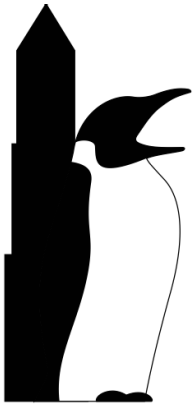
Proceedings of the

Linux Audio Conference 2010

May 1st – 4th, 2010

Hogeschool voor de Kunsten Utrecht

Utrecht, The Netherlands



Published by
Hogeschool voor de Kunsten, Utrecht, The Netherlands
May 2010
All copyