Supercollider Csound media-art FLOSS

1 Introduction

The development of pure:dyne [1] can be traced back to the inclusion of PureData in the dyne:bolic liveCD distribution [2]. As this addition later became increasingly popular, there was suddenly a demand to increase its support for PureData in a more serious production context. Meanwhile, the dyne:II [3] core that Denis Rojo [4] had been developing for the forthcoming version of dyne:bolic provided the necessary development tools needed to make such customized distribution for PureData. As a result, a collaborative effort had begun between dyne.org and Goto10 in early 2005 to work towards a distribution based on the dyne:II core.

After a year of development, pure:dyne started to take shape and began its beta testing. In late 2006, pure:dyne officially left beta to have its first public release. Today, pure:dyne gathers a growing user community and has been used in numerous workshops and performances.

Although many multimedia oriented live GNU/Linux distributions can be found nowadays, many aspects of pure:dyne still remain unique amongst them. This paper hopefully will introduce and demonstrate such features and design, and ultimately encourage its usage.

2 Design principles

Throughout the process of making pure:dyne, several design principles were clearly outlined from the beginning. They can be briefly listed as below:

- pure:dyne is made by practitioners for practitioners
- pure:dyne should be accessible to non technical users
- pure:dyne will be optimized and kept minimal

One of the most important aspects of pure:dyne is that it attempts to offer both "practical" and "portable" solutions for the practitioners in the fields of FLOSS based digital art. Although there are many portable distributions available, they are mostly used for demonstration purposes. pure:dyne, on the other hand, allows artists to build extensive works upon it while keeping the entire system, including artists' works, very portable. This makes it an attractive alternative for artists who wish to develop projects but do not have access to a dedicated environment.

Moreover, accessibility is also an important part of pure:dyne. pure:dyne recognizes artists who intend to take advantage of the innovations in creative FLOSS but do not have the resources and abilities to walk through the lengthy installation, configuration and even compilation of such software. Because of this, pure:dyne aims to provide a working environment that requires a minimal learning curve to be productive with it.

Lastly, pure:dyne follows a minimalist approach in system setup. This enables it to be more streamlined and "clutter free". For example, the default desktop environment is FluxBox as window manager and applications such as Rox-Filer and Xfe can be used for handling the conventional representation and navigation of directories. pure:dyne also includes window managers such as ratpoison, evilwm and dwm. Such an approach enables users to achieve greater productivity when using the system.

[1] http://puredyne.goto10.org

[2] The first inclusion of PureData can be found in dyne:bolic1.4

[3] dyne:II is platform in which a fully functional system can be built upon it. for more detail, please refer to section of this paper.

[4] founder of Rasta Software and the key maintainer of dyne:bolic

3 Usage

Before this paper proceeds further, there are two important concepts in pure:dyne that should be clarified.

- **Dock** - A dock refers to an "installation" of pure:dyne onto the host system. A dock contains all necessary components that are required to boot pure:dyne entirely from the storage device. The process of docking is extremely straightforward, it only requires copying the /dyne directory from the CD or ISO image onto a partition readable [5] by pure:dyne.
- **Nest** - A nest (.nst) is a file that a user can create once pure:dyne has successfully booted. This file contains a user's home directory and configuration files [6]. The nest file can be stored either on the hard disk or on a portable storage device such as a usb key. During the boot process, pure:dyne will look for the nest in any of the partitions it finds and mounts the nest at the appropriate location. Through the integration of UnionFS [7], users can easily save and store any modifications made on the system.

With further development of the dock and the nest in dyne:II [8], pure:dyne can be used with a great deal of flexibility. For example, a system running from a CD or hard disk, in combination with a portable storage device will result in a complete functional system. Once the system is successfully booted, a user can simply write to his or her own home directory and continue working the same way no matter which storage device is being used. One other obvious advantage of the docking system is that pure:dyne can co-exist with other operating systems in a very straight forward manner, as everything is contained in one single directory. Updating to a newer version of pure:dyne only requires to overwrite the content of the dyne directory. Lastly, by simply creating new users following the conventional GNU/Linux method, a nest can also support a multiple user system.

dyne:II also contains a modular system in which applications can be packaged and distributed. Each package is a compressed [9] .dyne file in the /dyne/modules directory. For instance, the applications contained in pure:dyne are simply a pure.dyne module of dyne:II. This means that users and developers can simply package their favorite applications [10] and exchange between them. To include a new module, simply copy the .dyne file into the /dyne/modules directory and either reboot or mount the module directly.

To summarize, pure:dyne can be used/installed in the following ways:

- Used with the CD alone, without saving user data
- Used with the CD in conjunction with a portable storage device that contains the nest
- Used with a dock on the hard disk plus a nest either on the hard disk or portable storage device
- Used with both the dock and the nest on the portable storage device. for example, running pure:dyne entirely from solid state memory.

4 Optimization

As mentioned previously, pure:dyne's main emphasis is in the context of real-time applications. Because of this, pure:dyne consists of several optimizations that are different from dyne:bolic 2.x.

Firstly, the optimization is targeted at the i686 architecture. This is because pure:dyne aims to support more modern hardware, as the real-time audio/visual applications are typically more demanding on cpu cycles. pure:dyne employs the kernel based on Ingo Molnars's real-time patch.

Secondly, pure:dyne makes the installation of necessary drivers to take advantage of the hardware possible and straightforward. For example, one can easily install the ATI and NVIDIA graphics driver to take advantage of the modern graphics cards in order to obtain the acceleration required by video/visual applications. Furthermore, it also includes support for various firewire sound cards through the FreeBoB driver.

Lastly, the gcc compilation flags used in pure:dyne are typically more aggressive than those used in dyne:bolic or usual binary based GNU/Linux distributions. Currently, relevant applications in pure:dyne are compiled with the following flags: -O3 -ffast-math -fomit-frame-pointer -mmmx -msse -pipe.

5 Applications

Applications that are optimized in pure:dyne can be briefly listed below [11] consult the pure:dyne website:

- PureData
  - PureData
  - Gem, PDP, PiDiP, GridFlow of the externals and abstractions from the PureData cvs

[5] current supported filesystems are: fat vfat msdos ntfs ufs befs xfs reiserfs hfsplus ext2 ext3

[6] A nest contains the /home, /root, /var, /tmp and /usr/local

[7] UnionFS allows transparent overlay of files and directory from different filesystems. http://www.unionfs.org

[8] Both dock and nest existed in dyne:I. However, these two elements were significantly further developed in dyne:II

[9] .dyne modules use the squashfs read-only filesystem. http://squashfs.sourceforge.net

[10] currently there are modules for Ardour, network tools, Gimp, OpenOffice, BitTorrent, dvd authoring and more

[11] For the complete listing, please

- Audio
  - SuperCollider
  - Chuck
  - Csound
- Visual
  - Fluxus
  - Packet forth

Besides the pure.dyne module, the pure:dyne distribution also comes with the audio.dyne module from dyne:bolic 2.x which provides the applications for hard disk recording, sequencing, and sound editing. Furthermore, Jack is provided by the dyne:II core. As a result, the combination of pure:dyne with the audio module would result in a fairly comprehensive digital audio workstation.

6 dyne:II and pure:dyne

As mentioned previously, pure:dyne is built using the dyne:II platform. There are, however, some subtle differences that should be pointed out.

dyne:II is a system derived from LFS[12] and initiated by Denis Rojo and Alex Gnoli. It provides the core functionalities such as booting, nesting, docking and the modular system. In other words, dyne:II offers a platform on which a complete live distribution can be built. For example, dyne:bolic 2.x is developped using the dyne:II core system.

pure:dyne, on the other hand, is not only a live distribution built using the generic dyne:II, but also contains several customized core components. Because of this, pure:dyne can be said to be using a customized dyne:II system developed by Goto10. For example, pure:dyne contains its own kernel and has a different optimization policy. Moreover, pure:dyne also provides some developers' tool that are unique to it. In short, pure:dyne not only consists of a pure.dyne module but also its own branch of the dyne:II core system.

The relationship between the two cores is by no means independent of each other. That is to say, the development remains very close between them. As a result, changes can be merged. For example, new features introduced in the modified pure:dyne core could potentially be merged into the generic dyne:II core after they are tested and have proven to be stable. Similarly, new components in generic dyne:II core can be adopted by pure:dyne's customized core. Last but not least, any dyne modules are universal whichs allows applications to be used and shared between the users of these different systems.

7 Which Dyne?

Because of the differences between dyne:bolic 2.x and pure:dyne, it can sometimes be ambiguous as to which of the two should be used. In general, dyne:bolic 2.x offers more stability across a wider range of legacy hardware and pure:dyne is somehow more "bleeding-edge", it adopts the latest drivers and patches to gain performance. Moreover, pure:dyne is created for a very specific context, while dyne:bolic 2.x can be seen as more generic.

	pure:dyne	dyne:bolic 2.x
Type	live-distribution	live-distribution
Core	dyneII customized	dyneII generic
Module policy	.dyne	.dyne
Target hardware	i686	i586
Optimization	aggressive	generic

Table 1: comparison of pure:dyne and dyne:bolic 2.x. note that only the audio related modules are listed in the table

8 Current status

Currently, pure:dyne is at its first official release (version 2.3.6). It has already proved to be usable and stable enough to be employed in real world scenarios. For example, pure:dyne has been used in many workshops where participants are able to learn the software and have a complete and functional system to carry on their learning after workshops ended. Furthermore, pure:dyne has also been seen used in live performances and even installations for a period of weeks without problems.

9 To do

Efforts will be aimed towards the development of the following parts of pure:dyne at this moment.

9.1 Documentation

Apart from the actual live distribution, pure:dyne would also like to provide documentation for the necessary components surrounding its usage and development. For example, there should be a knowledge base for users to learn more about the software it includes and also on more advanced configurations. Developers should also be able to find necessary information to further customize it to suit their needs and ultimately take part in producing the future releases.

9.2 Hardware support

pure:dyne would also like to provide more stable support for hardware that is commonly used by practitioners in the digital art scene. For example, the support for the MacIntel machines is currently being tested and will hopefully be included in the next stable release.

9.3 Software package

pure:dyne is always keen to discover interesting and innovative creative software to include in its package. This is a constant reminder for the pure:dyne developers. After the release of the second stable version,

[12] Linux From Scratch. http://www.linuxfromscratch.org/

a particular attention will be given to a compilation of interesting FLOSS based software art.

10 Conclusion

The collaboration between dyne.org and goto10.org has led to the successful production of pure:dyne. More importantly, many of its current users find pure:dyne useful and it has proven to fulfill its original goals. pure:dyne hopes to continue the fruitful relationship with dyne.org to achieve a higher standard in the future. This would hopefully contribute to raising the awareness of FLOSS culture and tools in the digital arts scene.

11 Acknowledgment

Goto10 would like to thank Denis Rojo and Alex Gnoli for developing dyne:II and their valuable help and advices throughout the making of pure:dyne. Goto10 would also like to thank the digital research unit of Huddersfield in UK for their support during the start of the project. Goto10 would also like to thank BEK in Bergen and Waag Society in Amsterdam for providing the hosting solution for pure:dyne. Lastly, Goto10 would like to thank everyone who contributed to pure:dyne by developing it, using it and disseminating it.

The One Laptop Per Child (OLPC) Audio Subsystem

Jaya Kumar
Independent Developer,
Gurgaon, India — KL, Malaysia
jayakumar.alsa@gmail.com

Abstract

The OLPC is a Linux based laptop-like device intended as an educational tool targeted at developing countries. The audio subsystem of the OLPC faces typical challenges such as minimizing power consumption, performance/quality and component cost pressures and tradeoffs, as well as less common challenges such as the need to repurpose audio input as an oscilloscope or analog input system. This paper explains issues encountered during support and development of ALSA and low level audio support on the OLPC. It will also touch on possible future plans for the low level audio software side of the OLPC.

Keywords

OLPC audio, Analog Input, Speaker-Microphone feedback, Power Management

1 Introduction

From a hardware perspective, the OLPC is a fairly full featured embedded device. It has strong multimedia capabilities. This includes a decent audio subsystem, decent video subsystem and even a video input subsystem. Table 1 provides an overview of the hardware features. Figure 1 is a picture of the beta test (B-Test1) unit of the OLPC courtesy of the OLPC team[1].

The OLPC kernel is currently based on a 2.6.19 kernel, thus incorporating ALSA 1.0.13. The operating system for the OLPC is based on a stripped down and heavily customized version of Fedora. The userspace audio engine is the csound server. Current audio applications incorporated in the OLPC operating system are TamTam and Squeak eToys. Lots of other multimedia applications have been run on this system including Quake, mplayer, Collabra's Telepathy among others.

[1]The image is Copyright (C) OLPC, used with permission under Creative Commons Attribution2.5 License

Figure 1: OLPC B-Test1 Unit

2 Hardware Architecture

The OLPC board currently uses an embedded x86 architecture. The audio controller for this architecture is a core within the cs5536 ASIC as shown in Figure 2. The cs5536 is the integrated southbridge for this architecture. This controller interfaces with an AD1888 AC97 codec from Analog Devices.

The communication between the Geode GX2 cpu and the cs5536 is PCI [1]. However, neither device implements a fully traditional PCI bus controller [2]. For example, the AC97 Controller (ACC) in the cs5536 is not actually a PCI core but rather communicates via GeodeLink with the GeodeLink PCI SouthBridge which then handles communication with the Geode to stream to/from host memory. The important PCI command types such as Memory Read/Write, I/O Read/Write and others are fully implemented by the GeodeLink PCI SouthBridge. But PCI Configuration Read/Write support is not implemented by either the Geode GX2 or the CS5536. This affects system software, including audio software, in a

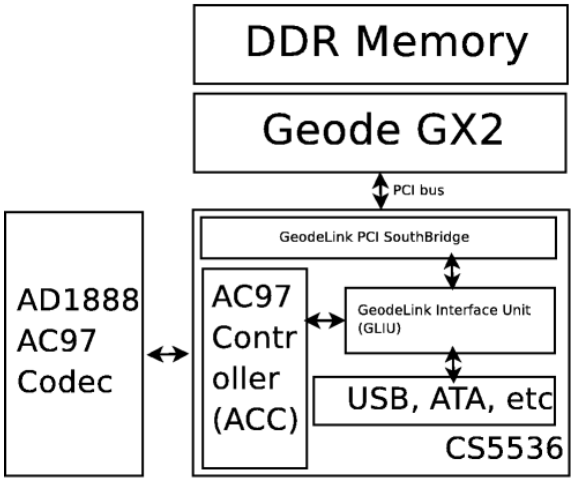


Figure 2: GX-CS-AC97 Architecture

unique way when contrasted with other x86 systems.

In typical Geode systems including the current version of the OLPC, the BIOS (LinuxBIOS on the OLPC) utilizes a legacy module called VSA (Virtual System Architecture) that identifies GeodeLink devices and virtualizes the PCI configuration space registers. This means that when a driver attempts to access a PCI configuration register, a SMI (system management interrupt) is generated. The SMI is then handled by software, in this case, the VSA code within the BIOS that performs MSR (machine specific register) based reads/writes to perform the appropriate task.

2.1 Mechanical Architecture

The OLPC exposes on-board audio via left and right speakers grill-mounted on both sides of the front fascia as pictured in Figure 3. On-board audio input is exposed via a ported microphone mounted on the front fascia. External audio output and input is enabled via two standard red and green 3.5mm jacks on the left side of the display head. The audio input jack is dual purposed for analog input as well. A general picture of the OLPC B-Test1 unit is provided in Figure 1.

3 ALSA on OLPC

The lowest level of the ALSA subsystem on the OLPC is the two hardware drivers. These are the cs5535audio driver and the AD1888 AC97 driver. Both have been part of the ALSA tree prior to initiation of the OLPC project. The cs5535audio driver prior to OLPC was a fairly

Hardware	Features
CPU	AMD Geode GX2 (2 Watts @ 366 MHz)
Memory	128 MB DDR400 SDRAM
Storage	512 MB NAND Flash
Audio	CS5536 ACC AD1888 AC97
Camera	VGA CMOS
Video	200dpi 7" LCD 1200x900 (BW reflective) 640x480 (Color transmissive) 250 nits
Wifi	Marvell 88W8388 802.11b/g/s
USB	3 x USB2.0 ports
SD	1 x SD slot
Battery	2 hour NiMH
Input	Keyboard, touchpad

Table 1: Hardware Features

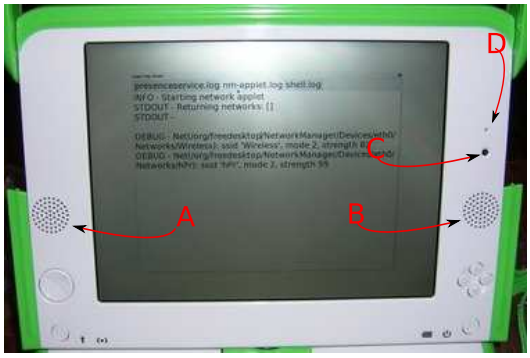


Figure 3: OLPC Front Fascia
A - Left Speaker
B - Right Speaker
C - CMOS Video Camera
D - Microphone

standard ALSA PCI driver. The same is true of the AD1888 AC97 driver.

3.1 cs5535audio OLPC support

Prior to OLPC, the cs5535audio driver was typically utilized in always-on devices in industrial control systems. In those environments, power management was not an attribute that was sought after. The introduction of the OLPC project gave rise to the motivation to add power management functionality to cs5535audio. This support was fairly straightforward in combination with the AC97 suspend/resume support

that is part of ALSA. From the ACC perspective, the task to be done by the driver is very typical. It utilizes ALSA to suspend PCMs, to take the AC97 codec through the suspend sequence and finally to turn off the AC Link and to take the ACC to D3 state. Further additions to cs5535audio to support OLPC were related to the analog input which is covered in Section 4.5.

3.2 AD1888 OLPC support

There is broad support for almost all of the AD1xxx series AC97 codecs from Analog Devices in ALSA. Testing with the OLPC found a minor issue related to duplicate controls. Further additions to support OLPC were bug fixes associated with power management, new mixer controls and analog input support.

3.3 AC97 Power Management

One of the important changes to ALSA that is very relevant to the OLPC is the addition of aggressive power-saving support of AC97 codecs. This addition was done by Takashi Iwai. The way that it works is that when all PCMs are closed and a reasonable delay has passed suggesting that audio activity has ceased, ALSA proceeds with switching capable AC97 codecs into suspend mode. Upon opening of a PCM, ALSA resumes the AC97 codec. This allows for reduced power consumption without noticeable loss of functionality.

4 Problems

Several interesting problems were identified as part of the bringup of audio on the OLPC. Some of these issues remain open.

4.1 AC97 read failure

One of the first problems identified with ALSA on the OLPC was two failed reads of the AC97 codec. Specifically, ALSA's AC97 support code attempts to read the AC97_VENDOR_ID2 on the AD1888 codec. This AC97 read fails even with a large timeout. The failure is derived from the ACC not asserting a bit in its status register called STS_NEW. However, the value that is read back from the AC97 register despite the lack of status bit assertion is 0x7E805368. Thus, the 0x5368 which is returned is correct, as per the AD1888 datasheet thereby allowing everything else to proceed as normal despite the apparent read failure. This issue was observed only with the OLPC board. Other systems such as cs5535/realtek combinations and cs5536/wolfson combinations have not exhibited this symptom. The cause of this read failure has not yet been identified and remains an open issue.

4.2 AD1888 duplicate controls

Marcelo Tosatti was the first to identify this issue on the OLPC. This problem had to do with the fact that ALSA creates the majority of controls in *snd_ac97_mixer_build*. For example, one of those controls was the "Surround Playback" control. On the AD1888, this mixer creation conflicted with the same control being provided by the *snd_ac97_controls_ad18xx_surround* table thereby resulting in a fail out. The solution was trivial which was to precondition the initial mixer creation with the AC97_AD_MULTI flag that represents that set of Analog Device codecs.

4.3 AD1888 magic resume register

As with many hardware designs, the AD1888 contains its share of magic registers that are not formally documented[3]. An unusual suspend/resume bug triggered the discovery of one such register. The observed symptom is best described by the test sequence:

```
insmod snd-cs5535audio.ko
amixer set PCM 100 on
amixer set Master 100 on
aplay /tree/test.wav
# playback works fine
# now suspend to D3
echo -n 3 > /sys/devices/pci0000
\:00/0000\:00\:0f.3/power/state
# try to play
aplay /tree/test.wav
# playback is blocked as expected
# now resume to D0
echo -n 0 > /sys/devices/pci0000
\:00/0000\:00\:0f.3/power/state
# now try to playback
aplay /tree/test.wav
# playback proceeds at normal rate
but no sound is heard on the output
```

After some debug, a register was found that made things work after resume which is AC97 register 0x60, or AC97_CODEC_CLASS_REV. The AD1888 datasheet does not refer to this register. The AC'97 2.3 spec says:

> Extended Codec Registers Page Structure Definition Page 00 of the

> Extended Codec Registers is reserved for vendor specific use. Driver writers should not access these registers unless the Vendor ID register has been checked first to ensure that the vendor of the AC '97 component has been identified and the usage of the vendor defined registers is understood.

Tweaking that register on resume allowed things to work correctly. The reason why this is the case is still unknown.

4.4 Uniqueness of PCI IDs

As was mentioned in Section 2, the BIOS currently provides the virtualized PCI configuration registers. This implies that the BIOS software loaded on to the SPI flash (PROM equivalent) is what determines the PCI Vendor/Subvendor and Device/Subdevice ID for the system. Almost all Geode systems use either the BIOS from Insyde Systems (a BIOS vendor) or LinuxBIOS. In the latter case, LinuxBIOS on the OLPC currently has a legacy build of the VSA included within it. This perpetuates the situation where almost all cs5536 based boards have exactly the same PCI Vendor/Subvendor and Device/Subdevice IDs. This makes it rather hard for the ALSA driver to identify which board it is being used on in order to apply board specific quirks.

In the case of the OLPC, it is sufficiently different that it warranted having a CONFIG_OLPC config entry. This enables the various associated drivers to apply OLPC specific quirks using that flag.

4.5 Analog Input support

An interesting feature on the OLPC is that it can repurpose the microphone input jack as an analog input. This capability can be used to interface the OLPC with analog input devices such as a photodiode. This was demonstrated by Barry Vercoe at WorldComp '06, where a spoon with a photodiode connected to the analog input was used as a conductor's baton to control audio.

The method by which the analog input capability is implemented is by having the trace from the input jack split to two separate paths. One path is a standard resistive-capacitive path to the microphone input pin on the AD1888. The other path is an unbiased direct path to the same pin. The selection of these paths is done via a standard analog switch controlled by an 8051 based embedded controller (EC). The EC is then interfaced to the host via GPIO pins. From an audio software perspective, communication with the EC is currently via x86 port IO. The B2 model of the OLPC will move this analog switch control over to a Geode GPIO, thereby avoiding the communication with the EC.

There are therefore several aspects to analog input support. The first set are features of the AC97 codec. These are the ability to disable the V_{Ref} bias that is normally internally applied to the microphone input and the ability to disable the high pass filter that is internally applied to limit input to typical audio frequency ranges. The second set is the need for the host to instruct the EC to toggle the analog switch. The current implementation for analog input support on OLPC is done by adding an OLPC specific quirk to cs5535audio. This quirk adds a new mixer control named "Analog Input". This function performs the task of toggling both the EC bits and the AC97 bits in order to achieve the desired functionality. This therefore exposes the analog input mode decision to userspace through the standard ALSA mixer API and can be controlled via amixer and other regular tools as well.

4.6 Speaker-Microphone Feedback

As can be seen from Figure 3, the right speaker and the microphone are physically very close. One of the effects of this industrial design decision is that there is significant amounts of feedback when both the speaker and microphone are active at the same time. This has a negative effect on telephony (voice/video over IP) type applications. This problem is not yet solved. Several proposals have been put forth including one suggesting that the right speaker be disabled when the microphone is active.

5 Future Enhancements/Issues

5.1 ASoC/DAPM

One of the most interesting future issues is continued improvement of power management. With this in mind, the ALSA System On Chip and Dynamic Audio Power Management support contributed by Liam Girdwood and others is very relevant to the OLPC. In particular, the ability to dynamically switch on and off DAI capabilities, tune AC97 clock rates, and performing switch-level power management on the AC97 codec while completely transpar-

ent to applications is very attractive for minimized power consumption on portable devices like the OLPC. Work is in progress to add ASoC/DAPM support for OLPC.

5.2 Speaker-Microphone Feedback

As was mentioned before in Section 4.6, this feedback issue is a challenging problem. One of the approaches that may be of interest from the ALSA software perspective is to use a dmix type plugin to downmix stereo streams and redirect to only the left channel when it is detected that the on-board microphone is enabled.

5.3 VSA elimination

The virtualization of PCI configuration registers by the VSA can be removed. This is possible if the kernel's generic PCI configuration dependent code was replaced by OLPC specific rd/wr-msr code. That code would handle the GeodeLink support. This would also aid boot performance. This work is in progress and is being done by Mitch Bradley of the OLPC team.

5.4 Preempt and Tickless

There are two kernel features being actively tested and developed on the OLPC. These are the tickless kernel and realtime preemption support. Realtime preemption is clearly of benefit to the audio subsystem [4]. Tickless should not be detrimental to audio. Tickless will serve to reduce power consumption on OLPC and also improve timer resolution.

6 Conclusions

The OLPC provides good audio quality and performance for a device of it's type. It provides a fully featured environment for application developers, as can be seen from the wide range of audio-enabled applications that are running on it such as TamTam, eToys, and others. It presents an interesting environment for ALSA and Linux Audio in general. It has aspects of a typical embedded system. That is, it has very tightly constrained and hard-wired resources in terms of cpu, audio controller, audio codec, memory and storage. But it also has aspects of a high end system. It has a wide set of multimedia capabilities thus requiring full broad ALSA and other audio functionality. This combination will be likely to continue to provide unusual and challenging problems to be solved by the Linux Audio community.

In the event that the OLPC project is successful, ALSA and Linux Audio in general will be contributing something directly positive to the daily lives of a large number of human beings through out the planet. Further, if the commercial aspects of the OLPC project are similarly successful, then this will encourage more hardware manufacturers to become more involved in the ALSA and Linux Audio community in general. This will help ALSA and Linux Audio in general to thrive.

7 Acknowledgements

My thanks go especially to Takashi Iwai and Jaroslav Kysela for their advice on many of the OLPC audio issues and reviewing all the code for the OLPC audio changes. Special thanks to Liam Girdwood of Wolfson Micro for coming to FOSS.IN in Bangalore to present ASoC/DAPM and improve ALSA development here. Thanks to the OLPC organization for kindly supplying the board for driver testing and development. Also my sincere thanks to all the members of the ALSA community for much help over many years.

References

[1] AMD. *AMD Geode CS5536 Companion Device Data Book*. AMD, USA, 3/14/2006.

[2] AMD. Amd geode gx and lx processor based systems virtualized pci configuration space. *AMD*, 1(32663C):3–20, 2006.

[3] Analog Devices. *AC '97 SoundMAX AD1888 Codec Data Book*. Analog Devices, USA, 2005.

[4] Lee Revell. Realtime audio vs. linux 2.6. *Proceedings of Linux Audio Conference 2006*, 2006(1):21–24, 2006.

FireWire (Pro-)Audio for Linux

Pieter PALMERS
pieterpalmers@users.sourceforge.net

Abstract

FireWire audio devices are flooding the (semi-)pro audio interface market. At LAC2005, the FreeBoB project has been presented, aiming to implement support for devices built on top of the BridgeCo BeBoB platform.
This paper discusses the evolution of the FreeBoB project towards a framework generic enough to support a virtually any FireWire based audio device, BeBoB based or not, conforming to the 'official' standards or not. An overview of the design and implementation of the library is presented, and some key issues and their solutions are described.

Keywords

FireWire, IEEE1394, driver, FreeBoB

1 Introduction

Since the presentation of the FreeBoB project's proof-of-concept code at LAC2005, a lot of progress has been made. An extensive number of changes and complete rewrites, followed by a 6 month beta testing stage, resulted in the release of FreeBoB-1.0, mid october 2006.

Meanwhile the interest of developers, users and vendors in FireWire audio on Linux was steadily increasing. This can be illustrated by a meeting held at LAC06 to discuss Linux support for non-BeBoB based devices. It became clear that the FreeBoB project could be a starting point to provide generalized FireWire audio support on Linux.

It was however obvious that the existing codebase was not easily extensible to provide support for the various options that exist for device discovery and data transport. The code had to be rewritten from scratch, marking the birth of FreeBoB-2.0, around may 2006.

Section 2 describes the FireWire system from a FreeBoB point of view. The design and implementation of the FreeBoB-2 library is described in sections 3 and 4. Finally section 5 provides some concluding remarks and briefly summarizes the end-user functionality.

2 System overview

This section discusses the different components in a FireWire audio setup on Linux. It starts with a description of the hardware bus topology and the way FireWire works from a FreeBoB perspective[1], based upon [1]. Next, the current linux1394 driver stack and userspace interface will be discussed [2]. To conclude this section, the FreeBoB-2 library structure will be presented.

[1] The FireWire bus is designed for a multitude of functions. Aside from media related transport such as audio and video, common applications are storage access (SBP-2), networking (eth1394) and device control (e.g. Agilent ParBERT 81250). All of these applications use different subsets of the FireWire standard(s).

2.1 FireWire from an audio point of view

2.1.1 Bus topology

In summary, the FireWire[2] bus is a high-speed, peer-to-peer, self-configurating bus supporting true hot-plugging and plug-and-play. Whenever a device is (dis)connected, the bus is reset and reconfigured. The interfaces and protocols are designed such that a bus reset does not influence the functional behavior of a device, meaning that devices can be added or removed while all others remain fully operational.

A FireWire topology appears as one 64-bit memory mapped space, of which the first 10 bit specify the bus the device is connected to, and the next 6 bits specify the node ID of the device. This makes that a single bus can support 63 nodes (one broadcast ID) and that there can be 1023 busses (there is one 'local bus' ID) in a FireWire topology. Note however that the capability to address multiple busses is not used yet. The case where one computer contains more than one host controller doesn't qualift as one of multiple busses, as the different host controllers don't share the same address space. In such a case the system consists of multiple “topology's”[3].

The remaining 48 bits of the address provide a 256 terrabyte address space inside the device. Parts of this space are standardized (e.g. Configuration ROM), but most of it can be freely used.

2.1.2 Data transfer

The FireWire specification defines two main types of data transport: Isochronous and Asynchronous, both being packet based.

Isochronous traffic has the following properties:

- Broadcast one-to-one or one-to-many on a specific 'channel'
- A node can send only one packet per cycle

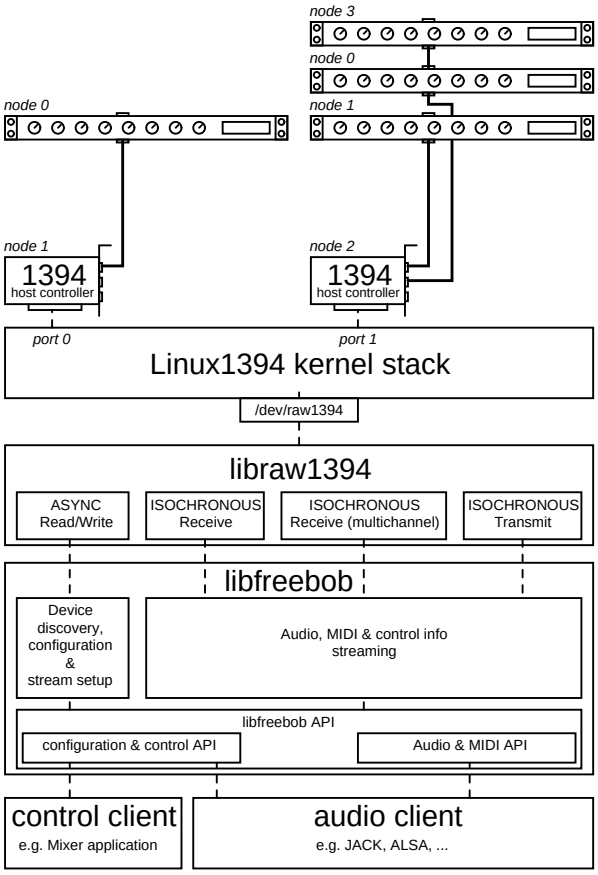


Figure 1: FreeBoB-2 system overview

- Up to 80% of the total bandwidth can be reserved and guaranteed for it
- It occurs at time intervals (cycles) that are regularly spaced in time (125us).
- No error correction or retransmission

The remaining part of the total bandwidth can be used by Asynchronous traffic, exibiting the following properties:

- Between two specific nodes
- No guaranteed bandwidth
- Nodes can send multiple packets per cycle
- An async packet is acknowledged and optionally responded to by the receiving node, enabling error detection and/or correction

Isochronous traffic should be used for time-critical, error tolerant data transfer, while asynchronous transfers are intended for non error tolerant data or non time-critical transfers.

FireWire audio devices always use isochronous traffic to transport the audio and midi data, and

[2] The FireWire bus was developed by Apple, and is standardized by the IEEE as IEEE1394. It is also called i.Link by Sony.

[3] There is no official terminology for this

usually use asynchronous transfers for device configuration and control[4].

2.1.3 Bus management roles

During the configuration phase, some special bus management tasks are assigned to nodes capable of executing them. For this description, the most relevant ones are the isochronous resource manager (IRM) and the cycle master.

The IRM keeps track of the isochronous channels that are in use, and the remaining bandwidth available for isochronous traffic. It provides the bandwidth guarantees that come with isochronous traffic.

The cycle master determines the timing for the isochronous traffic. At the start of every isochronous cycle, being a time slice with a nominal 125us period, the cycle master transmits a 'cycle start' packet. This packet contains the value of the cycle timer register (CTR) of the cycle master at the start of the cycle. This CTR is updated by the cycle master's clock source at 24.576MHz. The other nodes synchronize their CTR to this cycle start packet. The cycle master hence defines the concept of “time” on the bus.

2.2 The linux1394 stack

The linux1394 stack consists of a set of kernel modules and a userspace library that provides a clean interface to kernel space. The relevant parts of the stack are the *ohci1394*, *ieee1394* and *raw1394* kernel modules, and the *libraw1394* library.

2.2.1 Kernel modules

The linux1394 kernel stack was designed such that a multitude of host controllers could be supported. In order to accomplish this, a separation between the generic IEEE1394 bus functions and the driver for the hardware implementing them was introduced. The generic part is implemented in the *ieee1394* kernel module, while the host-controller specific code[5] is in modules like *ohci1394* and *pcilynx*.

The *raw1394* module together with *libraw1394* provides the kernel-user space interface. The *raw1394* module is not intended to be used separately.

2.2.2 Userspace API: libraw1394

When applications want to use the linux1394 subsystem, they should use *libraw1394* instead of directly addressing */dev/raw1394*.

It supports asynchronous traffic through blocking api calls that encapsulate the complete request-ack-respond process. It also enables applications to act as a target for async transactions by having them register handlers for specific FireWire address ranges.

Isochronous transfer is implemented by registering a callback function to process incoming packets or provide outgoing packets, together with some stream setup and management functions (init, start, stop, ...).

2.2.3 Next-generation FireWire stack

Very recently a new FireWire kernel stack has been proposed to replace the current one. It simplifies the stack by only supporting OHCI cards and eliminating the obese layers. It will also support some more advanced methods to handle isochronous traffic. These should result in lower CPU usage and lower latency due to less intermediate buffering.

The development of this new stack gives is an opportunity to make sure that the specific requirements of pro-audio clients are served. The modular architecture of FreeBoB-2 should allow

[4] Some devices (e.g. MOTU) make use of the isochronous transport channel to transport their status changes, e.g. due to front-panel operations by the user.

[5] In reality there is only one type of host controllers available on the general market, being the OHCI1394 compliant ones. OHCI1394 [3] is a specification for a PCI host controller designed mainly by Intel [XX: and Microsoft?]. It enables the use of a unified driver for all compliant host controllers, and is the reason why FireWire extension cards come without a device driver CD.

for an easy transition to the new stack, resulting in early adoption.

2.3 LibFreeBoB-2

LibFreeBoB-2 is a C++ library that provides the translation between the FireWire packet based and the audio API's frame/buffer based environments. It can be split into two main parts, being the discovery & configuration part and the streaming part.

The discovery & configuration part is responsible for enumerating the devices that are available and supported. It also provides device-specific configuration functions such as setting the samplerate or controlling the hardware mixer. The streaming part translates the isochronous packet streams into audio and MIDI[6] streams and vice versa.

Although there are standards both for device discovery (AV/C) and audio transport (IEC61883-6), a one-implementation-fits-all approach has few chances to work. The device discovery standards are rather vague, making it possible to implement two different subsets with the same functionality, being equally compliant. On top of that, not all manufacturers care about the standards (e.g. MOTU). This is why support for multiple detection & configuration methods and multiple stream processors is needed. Device support is then achieved by combining a specific detection & configuration method with a specific stream processing method. This allows to re-use the common parts between devices.

One example are the BeBoB and DICE-II devices. Both are standards compliant, but the BeBoB discovery cannot be used for the DICE-II devices due to another interpretation of the standards. On the other hand, the stream processor for both devices can be the same, as the standard doesn't leave any room for interpretation there.

The FreeBoB library provides an external C API that can be used to implement audio clients (JACK backend, ALSA plugin) and control clients (mixer application, device control panel, ...).

3 Device Discovery and Configuration Layer

The device discovery and configuration layer performs the following tasks:

- Enumeration of the supported devices present on the bus
- Enumeration of the capabilities of a device
- Configure the device, providing a generalized interface for configuration

This layer has very tight coupling with the device implementation. It is therefore less generalized, and will not be discussed further.

4 The Streaming Layer

The core function of the streaming layer is to (de)multiplex the isochronous streams (iso streams) into audio, MIDI and control streams having the appropriate format for the client. It should also recover the timing information.

First, the design of the streaming layer is described, presenting the general concepts. Then the implementation is described in more detail.

4.1 Design

Figure 2 shows a conceptual overview of the FreeBoB library. On the client side it exposes a set of “ports” that represent the different types of streams provided to the client.

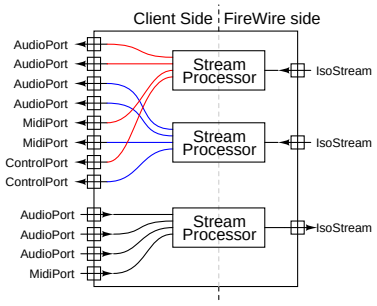


Figure 2: Conceptual overview of the data flow

On the FireWire side there are incoming or outgoing “iso streams”. Inbetween these two interfaces there is a “stream processor” doing the actual work.

[6] There is also support for 'control data' streams that give feedback on the device status.

4.1.1 Client side port interface

Ports define the data transport from and to the clients. They have a type (audio, midi or control) that defines the nature of the data flowing through them, and a datatype property that defines the format of this data (float, uint24, ...). A port also has a direction property (playback or capture).

They can support multiple signalling types, depending on the time granularity required by the data. Two signalling types are currently supported: period and packet signalling.

Period signalled ports transfer the data in 'periods' as they are defined by the client. This allows for period-at-once (de)multiplexing of the isochronous streams, increasing efficiency. This signalling type is used for audio buffers.

For packet signalled ports the demultiplexing is done at the moment a packet is received, making the data is available at the port as soon as it arrives[7]. For playback ports, the data is multiplexed into the stream as soon as it is available at the port. This allows for the high time-granularity as needed by MIDI data.

Another property of a port is the data transfer method. Currently we provide two data transfer methods: blocking read from/write to a ringbuffer, and a direct to memory decode method into client supplied buffers.

4.1.2 FireWire side streaming interface

An 'iso stream' is an abstraction for a sequence of isochronous packets over time. It is linked to a specific isochronous channel in a specific FireWire 'topology' (or port in the *libraw1394* nomenclature). It's type can be 'receive' or 'transmit'.

The actual packet handling for an iso stream is performed by an 'iso handler' that interfaces with *libraw1394*, as indicated in figure 3. The reason for this intermediate iso handler is that one iso handler can serve multiple iso streams. *libraw1394* supports a multichannel isochronous receive mode, that allows to receive multiple channels at once. This mode can become important when a lot of devices are present, because it needs less hardware resources (only DMA engine, instead of one per channel).

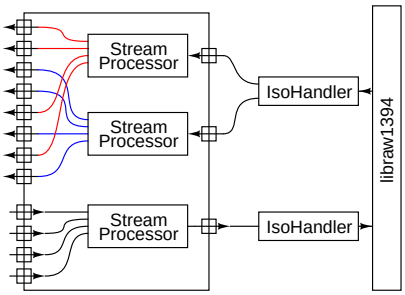


Figure 3: iso streams are fed or consumed by iso handlers

4.1.3 The Stream Processor

A 'stream processor' is responsible for the (de)multiplexing of the ports into/from iso streams, i.e. for the conversion between audio/midi frames and isochronous packets. Every different data transport protocol requires a different stream processor.

The information provided by the detection process is used to construct a set of ports corresponding to the composition of the expected iso stream. It is also used to configure the (de)multiplexing code.

The last important function of the stream processor is recovering timing information (when to signal that a buffer is ready) from the incoming streams, as well as encoding this timing information into the outgoing streams.

4.1.4 Making it work

The presented concepts (ports, stream processors, iso streams and iso handlers) are not 'active' entities. In order to have them perform their task, two 'active' entities are introduced: the 'processor manager' and the 'handler manager'.

The processor manager is responsible for managing and executing the stream processors in the correct manner. This encompasses their

[7] « as soon as it is available » in this context should be regarded as « as soon as the timestamp of the data expires ».

initialization, the detection of period boundaries, the initiation of data transfers, etc...

The handler manager iterates the iso handlers, meaning that it will make sure that the isochronous packets are transfered from the kernel to the iso handlers and vice versa.

4.2 Implementation

The FreeBoB-2.0 streaming layer is written in C++, in order to facilitate abstraction and code reuse. This subsection will present the classes used to implement the design described previously.

4.2.1 The client side port interface

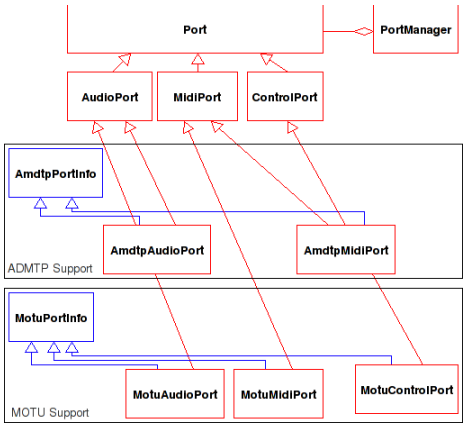


Figure 4: Class diagram for the client side Port interface, showing support for AMDTP (IEC61883-6) and MOTU streaming

Figure 4 shows a class hierarchy for the client side port interface. It starts with a *Port* base class, containing all functionallity needed to implement the operations and properties described in the previous section. This *Port* class is subclassed for every port type.

A generic *Port* does not contain any information on the location of its data in the isochronous packets, as this information is specific to the streaming protocol used for the isochronous traffic. We provide this information by subclassing from a *[*]PortInfo* class. The details of these *[*]PortInfo*'s are protocol dependant, therefore there is no common base class for them. The *StreamProcessor* for a protocol is aware of the details of its corresponding *[*]PortInfo* class.

In order to facilitate the management of a collection of *Ports*, a *PortManager* class has been implemented.

4.2.2 FireWire side streaming interface

The basic entities on the FireWire side, as shown in figure 5, are the *IsoStream* and *IsoHandler* classes. The *IsoStream* class implement the operations nescessary for processing a single isochronous stream. This can be either consuming a sequence of packets provided by an *IsoHandler* of the receive type (*IsoRecvHandler* or *IsoMultiRecvHandler*), or producing a sequence for the *IsoXmitHandler*.

To manage *IsoHandler*'s, an

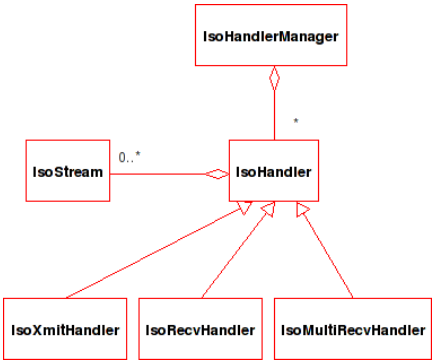


Figure 5: Class diagram for the FireWire side streaming interface

IsoHandlerManager class is implemented. This class creates and destroys IsoHandlers whenever they are needed. In order to link an *IsoStream* with an *IsoHandler*, the *IsoStream* has to be registered with the *IsoHandlerManager*. This class will decide wether to create a new *IsoHandler* or re-use an existing one (in case of multichannel receive). It will also determine the type of handler the *IsoStream* needs (receive or transmit), and its optimal settings. Finally it allows unregistering *IsoStreams*, destroying *IsoHandlers* that are not used anymore.

4.2.3 The StreamProcessor

The StreamProcessor classes are the workhorses in the FreeBoB library. They perform the actual operations to translate the client side data to/from isochronous streams.

Figure 7 shows that at one side, the

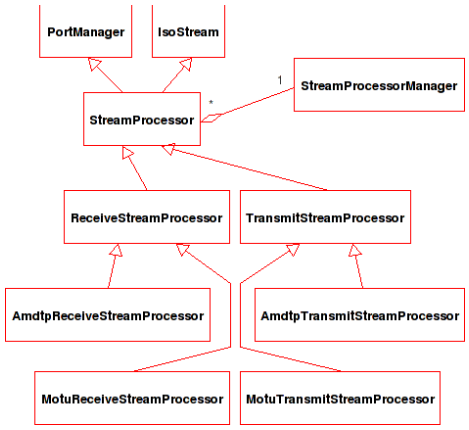


Figure 6: Class diagram for the StreamProcessor class

StreamProcessor is a *PortManager*, as it presents a set of *Ports* to the client side. At the other side, it is an *IsoStream*, as it processes a sequence of isochronous packets.

Leveraging the power of C++, we leave all common functionallity in the *StreamProcessor* base class, and only implement the protocol-specific parts in the child classes.

In order to manage and 'activate' the collection of *StreamProcessors*, a *StreamProcessorManager* class is implemented. This class will be discussed in more detail in the next subsection.

4.2.4 Making it all happen

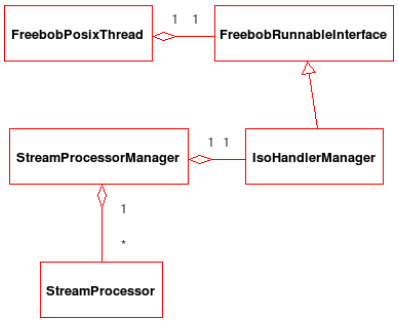


Figure 7: Class diagram for the active part of FreeBoB

The two basic 'active' operations of the library are the reception/transmission of packets, and the translation of these packets into audio/midi/control frames. These operations are implemented by the classes shown in Figure 7.

Isochronous reception and transmission is executed by the *IsoHandlerManager*'s workfunction, that keeps running while the library is started. The *StreamProcessors* should decide if they want to process incoming packets and should always be able to generate valid packets (albeit no-data or empty packets).

On the client side, a wait() function is provided by the *StreamProcessorManager* to have the client wait for a period boundary. Once a period boundary is detected, the client wait() function returns. It should then call a transfer() function to have the *StreamProcessorManager* instruct it's *StreamProcessors* to demultiplex and decode the queued packets (in bulk), and to transfer their contents to the *Ports*.

For the playback direction the inverse operations are performed, i.e. *Port* contents are encoded, packets are dequeued.

5 Issues and gory details

This section will talk about some implementation issues that were met when implementing libFreeBoB-2.

5.1 Embedding and using time information

The IEC61883-6 standard defines a method to embed the timing information of the frames in the packet. Most other streaming protocols (i.e. Motu) use variants of this technique. Therefore it is interesting to discuss this techique.

The idea behind the method is that every packet contains a timestamp that indicates the time at which the sample is to be 'presented'. The sender thus determines the time instant at which the receiver should process the sample. Calculation of the timestamp should be done by recording the FireWire cycle timer at the instant the sample is captured, adding the senders processing delay to this, and then adding some extra time to allow for transfer delays. Receivers are allowed to add some extra time to the received timestamp, as long as this extra time is constant.

The FreeBoB library handles timestamp synchronization by introducing a timestamp aware buffer. The basic idea is that every time one or

more frames are added to the buffer, the corresponding timestamp is passed along. The buffer uses these timestamps to calculate the timestamp of any other frame in the buffer. In order to cope with jitter issues it uses a delay locked loop as described in [4].

Since the timestamped buffer is able to predict the timestamp of any frame in the buffer, it is also able to predict the time instant at which one period of frames will be 'presentable'. This prediction is used by the processor manager to determine the time to wait for the next period. At initialization, one stream processor is elected as synchronization source, and this stream processor will be used to prediction the buffer transfer time instant.

5.2 Synchronization across topologies

Having the timestamp expressed with respect to the FireWire cycle timer poses a significant synchronization challenge. It is not nescessarily true that this cycle timer is related to the sample clock of the device. It can also happen that there are multiple cycle timers in a system, e.g. when two devices are connected to different host controllers (hence different busses). This scenario is currently unsupported, but will be in the future. Supporting it will require that one global time reference is elected, and all other time domains get synchronized to it. This will probably come down to electing the system timer as the global time reference, and implementing a cycle-timer to system time mapping for all domains.

5.3 IPC and multiplexing

There are some unresolved issues regarding IPC and the fact that different data is multiplexed upon the same stream.

The first issue is that capture isn't easily decoupled from playback due to them having to be synchronized. This prevents the use of playback and capture by different applications.

The fact that midi is transmitted along with the audio frames in the same stream poses some issues when interfacing to clients that don't have a midi API. Providing midi to a different application as the audio is also a problem.

In some cases, feedback regarding the device's status is multiplexed into the isochronous streams. This can be for example a notification of a mixer volume change due to a front panel control. Usually the application needing this kind of information (e.g. a mixer application) is different from the one consuming the remainder of the stream (e.g. jack).

5.4 FireWire based processing units

The FireWire bus is not only used for audio interfaces, but also for processing units (e.g. TC PowerCore, SSL Duende). As these devices use the same concepts and protocols as audio interfaces, they fit the FreeBoB framework. It is not clear how they fit in the 'bigger' picture on Linux. In the Mac/Windows world, these devices present themselves e.g. as VST plugins. It is currently unclear to what extent the Linux alternatives provide support for this.

6 Conclusion

This paper presents the FreeBoB-2 library as (part of) a solution for FireWire Audio on Linux. It described the design and implementation of the library, along with some solved and some remaining issues.

References

[1] J. Canosa, Fundamentals of FireWire, Questra Consulting, http://public.rz.fh-wolfenbuettel.de/~bermbach/research/FireWire/files/basics.pdf

[2] www.linux1394.org

[3] The OCHI1394 specification http://developer.intel.com/technology/1394/download/ohci_11.htm

[4] “Using a DLL to filter time”, Fons Adriaensen

Beyond open source music software: extending open source philosophy to the music with CODES

Evandro Manara MILETTO,
Luciano Vargas FLORES,
Daniel Eugênio KUCK and
Marcelo Soares PIMENTA
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Caixa Postal: 15064
91501-970 Porto Alegre, Brazil,
{miletto,lvflores,dekuck,mpimenta}@inf.ufrgs.br

Jérôme RUTILY
Institut Nat. Polytechnique de Grenoble
Domaine Universitaire
38402 St Martin D'Heres, France,
jerome.rutily@ensimag.imag.fr

Abstract

This paper presents CODES – COoperative Music Prototype DESign – a Web-based cooperative music composition environment. This means it allows any person to connect with other users, through the Web, and cooperate with them to draft simple musical pieces, in a prototyping way. Besides describing our main design decisions and overall implementation solutions, we will also briefly discuss our belief that CODES is not just open source music software, but extends the open source notion as it reuses publicly available code (frameworks) and tools, and allows open music production, in a way as collective as open source development.

Keywords

Music prototyping, cooperation, WWW, open source, free music

1 Introduction

Music technology has undergone considerable changes over the last decades, mainly with an increasing use of the Internet. An outcome of these changes is *networked music* [1] [2], which allows experimental artists to explore the implications of interconnecting their computers. Indeed, lately, Internet-based networked music has attracted a wider interest from the music technology community, and the existing applications have evolved towards more sophisticated projects and concepts including, for example, real-time distance performance systems, and various systems for multi-user interaction and collaboration.

Cooperative music composition environments are such a case, which early became an interest to our computer music research group [3]. The CODES project associates Computer Music with concepts from the fields of Human-Computer Interaction (HCI) and Computer Supported Cooperative Work (CSCW), with the intent to design interaction so that the system can be useful and usable even to non-musicians. Besides musicians, novices are also probably interested in creating music and participating in musical experiments, but they lack environments oriented to their profile. On the other hand, musicians have not yet developed the tradition of sharing their musical ideas and collaborating while composing, and so they would equally benefit from the experience in a cooperative environment.

CODES – COoperative Music Prototype DESign – is similar to other Internet-based systems for music composition, such as those in a survey by Weinberg [4]. They also enable users to contribute their own material and to manipulate (listening, altering, refining, etc.) others' contribution, usually through asynchronous interaction and off-line material manipulation. But in addition to common characteristics existing in other systems, CODES addresses several other important aspects to be considered in a collaborative environment for music composition/prototyping, such as group awareness, different levels of interaction, capability to export/import alternative sound formats, and support for music prototyping rationale and for long prototyping sessions. With this set of additional features, one can track back some part of the music prototype and understand the intention and the process followed by its author to compose it, as an alternative way of learning by example.

Also, since we want CODES to be used by anyone, it should not rely on traditional music notation, nor should demand knowledge of music theory for its users to prototype music. So, we developed mechanisms to represent sound patterns as icons, and the option to intelligently suggest them to the user, or to offer him an easier access to

those patterns which could fit well in his music prototype.

Attributes such as these reflect the free/ libre/open-source philosophy we follow in this project. This does not apply only to the software being open-source, but extends throughout the overall requirements of the project, such as for free access, availability, accessibility to non-musicians, portability, reuse of code, collective development approach, support to a collective/cooperative composition approach based on experimentation, etc. Moreover, music is made within CODES in the same collective/cooperative/prototyping way as in open source development. This kind of music will be the result of teamwork, will be a collective product, will be available on the Web and may be always open to further modifications. So, it cannot be dealt with according to the usual authoring laws of the traditional music market (for instance, we must find alternative licensing mechanisms that suit more in this case). It must be seen and treated by us as what some communities are already calling "free music" or "open music" [5] [6], and thus CODES extends the open source notion all the way to the music itself.

This paper is organized as follows. Our main design decisions for the CODES project are presented in section 2. Section 3 describes brief details on the architecture and implementation of the CODES environment. Some aspects of the CODES user interface and of how it is used are presented in section 4. Section 5 discusses briefly why we believe that CODES extends the open source notion beyond its source code development. Finally, section 6 presents some conclusions and future goals.

2 The CODES project: design decisions

The major motivation underlying our proposal is allowing non-musicians to access a virtual space in order to interact with each other, explore sounds together, discuss about this exploration, and retrieve all the discussed information anytime they want. Therefore, we soon discussed some aspects that could be essential to consider in the design of such an environment, which we still use to guide the development of CODES:

- It must run in a *virtual space only*, via Web browser, to ensure that the barrier of geographic distance among partners (physical presence) does not become an issue.
- It must *run on a great diversity of platforms* and browsers (at least, all W3C compliant [7]), minimizing requirements of use and thus increasing accessibility.
- It should allow *independence of a tutor*, and *non-structured groups*, despite the possibility of supporting structured groups with a tutor role, what is usually necessary in learning situations.
- Due to the exploratory nature of how it is used, a very important characteristic should be the users' possibility to perceive and analyze group members' actions on music prototypes, and to know the reasons behind each one of these actions. These are aspects related respectively to *awareness* and to *prototyping rationale*, for which CODES then must provide support. The concept of "awareness", from the CSCW literature, cannot be precisely and uniquely defined [8]. In the context of CODES, the adopted notion of awareness is "the understanding of the actions of other users, what provides for a user a context for his own actions". Prototyping rationale is a mechanism which allows each user to justify his actions, in order to make clear to the others the idea or reasons that guided him to make that decision. As a result, collaborators in a prototype will have access to an explicitly recorded track of all steps that led to the current prototype state.
- It must support *long prototyping sessions*: an important mechanism in any design activity is the ability to interrupt the session and to resume it in order to continue the process from the last break point. A music prototyping session can take many days or even weeks before a final result is reached.
- It should offer *alternative music encoding formats*, making it easy for users to export/import their music between different systems, thus integrating CODES into a wider context of music systems. Standard MIDI was chosen here due to its easy manipulation and compatibility. Although the sounds of synthesized MIDI files played on most PCs are still low quality, it yields some future possibilities, like the conversion from MIDI to conventional music notation. We are investigating the use of some mark-up languages for music – like MusicXML [9], Music Mark-up Language (MML) [10], and the Music Encoding Initiative (MEI) [11] – as interesting alternatives to be explored. We believe that in a near future one of them (or some variation thereof) will be the standardized format of choice for music content on the Web.

As we can see, several design decisions for the CODES project were targeted on allowing more freedom and "openness". We will discuss that

further on section 5. Next we will present briefly some system implementation details.

3 Architecture and implementation of the CODES environment

CODES implementation follows an open source philosophy, aiming as well at providing easy access to its software development and supporting tools. We went after various publicly available software frameworks and design patterns, to reuse well-known solutions. As a consequence, we chose to build CODES respecting the classic Model-View-Controller (MVC) architectural model (see Figure 1). On the server side, it is implemented trough the *WebWork framework* (Java) [12]. As Web server, CODES uses the Apache Tomcat Servlet Container [13], a reference for Java applications. Data persistence is done following the DAO design pattern for data access, implemented by the *Hibernate framework* [14], which stores data in a MySQL database [15]. The system is organized in a few modules (such as the Prototype Manager, the Users Manager, and the Cooperation Manager), and to coordinate them, CODES implements a Façade design pattern, that is an interface which provides modularity. Finally, for the user interface on the client side we are using AJAX (Asynchronous JavaScript and XML), which is supported by this architecture, and we implement it with the *Dojo JavaScript framework* [16].

The Java Sound API allows us to focus system development on both graphical user interface (GUI) and cooperation aspects, making the sound handling part easier because of its components that already offer sound control.

4 The CODES user interface

The user interface was designed to meet requirements related to interaction flexibility, robustness, and easiness of use, as well as to present adequate support when complex musical information is displayed, thus providing an effective interaction between users and the environment. We wanted to reach a balance between user interfaces that are so "easy" for the user that they end up depleting his expressiveness, and others that are so complicated that they discourage beginners.

In music, some peculiarities make the creation and conception processes different from those carried out in other fields. Musical composition is a complex activity where there is no general agreement about what activities have to be done and in which sequence: each person has his own style and way of working. So, the process of music composition is difficult to understand and, therefore, also to learn.

"Prototype" is not a common expression in music literature. In fact, a "composition" is known to be the result of a composer's creative activity. But the emphasis of our work is mainly on the *process* (prototyping), and not on the product itself. The repetitive cooperation cycle in CODES, where online partners refine a musical sketch until its final form is reached, clearly resembles the incremental prototyping cycle adopted in industry, and thus we call its result a *music prototype*.

Music is like an "artistic product", which can be designed through prototyping. A musical idea (notes, chord sequences, rhythms, etc.) is created by someone (typically for a musical instrument), and afterwards cyclically and successively modified and refined according to his initial intention or to ideas that come up during the prototyping process.

We believe *the experience of this prototyping process* is what's most interesting in using CODES. During such an experience, lots of knowledge sharing will take place, by means of the rich interaction and argumentation mechanisms associated, in this environment, to each prototype modification. So, *the process itself* will foster all

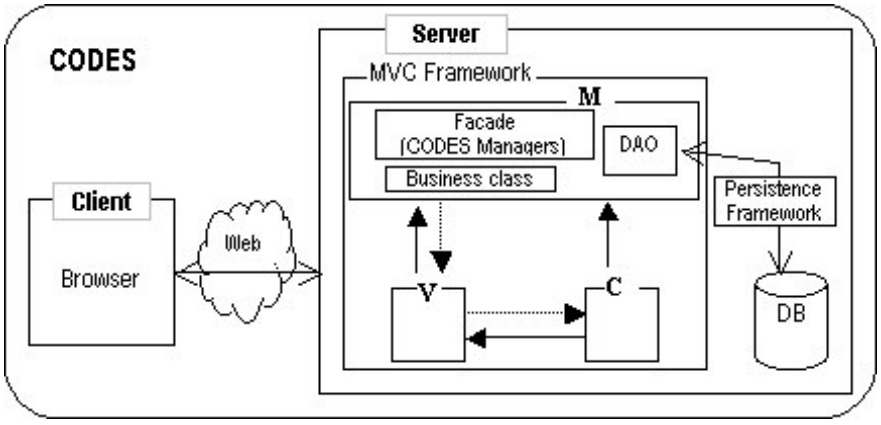


Figure 1: Architecture of the CODES environment

the participants' learning about music and music making, and that is exactly our main concern. It is not important to us, at the moment, the quality of the product (musical piece) that results from music prototyping with CODES. We are now more interested in enhancing the experience of music making, so that non-musicians may also have access to it, and we may get a better understanding of how this process works.

In CODES, a musical prototype consists of *Lines* (of instruments, arrangements, effects, etc. – see Figure 2) that can be edited. Editing is typically done by selecting sound patterns among many of the predefined patterns made available in CODES. Sound patterns are high-level musical structures (small sections of music files in MIDI format), which then make the processes of choosing sounds and prototyping easier. It will be made possible, in the near future of the project, for the user to edit the sound patterns, as a deeper level of composition.

A user can create more than one line, that is, someone can be the "owner" of more than one line (like user *Jerome* in Figure 2). By clicking the *"Play"* (>) button, the Sound Manipulation Manager starts the execution of all the lines enabled for playback (option *Mute* unselected). All patterns of the chosen lines, vertically grouped in the same timeline, are mixed and played, under complete user control, which can stop and restart at any time with usual control buttons (*Play, Stop, Forward, Rewind, Pause*).

User interaction, therefore, basically includes actions such as selecting sonic patterns, dragging and dropping them into lines (what is allowed through AJAX coding) and playing them, and combining them with other lines composed by his "partners" (other users) in the same music prototype. This combination can occur in different ways: overlapping (simultaneous playing), juxtaposition (sequencing), etc.

Cooperative music prototyping is herein defined as an activity that involves people working together in a musical prototype. Cooperation in CODES is asynchronous, since it is not yet necessary to manage the complexity of real-time events, if the present goal is just to support the development of musical prototypes. Users can access the prototype, do their experiments and write comments at different times.

In CODES, a musical prototype is initiated by someone, the "prototype owner". The prototype owner uses CODES to elaborate an initial musical prototype, and to ask the collaboration of other partners by sending explicit invitations (typically using an e-mail form from inside the system).

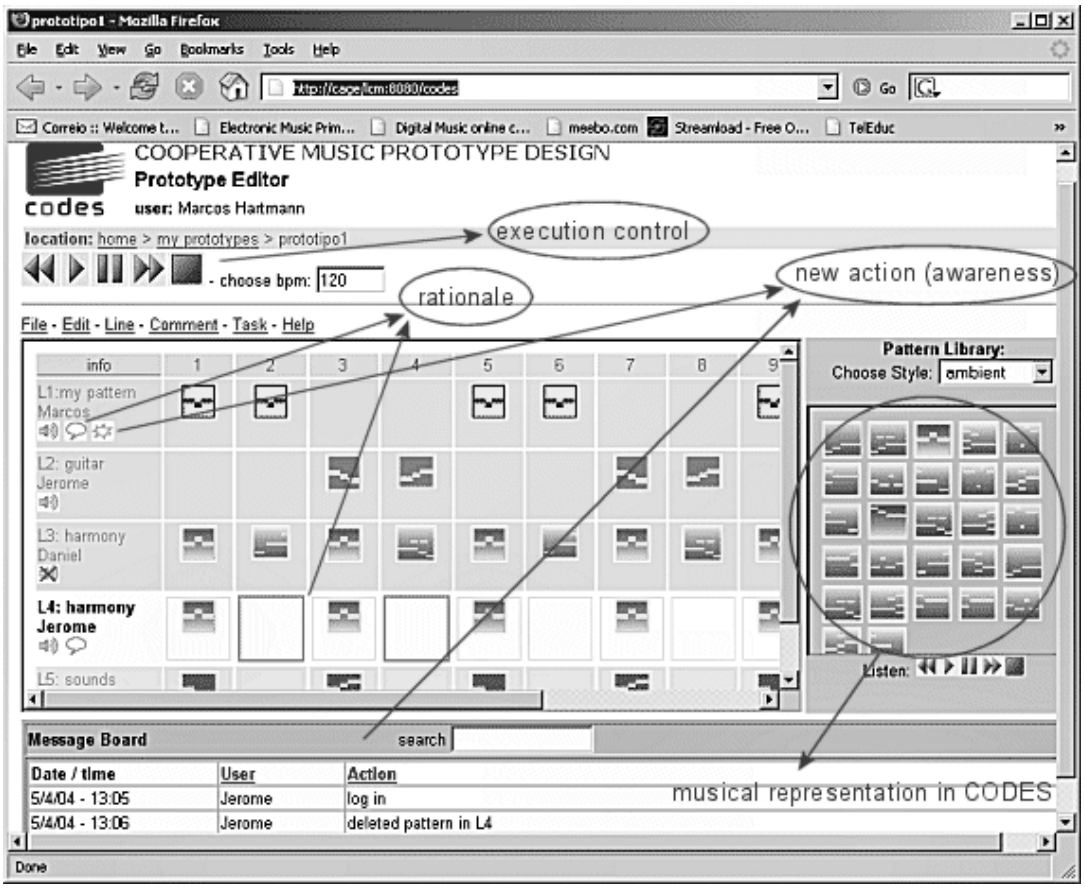


Figure 2: Elements of the CODES editing window

Partners who accept the invitation can participate in the collaborative musical manipulation and refinement of the prototype. The owner can also leave his prototype in an "open" space, in which interested users would discover it and then join the collaboration, as new partners. This way, the group of partners may evolve into a virtual community and, therefore, CODES may be classified as *communityware* [8].

For better support to cooperative musical activities, we propose in CODES three kinds of awareness mechanisms (see Figure 2) [17]:

1. *Music Prototyping Rationale*: to allow users to link their explanations with their actions on music prototypes;
2. *Action Logging*: to keep an explicitly recorded track of the steps that led to the current prototype state; and
3. *Modification Marks*: to indicate to a user that a prototype has been modified by others.

Actually, awareness mechanisms offer several advantages to music prototyping:

- keeping track of decisions;
- tracking progress in music prototyping and identifying conflicts, which may initiate negotiation processes between multiple points of view;
- supporting the construction of cumulative prototyping knowledge;
- assisting the integration of perspectives from multiple members of a group;
- "understanding" of each prototype, as there is no single answer or solution to a music prototyping problem.

CODES uses *icons* to represent musical information, as an alternative to the conventional music notation (score). Icons allow users to rely on both audio and visual clues more than on music theory to choose the right sound patterns. The idea here is to favor experimentation rather than theoretical knowledge. Each pattern can be individually listened to, before being selected and incorporated into a line on the prototype. Each icon traces its own sound pattern in a sort of "Cartesian plane", where the horizontal direction means duration of notes and the vertical means pitch variation. See an example in Figure 3, which shows the CODES notation and the correspondent musical staff.

This loose representation of pitch and duration is inspired in the early "piano roll" metaphor, largely used in music editing software. This way the user has a visual feedback of the sound even without listening to it in beforehand. In fact, we believe no previous musical knowledge should be required from any user to create music prototypes using CODES. The user does not need to know conventional music notation to create prototypes: he may select, play and combine such patterns in an interactive way, by direct manipulation and experimentation, without taking into account the formal representation format. However, the capability to convert from musical prototype to the score version is one of our next goals, in order to better support pedagogical uses and further music theory learning possibilities.

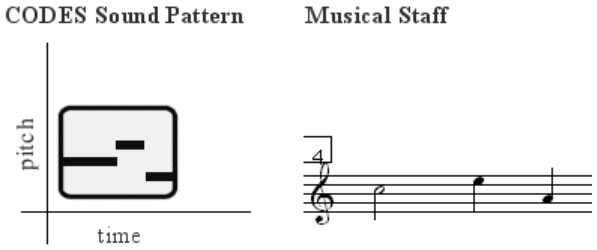


Figure 3: Musical representation in CODES

5 Beyond the open source music software

CODES is open-source from the beginning, since it is a scientific/academic research, conducted at a public university, and under public funding. But we want our project to take this quality further beyond the mere software that is being developed. As we saw in sections 1 and 2, requirements and design decisions such as reuse of code, free access and availability even to non-musicians, cooperative music prototyping approach, and alternative music encoding formats, all show that the CODES project is being entirely conducted under a free/libre/open-source philosophy. In other words, the system is being built to be itself a tool for open content production.

Sure there will be certain parts in such a project where achieving the desired freedom/openness will not be entirely possible. But we believe that just taking this open source notion as a principle is already important.

Going further with the free philosophy of our research, we are also studying *alternative licensing* options for all the intellectual products of this project: the software that forms the system; the sound patterns available in the system; and even for all music that will result from using CODES. These more flexible intellectual work licensing options are vital in the current global "information age" context where, according to Esther Dyson, "the Net dramatically changes the economics of content" [18]. Traditional copyright or intellectual property laws cease to be applicable, because they are too limiting, or they simply do not manage to survive.

Interesting alternatives may be those of the

Creative Commons initiative [19], and for the software, the CC-GNU-GPL, which is being suggested by government efforts in Brazil [20]. As far as it concerns the users, it will be made clear why they should "open their work to the world", at least to agree with the philosophy of the project. This discussion will be an important part of the CODES project website, contributing to increase general consciousness about the "free/libre/open-source work" issue, which is becoming more and more important in present times.

6 Conclusion

CODES software is still under development, but we have already conducted a number of small scale studies with a partially functional prototype, in order to evaluate its use, identify and correct problems, and determine new requirements. The most interesting result of this preliminary assessment was that, in addition to the "conventional" edit-listen-publish-cooperate-refine procedure for which we have initially developed CODES functions, users were able to find other creative applications for it:

- as an effective support for music learning;
- as an entertainment tool (DJ-like performance/experiments);
- as accompaniment system for human live performance.

This flexibility to accept new uses, together with several features to support the use by non-musicians, indicate both that we are applying a free/libre/open-source philosophy in all facets of the whole CODES project (not just in open-sourcing the software code), and that this notion can indeed be extended to other fields besides source code development. This discussion was our aim with this paper, along with presenting briefly the architecture, implementation and user interface details of the CODES environment.

We have shown throughout this paper that the open source notion can even be extended to the very main purpose of a music software, and equally to the music that will result from its use. In this case, developers must care to treat such music as "free music", or open content, and to take the proper measures to make it practicable (e.g. dealing with alternative licensing options).

By allowing non-musicians to have access to a musical creative experience, we are again practicing a free philosophy, since we are "freeing" users from the need to know music theory and to know how to play a musical instrument. As in open-source software philosophy, where source code is available to the public, we believe music should not be held exclusively by those who know the "secrets" (theory) of how to make it.

We also want to make clear that, despite focusing on non-musicians, we are not discarding the use of CODES by actual musicians too. In fact, it will be even of interest to assess, in the near future, what kind of uses that experienced musicians may find for this cooperative music prototyping environment.

Another aspect remaining to be explored is the implementation of real-time interaction in the CODES architecture. Interaction among users in a synchronous fashion may bring up new features and properties not considered until now.

7 Acknowledgements

This project is being partially supported by the Brazilian research funding councils CNPq and CAPES.

References

[1] A. Barbosa. Displaced soundscapes: a survey of network systems for music and sonic art creation. *Leonardo Music Journal* 13: 53–60, 2003. MIT Press, Cambridge, Massachusetts.

[2] *Organised Sound*, 10(3) [issue on Networked Music]. Cambridge University Press, Cambridge, UK, Dec. 2005.

[3] UFRGS Computer Music Research Group. *LCM – Computer Music Lab*. http://www.inf.ufrgs.br/lcm/, accessed in Dec. 2006.

[4] G. Weinberg. The aesthetics, history, and future challenges of interconnected music networks. In *Proceedings of the International Computer Music Conference* (Göteborg, Sweden, 2002). ICMA, 2002.

[5] R. Samudrala. *The Free Music Philosophy*. 1998. http://www.ram.org/ramblings/philosophy/fmp.html, accessed in Dec. 2006.

[6] *Free the sounds... and you free the music*. http://freethesounds.org/, accessed in Dec. 2006.

[7] W3C. *World Wide Web Consortium*. http://www.w3.org/, accessed in Dec. 2006.

[8] O. Liechti. Awareness and the WWW: an overview. *ACM SIGGROUP Bulletin* 21(3): 3–12, Dec. 2000.

[9] M. Good. MusicXML: an Internet-friendly format for sheet music. In *Proceedings of the XML Conference and Exposition* (Orlando, 2001).

[10] J. Steyn. *Music Markup Language*. http://www.musicmarkup.info/, accessed in Dec. 2006.

[11] P. Roland. The Music Encoding Initiative (MEI). In *Proceedings of the International Conference on Music Applications using XML* (Milan, Italy, 2002).

[12] OpenSymphony. *WebWork*. http://www.opensymphony.com/webwork/, accessed in Dec. 2006.

[13] The Apache Software Foundation. *Apache TomCat*. http://tomcat.apache.org/, accessed in Dec. 2006.

[14] Red Hat. *Hibernate*. http://www.hibernate.org/, accessed in Dec. 2006.

[15] MySQL AB. *MySQL: The world's most popular open source database*. http://www.mysql.com/, accessed in Dec. 2006.

[16] Dojo Foundation. *Dojo, the Javascript toolkit*. http://dojotoolkit.org/, accessed in Dec. 2006.

[17] E. M. Miletto et al. CODES: supporting awareness in a Web-based environment for collective music prototyping. In *Proceedings of the Brazilian Symposium on Human Factors in Computer Systems, IHC 2006* (Natal, Brazil, Nov. 2006).

[18] E. Dyson. Intellectual value. *Wired* 3(7): 136–141 and 181–185, Jul. 1995. http://www.wired.com/wired/archive/3.07/dyson.html, accessed in Dec. 2006.

[19] *Creative Commons*. http://creativecommons.org/, accessed in Dec. 2006.

[20] Technical Committee for the Implementation of Free Software in the Federal Government. *Portal – Free software licenses*. http://www.softwarelivre.gov.br/Licencas/, accessed in Dec. 2006. (In Portuguese.)

Audio on Linux: crashing into the 80/20 limit
Keynote by *Paul Davis*

It seems appropriate, five years after the first Linux Audio Conference, to review how much has changed, how much has stayed the same, and where we, as a community of developers and users, are going. In particular it seems time to talk about the way that a well known rule of software developer ("80% of the work takes 20% of the time, and the remaining 20% of the work takes 80% of the time") seems to characterize the current state of audio on Linux. This appears true for desktop audio, for music applications, for device drivers and more. This talk will review some of the highlights of the 80%, but will also talk about how this situation has occured, and what could be done to move forward.

Saturday: 11h00, H0104

Open Source as a Special Kind of Component-Based System Development
Keynote by *Steffen Evers*

Saturday: 11h30, H0104

Panel discussion “if (Linux Audio), then {...}, else {...}”
Moderated by *Stefan Weinzierl*

We have invited several speakers from various backgrounds, both from within the community and from outside of the community, to shed their light on Linux Audio, and the future. Topic of discussion are the opportunities and possibilities for Linux Audio in the industry, in research, in art and in education.

Saturday: 12h00, H0104

openSUSE JAD - Tutorials for installation and producing music
Michael Bohle and the JackLab Team

openSUSE JAD - how does it work? A user friendly installation with YaST and other tutorials from the Community with the JackLab Team,

- How to upgrade a standard openSUSE 10.2 installation to a fully functional professional Digital Audio Workstation in 3 easy steps using various sources including the JAD repository.
- A tutorial to show the advancement of linux audio and how to workaround things that are still problematical, for example the limited ability to use VST plug-ins or missing "total recall".
- We will implement a virtual music production environment with real instruments, software-based realtime effects and tone generators.
- Basics of the following software will be covered: Jack, Ardour2VST (and how to compile Ardour2 with VST support), energy XT2, Rosegarden, various VST plugins and how to get them working in Linux.
- The following will be covered in more detail: advanced Jack transport and routing, using several synced software applications at the same time.

Introduction of 1 CD JAD Install (JackLab Audio Distribution) - first developers release We want to present the first 1-CD install of the openSUSE 10.2 based JackLab Audio Distribution and freely give away this CD to all participants.

Thursday: 15h15-16h15 and Saturday: 15h45-17h45, H0107

Integrating Documentation, End-User Support, and Developer Resources using *.linuxaudio.org

Ivica Ico Bukvic, Robin Gareus and Daniel James

In an ongoing mission to consolidate online resources Linuxaudio.org is currently working on integrating LA* mailing lists as well as deploying a comprehensive end-user documentation, indexing, and developer support platform. With Dave Phillips' recent announcement to retire as the maintainer of the invaluable linux-sound.org online resource, this project has become the top-priority initiative at Linuxaudio.org. At this year's LAC we envision a talk to announce ongoing work at *.linuxaudio.org in hopes to kick off a workshop and discussion (developers) as well as present the results and user-aspects to a larger audience (users). This talk is primarily documentation-oriented and is intended to serve as a catalyst for a community workshop and discussion (doc-editors, web-devel, artists, etc.). It is our hope that the ensuing feedback as well as recruitment will assist in developing the next-generation online documentation, indexing, and support platform for delivering comprehensive and tightly integrated content, incorporating most of the current communication technologies and protocols (listserv, wiki, forum), and requiring minimal maintenance overhead.

Thursday, 17h30-18h30, H0107 (for developers) and Saturday, 14h30-15h30, H0107 (for users)

Buzztard Music Production Environment

Stefan Kost and Thomas Wabner

The buzztard [1] project aims to provide a free, open source music studio that is based on the concept of the windows only and closed source software buzz [2]. Characteristic for this software genre is that all audio is generated by virtual instruments. The buzz software is not really further developed, as the main developer has lost his source code. In comparison to buzz, we hope that our software will improve in usability and features in the future. To allow migration for buzz users, we are providing song-file import and buzz-machine (plugins) reuse.

Keywords tracker, virtual music studio, GStreamer.

Architecture The software uses GStreamer as a media framework. The main UI is build using Gtk+ and optionaly various other gnome technologies. The project consists of several modules, namely a core library, front-ends and extensions.

Status The project has released a first version in the end of October 2006. Version 0.2 release is scheduled for spring 2007.

References

[1] Stefan Kost et al. 2002-2006. Buzztard Music production Environment.
http://www.buzztard.org
http://en.wikipedia.org/wiki/Buzztard

[2] Oskari Tammelin. 1997-2000. Buzz (3rd Generation Tracker).
http://www.buzzmachines.com
http://en.wikipedia.org/wiki/Buzz_%28software%29

Thursday, 13h00-13h45, H0107

blue: a music composition environment for Csound

Steven Yi

blue is a music composition environment for Csound. It is written in Java and works anywhere a Java Virtual Machine version 1.4 or higher is available. By using blue's SoundObjects, Instruments, NoteProcessors, timeline, mixer, parameter automation, and other features, users are able to work quickly and intuitively to express and shape musical ideas. With the tools provided in blue, the user is able to take advantage of all of the features of Csound in an environment designed to make the compositional experience focused, productive, and enjoyable.

blue Homepage: http://www.csounds.com/stevenyi/blue

Thursday: 14h00-14h45, H0107

Firewire Audio on Linux
Pieter Palmers

Together with the "Firewire Audio on Linux" talk, this demo will provide an overview of the current status of Firewire based audio and music on Linux. We will demonstrate some firewire audio interfaces, illustrating what we currently can and can't do. We'll also show some more exotic features that make the Linux implementation stand out compared to their Windows/MacOS counterparts.

Thursday: 16h30-17h15, H0107

Stereo, Multichannel and Binaural Sound Spatialization in Pure-Data
Georg Holzmann

The goal of this workshop is to show how to position sound in space (stereo, multichannel and binaural). This should be done from a user point of view, without explaining the detailed mathematic behind the algorithms. Therefore existing and open-source implementations in Pure-Data will be used and explained.

Topics:

- stereo-panning methods
- vector based amplitue panning (VBAP)
- ambisonic
- binaural ambisonic and 3D room simulation

To all topics I will explain the handling of the Pd implementations and the advantages/disadvantages of the specific methods demonstrated on examples.

Friday, 13h30-14h30, LA Pool (H2038)

A Software-based Mixing Desk for Acousmatic Sound Diffusion
André Bartetzki

By commission of the Studio für elektroakustische Musik (SeaM) at the Musikhochschule in Weimar, I'm developing a software-based mixing desk for acousmatic performances. The software is written in the SuperCollider3 language and is therefore platform independant. Besides the necessary fast computer the hardware consists of multi-channel audio interfaces (+24 in- an outputs) and a large MIDI-controller with 24 faders, rotary knobs etc. as well as a USB key pad with assignable buttons. Within this project I try to overcome the usual difficulties with traditional analog or digital mixers, which are barely suited for the concert diffusion of tape music. Mixing desk are well structured to reduce a larger number of (mono) input signals to less outputs. But in concerts of electro-acoustic music we deal very often with less source signals (sometimes multi-channel) to distribute them to a larger number of loudspeakers. There are more than 48 speakers in some acousmatic setups, sometimes grouped together in order to get different layers of depth, height, un/directivity etc. Classical mixing desks have special purpose outputs (direct out, aux, groups) which are not flexible enough for acousmatic performances in terms of routing, accessability and controllability. The structure of this software solution reflects these aspects among other things through two new layers in addition to the ususal input and output channels: a layer of dynamically controllable routing matrices and a layer of in-between multi-channel outputs. This concepts allows the user to mix one-to-one, many-to-many, one-to- many and many-to-one routing and superimposing grouping principles according to the needs of the performance.

Saturday, 18h00-19h00, Tesla (before the concert)

From resistors to samples: Developing open hardware instruments using Arduino, Pure Data and Processing

Recursive Dog collective (Dolo Piqueras, Emanuele Mazza and Enrique Tomás)

The advent of Arduino, a simple open-source hardware system for data acquisition and prototyping, has made it possible for anyone to design musical instruments. No experience in electronics or programming is required and, more importantly, there is no need to invest large sums of money in commercial interfaces.

By combining Arduino with Pure Data, a free audio and visual programming environment in real time, we have the tools needed to make electronic musical instruments based on interacting with the physical world. Based on these technologies, but also as part of the Experimental Music Instruments (EMI) project, RecursiveDog has designed some complex instruments for live performance that you can download from our website (http://www.recursivedog.org).

In this workshop, the technical knowledge needed to design and produce electronic musical instruments will be taught, using the free open-source tools mentioned above: Pure Data, Arduino and EMI. RecursiveDog will show you how to become a digital luthier for some hours and design a small music instrument to perform with. Each one of the instruments and prototypes produced will be used in the Sunday's final jam session.

Thursday, 13h00-14h30, MacPool (H3014), at other times that the MacPool is open, work on the instruments can be continued.

Developing Shared Tools: a Researchers Integration Medium

Fábio Furlanete and Renato Fabbri

The liNICS Computer Music dedicated Linux distribution Case Study.

The liNICS Linux distribution started as a tool for a doctoral research experiment in Computer Music. After that, it has been widely used at the Interdisciplinary Nucleus of Sonic Communication (NICS), an institution based at the University of Campinas, Brazil. From the beginning of 2006, liNICS has allowed a considerable interaction between musicians, engineers and mathematicians by use and development at different levels of the system. Feature demand guided feedbacks and how-to requests made salient how useful it is to count with each other's knowledge. The result of this strong interaction between artists and researchers has been the production of multi-author papers, new tools learning, group study proliferation, music production, and the widening of interests. In such a way, it became clear the capacity of intensifying people's involvement with each other's work by the development of a shared tool, and how the open development format can be useful for an already established research institution. This workshop is meant to be a BoF session. The initial issue established by a liNICS experience overview, some technical procedures and the social phenomena involved.

Thursday, 15h00-17h00, MacPool (H3014)

Livecoding with SuperCollider

Alberto De Campo and Powerbooks Unplugged

Writing code as a performance style has become an underground trend (cf toplap.org). Among current interactive programming environments, SuperCollider3 and its extension library JITLib offer particularly elegant support for the expression of musical ideas as code, and rewriting in realtime. The ensemble *'powerbooks unplugged'* explores this approach in its most communal form: playing on unamplified laptops, sharing both sound events and the code that generates them, thus holding musical conversation by exchanging ideas. The workshop sessions will show the basic concepts; participants are encouraged to bring their own laptops, with a current install of sc3 and WLAN card. Participants are also invited to join the Live Coding session(s) on sunday.

More info: http://toplap.org, http://pbup.goto10.org

Thursday: 13h00-14h30, LA Pool (H2038) and Friday, 11h00-12h30, LA Pool (H2038)

,

Python for Sound Manipulation
Renato Fabbri and Fábio Furlanete

Python is an dynamically typed, interpreted and interactive programming language that uses automatic memory management and is able to easily import diverse libraries. Therefore, it is adequate for non-professional programmers willing to use diverse functionalities, such as usual multimedia computer users. For audio manipulation, there is the SndObj library, called upon Python by the PySndObj module, that integrates traditional audio tools such as filters, syntethizers and signal mixers. On the other hand, we have Numerical Python, that adds fast multidimensional array facilities to Python, and ctypes, a ffi (Foreign Function Interface) package for Python, that can wrap C libraries in pure Python. In that way, we can create and modify simple and clean Python programms for audio and music creation with basic SndObj library objects, and, progressively modifying the code, introduce array processing and the use of external libraries for audio manipulation. Besides learning how to use Python for synthesis and audio manipulation, the attenders will learn to load different file formats and output sound through Alsa and jack.

Friday, 11h00 - 13h00, LA Pool (H2038)

Canorus - a music score editor
Reinhard Katzmann and Matevž Jekovec

Canorus is a free next generation cross-platform music score editor. It could be called a sequel of a well-known KDE music score editor, NoteEdit. Canorus was founded by NoteEdit developers mainly due to NoteEdit 's poorly-designed bases (like lack of developers documentation). Canorus means sweet, rich, deep, warm, friendly and gentle sound and harmony. This is exactly what Canorus should present: A friendly user interface, deep and strong fundamentals, a warm welcome to newcomers and a rich set of features. The workshop is intended for developers and end users. It presents the current state of development, showing a demo of the program, last but not least presenting the roadmap for the further development process.

For more information about Canorus please read http://canorus.berlios.de/wiki/index.php

Friday, 13h30-15h30, MacPool (H3014)

Stochastic Composition with SuperCollider
Sergio Luque

After starting with a general introduction to SuperCollider3: the programming language and the server architecture; we will explore several ways of working with stochastic procedures to compose algorithmically and to synthesize sounds. We will make use of SuperCollider's powerful Patterns and Events framework to create musical structures by combining random distributions, tendency masks and Markov chains. Also, we will read through the code of the pieces that I will present during the Friday night concert. Note: this workshop is completely SuperCollider beginner friendly.

More info: http://www.sergioluque.com

Sunday: 11h00-13h00, MacPool (H3014)

Compiling Simulink Models as SuperCollider UnitGenerators
Martin Carlé and Sönke Hahn

Simulink is a widely used modeling language for general purpose simulations and systems design. It extends the basic matrix computational features of MatLab towards a compilable data- flow paradigm with a graphical interface similar to MAX or PD. Our demonstration will show how functional models and simulations within Simulink can be tuned for realtime execution and compiled as multichannel in-out SuperCollider UnitGenerators. It should be discussed how this links or presents an alternative approach to Faust (Functional AUdio STreams) and Q. Further, the SimPowerSystems Toolbox allows to simulate a variety of electrical circuits. We will demonstrate how simple vacuum tube models, tube-amplifier circuits and ancient ENIAC flip-flops are to be set-up, compiled and installed within an automated research and development circle. While Simulink is commercial software, sources and compiled UnitGenerators are not. Other than for MatLab in Octave there is no free counterpart we know of. Since Simulink is a "quasi industry standard" it offers enormous access to free (especially physical) models and signal processing knowledge. Our intents are to explore them and to provide an easy and effective way of using them for musical purposes. The whole development framework heavily depends on free and open source software (JACK, SuperCollider, Linux etc.) and our contributions to the processes will be published under GPL.

Thursday, 17h30-18h30, LA Pool (H2038)

Video Editing with the Open Movie Editor
Richard Spindler

The Open Movie Editor is a simple video editing software that integrates with the Jack Audio Connection Kit and provides synced editing with all audio applications that implement the jack transport control, like ardour for example. It's designed with usability in mind, therefore easy to use for the beginner, yet aims to be powerful enough for the amateur film and video artist. The demo will start with a comprehensive yet compact overview of the available features and how to use that functionality. It will conclude with a hands on experience and allow the audience to make its first steps with the software. There will also be room for potential users to propose features that they would like to see in a video editing tool for linux.

The Homepage of the software is http://openmovieeditor.sourceforge.net/

Friday, 14h30-15h30, LA Pool (H2038)

Faust Hands On Demo
Yann Orlarey and Albert Gräf

This hands on demo will give the audience the opportunity to discover and practice Faust (http://faust.grame.fr): an easy and powerful audio programming language. It should be of interest for ALSA and Jack developers but also for users of existing languages like PD, SC or Max.

As we will see, Faust offers an interesting alternative to C for the development of high performance audio applications and plugins. Faust is a very expressive language, programs can be typically 100 times shorter than the equivalent C programs. It is the first audio language to be fully compiled. Faust programs are translated into highly optimized C++ code. This code works at sample level without any overhead and can compete with hand written C code in terms of efficiency. Finally Faust offers an easy deployment on multiple platforms. From a single Faust program, the compiler can generate native implementations for Alsa, Jack, LADSPA, SC, PD, VST, MAX,...

As we will also see during this hands on demo, Faust is not aimed at replacing existing music languages, but at offering a useful complement to them. We will show how easy it is to produce native plugins for various architectures. In particular we will demonstrate the production of PD plugins and patches thanks to the powerful faust2pd utility (http://q-lang.sourceforge.net/examples.html#Multimedia).

Saturday, 16h00-17h00, LA Pool (H2038)

Technical tour of the T-Labs
Sascha Spors

In collaboration with Berlin's Technical University (TU Berlin), Deutsche Telekom Laboratories have been established on the TU Berlin campus. Our mission is to conduct pioneering research into innovative information and telecommunications technologies for the modern marketplace. Behind the scenes, Linux Audio is being used for the research and development of audio applications, such as binaural technologies and wave field synthesis.

Friday: 13h00 till 15h00, meeting point 13h00 at the Info desk. Maximum of 30 persons can attend

Wave Field Synthesis compositions

For the Wave Field Synthesis system in lecture hall H0104, four compositions have been prepared, which were played in a loop on all days of the conference.

East (from *Atlas*) (2007) 17:00
Christian Calon

Based on an idea of cartography, the Atlas project, in its final form will be a *spatial installation*. Musically, it is an hommage to man's creative enterprise which consists into probing the unknown with the help of sound making instruments and then to turn ephemeral impressions into imperishable creations.

This part of Atlas, *East*, was realized first as a concert version. The composition and the spatialization followed the concept of soundfield, which a WFS excels at reproducing. All sounds were generated as correlated multitrack objects to be placed and spatialized as small constellations in this larger sound field, in order to create a multi-directional space for the listener.

The music is based on the sounds of traditional music instruments from the Far Eastern regions of the Earth.

The realization of East was made possible with a commission from the Inventionen Festival/DAAD, Berlin, and the Canada Council. I am very grateful to these institutions whom I warmly thank for triggering and supporting this stage of the large Atlas project.

In its final spatial installation form, Atlas will stage several parallel maps:

- in sound, it is a tribute to the creative enterprises of man in probing the unknown with the help of his musical instruments
- as a silent and virtual monument, the projected *bodyscape* images will present as another cartography, a surround topography of skins and faces of mankind in its richness and diversity
- at the same time projected texts appearing on or around the bodyscapes, will draw and list an underlying world map of *Infamies*, acts of hate, violence and power, perpetrated by man on his fellow man.

Biography

His first works emerged in Canada and soon brought him international attention. In 1989-90 he acted as vice-president for the CEC. In 1991 he was appointed to the musical direction of the GMEM (France) and in 1995, as a guest of the DAAD, he went to Berlin where he lived for several years.

His concert works, sound installation or radio projects have all in common the exploration of the listening experience. The conception of sound shapes projection and the importance of listening contexts are at the heart of his creative research leading to a on-going process of investigation of new technologies. In parallel, he pursues his reflection on the narrative forms through writing and composing for the radio medium.

His work is performed worldwide and received honors in major international competitions: 2006 - Prix Opus, Quebec, 2004 - Represents Canada at the World New Music Days (Switzerland), 2003 - Distinction at the International CIMESP competition (Brasil), Selection of the Confluencias International Competition (Spain); 2001 - Grand Prix Phonurgia Nova International (France); 1999 - Grand Prix Marulic of the UER/EBU (European Broadcasting Union); 1997 - Distinction at Prix Ars Electronica (Austria); 1996 - Lynch-Staunton Prize, Canada Council (Canada); 1995 - Distinction at Prix Ars Electronica (Austria); 1995 - Berlin DAAD guest (Germany); 1994 - 2nd Prize, Bourges International Competition (France); 1991 - 2nd Prize, NEWCOMP International Computer Music Competition (USA); 1989 - 1st Prize, Bourges International Competition (France); 1989 - Canada representation at the World Music Days ISCM (France); 1988 - 2nd Prize, NEWCOMP International Computer Music Competition (USA); 1985 - 1st Prize, Luigi Russolo International Competition (Italy)

His first solo CD « Ligne de vie » (IMED 9001) was proposed for the 1990 Grammy Awards (USA) and the second CD « Les corps éblouis » (IMED 9838) was nominated for the album of the year at the 1998 Opus Awards (Canada). His music is published on the Empreintes DIGITALes label (Montreal) and also appears on various labels (coming:: The Ulysses project, surround DVD).

A free-lance artist, he now lives in Montreal.

Part of the WFS-Loop

Rituale (2004) 15:00
Hans Tutschku

The 15 minute piece “Rituale” (2004) processes human voices and instrumental sounds from various cultures to a sound ritual. It is a continuation of the work on “Rojo” and “object-obstacle”, which were both also concerned with the theme of rituals. The composition uses extensively the possibility to place sound sources between the speakers and the listeners - and so inside the listening room. In this way the sounds come very close to the listeners.

“Rituale” was originally (in 2004) created for the IOSONO Wave Field Synthesis system in the Ilmenauer Lindenkino, and was adapted for the lecture hall of the TU Berlin in 2007.

Biography Member of the "Ensemble for intuitive music Weimar" since 1982. He studied composition of electronic music at the college of music Dresde and had since 1989 the opportunity to participate in several concert cycles of Karlheinz Stockhausen to learn the art of the sound direction. He further studied 1991/92 Sonology and electroacoustic composition at the royal conservatoire in the Hague (Holland). 1994 followed a one year’s study stay at IRCAM in Paris. He taught 1995/96 as a guest professor electroacoustic composition in Weimar. 1996 he participated in composition workshops with Klaus Huber and Brian Ferneyhough. 1997-2001 he taught electroacoustic composition at IRCAM in Paris and from 2001 to 2004 at the conservatory of Montbéliard. In May 2003 he completed a doctorate (PhD) with Professor Dr. Jonty Harrison at the University of Birmingham. During the spring term 2003 he was the "Edgar Varèse Gast Professor" at the TU Berlin. Since September 2004 Hans Tutschku has been working as composition professor and director of the electroacoustic studios at Harvard University (Boston). He is the winner of many international composition competitions, among other: Bourges, CIMESP Sao Paulo, Hanns Eisler price, Prix Ars Electronica, Prix Noroit and Prix Musica Nova. In 2005 he received the culture prize of the city of Weimar.

Part of the WFS-Loop

Streams (2007)
Victor Lazzarini

Streams is a multi-dimensional woodwind quartet, set in 3-D physical space and in the several dimensions of frequency-space. The piece is roughly composed of three overlapping sections, starting with a slow continuous-sound part where each of the four instruments get split in two and glide around the space eventually condensing back together. The middle section starts with chord-forming lines that are spun around the space, continuing into brief statements of melodic motives that eventually lead to interlocking ostinatos. These accellerate to an impossible tempo and then slow down to the original speed. The third section is dominated by an ever-ascending canon (by tone), played by slightly un-synchronised parts. A final coda is based on the ideas/texture of the first section. The piece was composed mostly using software specially developed by the composer for spectral and time-domain processing. This wavefield-synthesis version is dedicated to the TU-Berlin WFS team: Marije, Simon, Torben, Thilo, Daniel and Eddie.

Biography Victor Lazzarini is a composer and researcher working mainly in the area of Computer Music. A graduate of the Universidade Estadual de Campinas (UNICAMP) in Brazil, he completed his doctorate at the University of Nottingham in 1996. His past musical activities also included movie soundtrack composition and performance as jazz pianist and arranger. He is currently a Senior Lecturer at the Music Department, NUI Maynooth, Ireland, where he also directs the Music Technology Laboratory.

Part of the WFS-Loop

Reale Existenz! (2007)
André Bartetzki

This piece is based on short fragments of a lecture by the Austrian physicist Schrödinger. Erwin Schrödinger, one of the inventors of quantum physics, got very popular due to his thought experiment with a cat in a closed box in which he tried to illustrate the superposition of quantum states. Coupled to the state of a decaying atom (via a Geiger counter and a flask of acid) the cat is after a while both dead and still alive according to the superposition of the two possible states of that unstable atom. Only a collapse of the wave function of this system - caused by an observer or by the influence of the macroscopic environment - could the cat release of its indecisive state.

Biography André Bartetzki was born in Berlin in 1962. After a professional training and a few years of working as sound technician at the East-German state broadcast station and recording studios he studied sound engineering at the Hochschule für Musik "Hanns Eisler" in Berlin. During his studies, he began to set up a studio for electroacoustic music at the Hochschule, and between 1992 and 2002 he has lectured there and directed the studio. He has also given lectures and workshops in sound synthesis and algorithmic composition at the Technical University in

Berlin, the Bauhaus-University Weimar, the Hochschule fuer Musik und Theater Rostock, the KlangArt festival in Osnabrück and at the Academy of Arts in Berlin. Between 1999 and 2004 he worked at the electroacoustic studio at the Musikhochschule "Franz Liszt" and at the Media Arts faculty of the Bauhaus-University in Weimar.

Besides teaching, he works frequently as a programmer, sound designer and sound engineer with ensembles, soloists and composers of new music. His software CMask for algorithmic composition is being used by many composers around the world.

Since the mid-nineties he has been developing and performing his own musical and media art projects: tape music, performances with live-electronics, video and sound installations. His works were performed at international festivals for contemporary and electroacoustic music such as the Kryptonale Berlin, the ICMC 2002 Gothenburg, the BIMESP 2002 Sao Paulo, the SICMF 2003 and 2004 in Seoul, the ACMC 2005 in Brisbane. He became Finalist at the Bourges Festival and at the CIMESP Sao Paulo 2001. In 2004 he received a commission for the European Bell Festival by the ZKM Karlsruhe.

Between 1997 and 2004 he was a member of the board of the German Association for Electroacoustic Music (DEGEM), where he has worked as the editor of the DEGEM newsletter.

Part of the WFS-Loop

MODES OF INTERFERENCE / 3
Agostino Di Scipio

A small network of electric guitars and amplifiers, left free to resonate from high-gain feedback. A computer handles this feedback system, preventing sustained saturation and soliciting various system resonances. The process remains subject to perturbations from the environment (room, court, or else) mediated by the body of the guitars and the strings. Overall, this work is also a comment on the electric guitar, today little more than a torn-out, consumed pop-culture icon, whose elementary phallic symbolism is repeated myriads of times every day around us. But especially, the work proposes a kind of deconstruction of the electric guitar sound: no violent act of performing is staged here to let it come into existence, no dramatic gesture ("guitar hero") is needed to let the feedback system regulate its own unfolding in time.

First presentation took place in the XVII century court of the Conservatory of Music of Naples, 23.02.2007.

Biography Agostino Di Scipio has been living in L'Aquila since 1985. Composer of a variety of sound works, including tape music, sound installations and music scored for instrumentalists (soloists or ensembles) + interactive computer systems. Many of his compositions develop from unconventional sound synthesis/processing methods inspired to phenomena of noise and turbulence, and recently focus on the "man-machine-environment" feedback loop (for instance his live-electronics solos titled Ecosistemico Udibile). Currently full time Electronic Music Professor at the Conservatory of Naples, and instructor in live electronics at Centre Creation Musicale Iannis Xenakis (CCMIX), Paris. A former "visiting faculty member" at the Dept. of Communication and Fine Arts of Simon Fraser University (Burnaby-Vancouver, 1993), and "visiting composer" at Sibelius Academy Computer Music Studio (Helsinki, 1995), in 2004 Di Scipio was artist-in-residence for the DAAD Berliner Künstlerprogramm.

Thursday: 14h00-17h00, Friday, Saturday, Sunday: 12h00-17h00, H3021

Command Control Communications
Hanns Holger Rutz and Cem Akkan

Command Control Communications is an interactive installation with video by Cem Akkan and noises by Sciss. One recipient at a time can choose the “categories” of movies to be shown. Brief loops of commerical hollywood movie advertisement are used in a junk-art or arte povera fashion: making an aesthetic object out of trash. We are playing with the stereotype point of view exhibited in practically all of the trailers, which is exaggerated by using a very low image quality with typically eight frames per second, by sorting them in partly odd categories and mainly by confronting the viewer with four simultaneous images.

While produced solely on the laws of entertainment of the mass taste and capitalist profit maximization, the themes and stereotypes clearly reflect the sickness of a society which is the most violent in the western world. The excessive use of violence and reference to the Christian symbolics (also reflected in the categories of “evil” and “mystery”) seem to feed an already existing paranoia. When it comes to the “messages”, a surprising congruence with the political debate as transported by the media democracy is revealed. While it is generally acknowlegded that countries at war tend to have a cinema that serves as a psychological aid or distraction for the people, we rather gain the impression of a permanent war.

Sciss aka Hanns Holger Rutz (b. 1977) began to create electronic and noise music around 1999. Between 1999 and 2004, he studied computer music at the Electronic Studio of the Technical University Berlin. Since 2004, he

is research associate at the Studio for electroacoustic Music (SeaM) in Weimar. His electroacoustic works include CD-albums (e.g. “Residual” 2002) and E.P.’s (e.g. “Crosshatch” 2004), multichannel concert pieces (e.g. “Strahlung” 2006), works with video (“Achronie” 2002, with video by Cem Akkan), and installation pieces (“Zelle 148”, Erfurt 2006, “Rigid String Geometry” Weimar/Berlin 2006, among others). Live-electronic works encompass solo performances, varying duo projects, the ChromaticField trio, and the quartet Hennig/Markowski/Sciss/Sienknecht. Since 1999, development of audio software. In 2003 co-organizing the Salon Bruit platform for experimental music.

Cem Akkan, born 1959 in Istanbul, Turkey. Occupation: photography, recording engineer. Studied communication sciences in Vienna, lives in Berlin and New York since the early 1990s. Studied audio engineering in NYC. Various projects involving video and sound.

Thursday: 14h00-17h00, Friday, Saturday, Sunday: 12h00-17h00, H3008

fijuu
Julian Oliver and Steven Pickles

fijuu is a 3D, audio/visual installation. Using a PlayStation-style gamepad, the player(s) of fijuu dynamically manipulate 3D instruments to make improvised music. fijuu is built using the open source rendering engine OGRE and runs on Linux. in the future fijuu will be released as a Linux live CD project, so players can simply boot up their PC with a compatible gamepad plugged in, and play without installing anything (regardless of operating system). This effectively turns the domestic PC into a console for game based audio performances.

http://fijuu.com

Julian Oliver is a New Zealander, free-software developer, composer and media-theorist based in Berlin, Germany.

Julian has presented papers and artworks at major international electronic-art events and conferences worldwide and, under the moniker ’delire’, has performed game based electroacoustic works at prominent venues throughout North America, the EU, Japan and the South Pacific. Julian has given numerous workshops in game-design, independent game-engine development, virtual architecture and open source development practices worldwide. Julian’s work in games began in 1998 with the modification of popular 3D shooter engines in an effort to bring disciplines of architecture and computer music to game design.

In 1998 Julian founded the game-based media laboratory Select Parks, which currently hosts over 120 game-based artworks and developer resources.

Steven Pickles aka ’pix’ is an Australian artist, programmer and free software developer currently based in Berlin, Germany.

He has collaborated on numerous projects with groups such as farmersmanual, FoAM, Select Parks and meso. Steven completed his B.Sc in Computer Science at the University of Adelaide in 1999 and his generalist approach has lead to research and experimentation in numerous technical fields including sound sythesis, computer animation, visual programming, physical interfaces and embedded programming.

Thursday: 16h00-17h30, Sunday: 11h00-14h00, H2038

Yue

The Yue project started about three years ago in Reggio Emilia, Italy, and it’s composed by four musicians (Daniele Torelli, Luca Piccinini, Luca Bigliardi, Sara Menozzi) and a video artist (Andrea Bagnacani) who want to use free software only. Our activity is mainly live-oriented, and in the last two years we played many concerts all over Italy ranging from discos to outdoor festivals to pubs, and at the LAC2006 in Karlsruhe. The software parts of our work is realized with free software running on Debian GNU/Linux systems, the whole live setup includes Seq24, Ghostes, DSSI plugins like Whysynth and Xsynth, samplers like Fluidsynth-DSSI and Specimen, Ardour, many LADSPA effect, Freej and Effectv for the video part. We also use “normal” instruments, such as guitars, keyboards and Sara’s beautiful female voice. Our music is all self-made and available under the Creative Commons Attribution-Share Alike license.

Thursday: 21h00, C-Base

Video Piece
Jim Hearon

"Above-Under" is a free interpolation of the experimental film "Oben/Unten" (1967) by Luz Mommartz. The experimental black and white film is available in the Prelinger Archives, an original collection of over 60,000 "ephemeral" (advertising, educationl, industrial, and amateur) films, several of which are available via the current internet streaming and downloading video site: «http://www.archive.org/details/prelinger». The original soundtrack was by the ICENI, a

five-man group who are also shown in the film. My version, called "Above-Under", colorizes and processes the video, and employs a new original digital soundtrack. In several instances the audio is created by the video, and at other times it is a freewheeling improvisation in the spirit of the original soundtrack. I did not use, sample, or otherwise employ the original soundtrack in anyway. I was also inspiried by a group of my students who are lately into glitch music, and wanted to incorporate some of those aspects into the music.

Biography Jim Hearon is an improvisor, and electronic violinist working in multimedia and interactive graphical environments. After working in the music industry for many years, and teaching part-time at a number of schools, Jim moved to Honolulu in 2005 to work at the University of Hawaii at Manoa. Jim is an avid long board surfer, frequently in the Pacific Ocean at Waikiki Beach, and is married to Yuki Horikiri, a Japanese jazz pianist working with Educational Technology.

Thursday: 21h00, C-Base

Life coding over live coding
xxxxx

Life coding over live coding. Hardware and software construct active and highly audible circuits open to visible re-configuration. Dramatic CPU and software acts are rendered brutally evident within constructivist, process performance; an open laboratory. The symphonic rise of the attempt to piece together fugal systematics is played out against the sheer noise of collapse and machine crash within a deserted borderland of control.

Biography

ap/xxxxx was founded by Martin Howse in 1998 to necessitate the code-terms expansion implied by a growing and politically active free software movement. With wilfully avant-garde intent, and through intervention, performance, staged events, seminars, hardware constructions and readily accessible software, ap interrogates in live descriptive process software culture and history. Software is viewed as substance. Performances, lectures and large-scale curated events range across international venues and festivals with further projects elaborating networked environmental code-creation machines, a language for the streamed entry of endless cinema and the development of promiscuOS, a totally untethered and highly promiscuous operating system.

Thursday: 21h00, C-Base

faltig
Frank Barknecht

FrankBarknecht is my name. I like PureData.

http://footils.org - http://goto10.org

Thursday: 21h00, C-Base

Linux Cound Night - Plug 'n' Chill

As usual the club concert is concluded with a Plug 'n' Chill session, where you can plug in your laptop and join in with the others to play.

If you want to participate, please let us know at the info desk.

Thursday: 21h00, C-Base

"De la incertidumbre" for computer (2005)
Sergio Luque

Translation: (From the uncertainty)

I.- Y fue que le pareció convenible y necesario (And that was that he fancied it was right and requisite) - Duration: 9:28

This piece is loosely based on the first chapter of Don Quixote:

"In short, his wits being quite gone, he hit upon the strangest notion that ever madman in this world hit upon, and that was that he fancied it was right and requisite, as well for the support of his own honour as for the service of his country, that he should make a knight-errant of himself, roaming the world over in full armour and on horseback in quest of adventures, and putting in practice himself all that he had read of as being the usual practices of knights-errant; righting every kind of wrong, and exposing himself to peril and danger from which, in the issue, he was to reap eternal renown and fame. Already the poor man saw himself crowned by the might of his arm Emperor of Trebizond

at least; and so, led away by the intense enjoyment he found in these pleasant fancies, he set himself forthwith to put his scheme into execution."

II.- Interludio - Duration: 11:28

An interlude (only synthetic sounds were used in this piece).

III.- ¿Qué gigantes? - Duration: 6:00

This piece takes as its point of departure the concept of uncertainty present in the book Don Quixote. Uncertainty about all things, nothing is sure nor permanent.

In this book, the events, the persons and the objects are perceived in very different ways by all the characters: Milan Kundera affirms that this book has no characters at all, but egos.

"Fortune is arranging matters for us better than we could have shaped our desires ourselves, for look there, friend Sancho Panza, where thirty or more monstrous giants present themselves, all of whom I mean to engage in battle and slay, and with whose spoils we shall begin to make our fortunes; for this is righteous warfare, and it is God's good service to sweep so evil a breed from off the face of the earth. 'What giants?' said Sancho Panza."

Biography Born in Mexico City in 1976. He is currently pursuing a PhD in Composition at the University of Birmingham. He received a Master's Degree in Sonology (with distinction) at the Royal Conservatory in The Hague, studying with Paul Berg and Kees Tazelaar. He received a Master's Degree in Composition from the Conservatory of Rotterdam, studying with Klaas de Vries and René Uijlenhoet. He has a Bachelor's Degree in Composition from the Musical Studies and Research Centre (CIEM, Mexico), and has been admitted Associate in Musical Theory, Criticism and Literature by the Trinity College London.

His music has been performed in Mexico, Germany, the United Kingdom, the Netherlands, France, the United States, Chile, Spain, Australia, Switzerland and Cuba.

Thursday: 21h00, Cervantes

RecursiveDoor

Recursive Dog collective (Dolo Piqueras, Emanuele Mazza and Enrique Tomás)

Aldous Huxley´s The Doors of Perception was first published in Great Britain in 1954. The book was inspired by William Blake´s words If the doors of perception were cleansed everything would appear to man as it is, infinite. For man has closed himself up, till he sees all things through' narrow chinks of his cavern. Huxley´s book is considered to be one of the more profound studies of the effects of mind-expanding drugs and what they teach about how the mind works. As in Huxley´s essays, RecursiveDoor functions as an examiner and critic of social mores, societal norms and ideals concerning art perception using real time generative audio and visuals controlled by our non- traditional interfaces. Using Huxley´s words, in RecursiveDoor space and dimension become irrelevant, and perceptions seem to be enlarged and at times even overwhelming.

Recursive Dog is a Spanish artists collective providing recursive ideas and hacktivism around generative art and free culture. Recursive Dog uses open source software and hardware like Arduino, Processing and Pure Data or Csound to develop genetic structures that also depend on the sound activity produced by our performance or by the audience. You can also join us in our open workshops and become an electronic gypsy like us.

More info: http://www.recursivedog.org

Thanks to Facultad de Bellas Artes de Valencia, Laboratorio de Luz and Facultad de Bellas Artes de Cuenca, Art Department, IndEvol Group

Friday: 19h00, Cervantes

CYT (2007) 8:00 / DUX (2006) 7:45 / TAU (2005) 5:40

Edgar Barroso

CYT Cyt is the greek root for cell, the human body is a system made up of discrete organs and tissues, however, individual cells that compose these essential tissues are often short-lived. The skin covering our body today is not really the same skin we had a month ago. The piece follows this principle in the sense that it is constructed with "sound cells" that have specific functions and are constantly regenerating to sustain "life". The sounds are classified depending on the potential of self-renewal, proliferation potential and the degree of differentiation. In addition, the numerical balance of this sounds, will determine the way in which musical ideas interact, and the variations that could emerge if that fragile balanced is changed.

DUX From Latin "leader", four sounds conforms the leaders of their legions of sub-sounds. The piece begins with a single sound, that "hauls" to a series of others that are forming and connecting the music ideas throughout the piece. The form is determined by this same criteria, is a piece divided in four clear sections, in which each dux sound has a protagonist role, these sounds are formed by two recordings, cello and contact microphone and the other two

are generated by synthesis. The creation of sub-sounds are mainly done by re-synthesis and audio processing changing their location within the space.

TAU The piece is based on the metaphoric idea of one of the six components that form leptons, called Tau. This particle, in spite of being considered as a lepton, that literately means "slight mass" has more than three thousands times more mass than an electron. From that idea I wanted it to make an analogy with the sounds, to represent isolated elements that can exist with no need of other particles, but that will possibly have much more weight than another phenomena, which at first, would seem dominant, and perhaps represent sound particles. (Not every thing is the way it "sounds").

Biography Born in Mexico in 1976. Edgar Barroso is at the present time a composer in residence at the PHONOS Foundation. His education includes a Master in Digital Arts, a Postgraduate Diploma in Composition and Contemporary Technologies and a Bachelor in Music Composition. His music has been interpreted in important forums in North America, Ibero-America, Asia and Europe. He recently gained the 1st Grand Prize of the Harvard University Live Electronics International Composition Competition (USA), and is a finalist of the International Composition Competition "Ensemblia 2007" (Germany). In addition he is dedicated to instrumental practice as a cello player, exploring diverse techniques of improvisation with/out live electronics.

Thanks to PHONOS Foundation, MTG Music Technology Group, Universitat Pompeu Fabra

Friday: 19h00, Cervantes

Kitchen <-> Miniature(s)
Fernando Lopez-Lezcano

A good quality sound recorder and a kitchen. Humanity tuned to common shapes and sizes that create shared resonances I have come to recognize everywhere there is a kitchen. These tightly chained miniatures explore a few of the many kitchen utensils and small appliances that I recorded (that is, anything that would fit with me inside my bedroom closet). Featured prominently through the piece is the mechanical timer of a toaster oven, as well as cookie sheets, plates, trivets, the klanging sound and inner resonances of the lid of a wok and many more kitchen instruments. More than 3000 lines of Common Lisp code are used to create large scale forms and detailed sound processing. Without Bill Schottstaedt's CLM (Common Lisp Music), Juan Pampin's ATS (Analysis, Transformation and Synthesis) and Rick Taube's Common Music this piece would not have existed. Grani (a granular synthesis software instrument) and other old software friends I have created over the years helped as well.

Biography Master Degrees in Electronic Engineering (Faculty of Engineering, University of Buenos Aires, Argentina) and Music (Carlos Lopez Buchardo National Conservatory, Buenos Aires). He has been working in the field of electroacoustic music since 1976 (instrument design, composition, performance). Has also worked in industry for almost 10 years as a microprocessor hardware and software design engineer for embedded real-time systems, and taught computer music at Keio Universiy, Japan. He created and maintains the Planet CCRMA at Home Open Source package collection of music and sound applications for Linux systems. Currently keeps computers and users at CCRMA happy (most of the time), teaches courses at CCRMA, makes music when time allows, and enjoys the company of good friends. His music has been released on CD and played in the Americas, Europe, and East Asia.

Friday: 22h00, Tesla

schnitt//stelle
Orm Finnendahl

Version 2.0

In the cooperation with the ensemble Mosaik a composition for flute, oboe and piano will be created. This cycle explores the possibilities of improvisation and composition in the context of the current state of the art of computer technology. The sound material from the musicians is automatically transformed in various ways by the computer. These transformations are realised in realtime, so the musicians can react to these transformations and influence them.

An important aspect of the collaboration with the ensemble is the procedural creation process, in which the musicians can experiment on their own with the programs that have been developed and optimised for them. The results of these experiments then influence the computer programs during the various stages of the rehearsals, so that the end result is developed in a process of getting closer to each other. This way of working is prompted by the way of development of open source, as seen in the creation of software, as well as in projects like the online lexicon Wikipedia.

Biograpy Born in Düsseldorf in 1963, Orm Finnendahl studied Composition, Musicology and Computer Music in Berlin after some involvement in the Berlin experimental music scene. 1988/89 scholarship at the California Institute of the Arts in Los Angeles. 1995-98 continuing studies with Helmut Lachenmann in Stuttgart. Collaborations with ensembles specializing on contemporary music (Ensemble Modern, recherche, Mosaik, Champ d'action, etc.) as well

as with video and multi media artists, dancers and soloists (Palindrome, AlienNation, Burkhard Beins, etc.). Numerous awards and prizes, among them Kompositionspreis Stuttgart, Busoni Prize Berlin, CYNETart Award Dresden and Prix Ars Electronica Linz. A portrait CD for the "Edition Zeitgenössische Musik" of WERGO Records is in preparation.

Currently Orm Finnendahl is Professor of Electronic Composition and Head of the Electronic Studio at the Musikhochschule Freiburg.

Friday: 22h00, Tesla

Strahlung (2006) 9:54
Hanns Holger Rutz

Strahlung (german for radiation) is a kind of atmospheric landscape that - albeit not conceived as program music - borrows from the uncanny immateriality of radiation. It picks up the occasional but recurring dream motif of radioactive contamination. The Czernobyl catastrophe twenty years ago is stuck in my mind as a psychic object, a lucid mnemonic island, and is connected to a paradoxical simultaneity of both attraction by and fear of an invisible nature - or a kind of animism as is described for example by the Strugatzki Brothers and in Tarkowskij's "Stalker".

Tape music 8-channels, studio: SeaM, Weimar.

(*See the installation "Command Control Communication" for a biography*)

Friday: 22h00, Tesla

North (from *Atlas*) (2006) 15:00
Christian Calon

The first part of Atlas, North is presented today in a concert version. The music is based on the sounds of traditional music instruments from the north-western regions of the Earth, pre-dominantly Europe, the Arctic, North America and the Near-East. The realization of North was made possible with a commission from Sonic Arts Network and a residency at SARC. I am very grateful to both organizations whom I warmly thank for triggering and supporting the first stage of this large project.

(*See also "East" in the WFS-Loop*)

Friday: 22h00, Tesla

"Expression" for 8 loudspeakers (2006) 11:30
André Bartetzki

"Expression" is the title of the last piece on the last record of the same name that John Coltrane has recorded together with his quartet shortly before he died in 1967. This title was chosen by himself. When he was asked to add explanatory texts about the pieces on that album he answered: *"with absolutely no notes. (...) By this point I don't know what else can be said in words about what I'm doing. Let the music speak for itself."*

(*See "Reale Existenz¡' in the WFS-Loop for a biography*)

Saturday: 19h00, Tesla

"Gebrochene Klanggestalten" for 4-channel tape (2004) 4:20
Weiwei Lan

The German title "gebrochene Klanggestalten" means "broken sound -figure". This composition processes sound materials from the everyday life: sound of a slamming door, howling vacuum cleaner, rubbing sounds generated with a tea-ball on a cooking pot. Using transformation and montage-technical sounds become sound-figures, which are fractured in multiple ways, for example jumping movements in spacing into small grains.

The synthetic sounds are generated by granular synthesis and phase vocoding. The sound synthesis is implemented by the programming language Csound. The Csound score is generated with the language Scheme.

Biography Weiwei Lan was born in 1977 in Dandong, China

She started learning the piano at the age of 9. From 1996 until 2001 she studied composition in Peking (graduated from the China Conservatory of Music in composition with a Bachelor's degree). Her teachers include Wanchun Shi, Kunshen Ruan, Weijie Gao, Yibing Cheng.

Since 2002 she has studied electronic composition at the Institut für Computermusik und Elektronische Medien (ICEM) at the Folkwang-Hochschule in Essen (Germany) with Prof. Dirk Reith. Since 2005 her compositions and performances have been heard in Europe.

In 2006, Lan was awarded first prize at the China III Electronic Music composition competition ”MUSICACOUSTICA-Beijing 2006“.

Currently she concerns herself with Live-Electronics, Live-Performance and Sound-Poetry in speech-composition.

Saturday: 19h00, Tesla

The Electronic Unicorn "das elektronische einhorn" (2006)

Georg Holzmann

audio performance with the unicorn interface

The unicorn is the only fabulous beast that does not seem to have been conceived out of human fears. In even the earliest references he is fierce yet good, selfless yet solitary, but always mysteriously beautiful. He could be captured only by unfair means, and his single horn was said to neutralize poison. (Marianna Mayer)

An unicorn skeleton found at Einhornhöhle ("Unicorn Cave") in Germany’s Harz Mountains in 1663 proves that the so-called unicorn had only two legs, one white "magic" hand and four mysterious plates out of metal. The skeleton was examined by Leibniz, who had previously doubted the existence of the unicorn, but was convinced thereby.

the interface

The unicorn interface consists of 3 main parts: a glove with contacts on the fingertips, a board with 4 contact plates and a potentiometer on the forehead, the unicorn. The goal of the interface is, to be able to control various parameters at the same time.

If you touch one of the four contact plates, you can control a specific parameter with the unicorn, relative to it’s current value. Here it is important, with which finger you touch a plate, because each finger corresponds to one (musical) voice. For instance plate1 controls volume and plate2 pitch. If you have now finger1 on plate1 and finger2 on plate2 and turn the unicorn, you will change the volume of voice1 and the pitch of voice2.

Additionally there are two buttons on the board to change between presets.

technical realization

A microcontroller (arduino board - an open-source physical computing platform) is used to communicate with the computer and all sound synthesis is implemented in Pure Data under Linux.

Biography Georg Holzmann, geboren 1982 in Graz, Austria

since 2002 study of audio engineering (focus in computer music and signal processing) at the institute of electronic music (IEM), Graz

development of audio and video open source software (mainly for Pure Data)

various audio-visual performances and installations at various places, e.g.: Logos Foundation (Gent, Belgium), NIME06 (IRCAM Paris, France), Museumsquartier (Vienna, Austria), pixxelpoint Festival (Nova Gorica, Italy/Slovenia), art@radio (Baltimore, USA), piksel (Bergen, Norway), Radio OE1 (Vienna, Austria), Musikprotokoll (Graz, Austria), ZKM Karlsruhe (Germany), KIBLA (Maribor, Slovenia), Kunsthaus Graz (Graz, Austria), Porgy&Bess (Vienna, Austria).

more information: http://grh.mur.at

Saturday: 19h00, Tesla

ODD (2006) 10:35

Edgar Barroso

Commissioned by DAAD (Deutscher Akademischer Austauschdienst) to be premiered at the Inventionen Festival 2006.

ODD was conceived based on the SMS tools, which is a set of techniques and software implementations for the analysis, transformation and synthesis of musical sounds, developed by Xavier Serra an his team at MTG (Music Technology Group). One of the main processes the SMS offers is the separation of stable pitch components from the noise elements, naming them "residuals". The textures of the piece are made from this components, that after a process of constant transpositions creates very dense no-pitch sound masses. In a sense, ODD is a trio, having threes recognizable sound sources, a violin, a set of percussions and a female voice, which are surrounded and interrupted by this residual permanently moving textures. Also the morphing and transformation processes are constantly used to get "variations" ans "transitions" of this instruments. As a metaphor, the spazialization of sound was founded on the geometrical concept of odd functions that are symmetric with respect to the origin, meaning that its graph remains unchanged after rotation of 180 degrees about the origin. The idea was to create a permanent moving sonorous space in which the trajectories of sounds were applied equally to different sounds, but the resulting effect have totally different semanthic meanings. The meaning of space and distance is determine by a complex system of amplitude layers. The work´s structure is based in four clear moments define mainly by its background sonic textures, in its internal

construction it is also the result of selecting graphic information given by the SMS analysis and subjectively interpret it as a "score" of the incoming musical events. ODD used the SMS tools as a "prism" that can disperse a "light" (sound) wave.

(*See the Cervantes concert for a biography*)

Saturday: 19h00, Tesla

Distance Liquide (2007) 13:00
Hans Tutschku

8-channel electroacoustic composition / commissioned by INA-GRM 2007 / studio : GRM Paris
first performance : January 13, 2007, Maison de la Radio France Paris

To my mother

The picture of liquid, moving and fast dissolving forms became in this composition the metaphor for sound spatialisation in the electroacoustic space. Each musical gesture is bound to a specific space movement, which underlines its character. On the basis of recorded sequences with a gong and percussion instruments, trumpet, flute and vocal fragments, these rather distant sound elements develop a common musical discourse. Their very different spectra are reduced occasionally to the loudest harmonic components, keeping just pure pitches and melodies: their differences disappeared.

(*See "Rituale" in the WFS-Loop for a biography*)

Saturday: 19h00, Tesla

NTSC - NotTheSameColor (Video Sound Duo)
Dieb13 (Turntables/Computer/Devices) and Billy Roisz (Videomixers/Audiomixer/Videosynthesizer)

The question of a synthesis between image and sound has interested composers and artists from the early days of Modernism. The 20th century was full of experiments trying to translate one medium into the other, to synchronise the missions of image and sound. With the rise and development of musical electronics another challenge has arisen that aims beyond a simple synchronisation of image and sound: the merging of the two media in such a way that an image-producing medium will generate music and/or a sound-producing medium images. The ensembles Video Sound Duo Dieb 13 und Billy Roisz and Team Farmersmanual will give an insight into their experiments in this area at Sonic Arts Lounge.

The setup of NTSC consists of various audio and video instruments, connected in a way that allows multiple ways of feedback and physical interaction. Audio and video signals leave their domain to get a new function and meaning. Sound causes images and video signal can be heard. The speaker and the screen finally define whether a signal will appear audible or visible. Thus, signal routing becomes an integral element of the creative process. The instruments are partly self built (e.g. ultrasonic sound-/video- synthesizer and "embedded" mini-computer). The linux-based sound software is self-written. NTSC is a continuously developing project about interactions between sound and video off the well beaten paths of computer analysis and synthesis in a live context.

http://ntsc.klingt.org

Saturday: 22h00, MaerzMusik - Sonic Arts Lounge - Haus der Berliner Festspiele

rf (gophgonih) | Total Automation vs. Human Interaction *Farmersmanual*

With the help of acoustic feedbacks Farmersmanual will weave sound, light and radio frequencies into an endless loop that spreads slowly but continuously and then dissolves again; the process will be supported by a selection of tailor-made as well as standardised hard and software.

http://web.fm

A co-production of Elektronisches Studio der TU Berlin, TESLA im Podewils'schen Palais, Berliner Künstlerprogramm des DAAD and MaerzMusik | Berliner Festspiele With support of the Österreichisches Kulturforum Berlin

Saturday: 22h00, MaerzMusik - Sonic Arts Lounge - Haus der Berliner Festspiele

Livecoding
Alberto De Campo and Powerbooks Unplugged

PBUP Latento
The laptop is the next guitar, i.e. *the* new folk instrument
The laptop is a complete instrument as is

The laptop can also be the entire interface
The laptop in its current physicality is best used while it historically exists - now
Code and music belong to everyone
Sunday: 16h00, Lichthof

Open Hardware Jam
Recursive Dog

The musical results from the workshop on Thursday.
Sunday: 16h30, Lichthof

Unplugged Jam
Livecode vs. Open Hardware

Starting with a jam between the livecoders and the Arduino based sensor driven music, everyone will be able to join into this unplugged jam.
Sunday: 17h00, Lichthof

Index of Authors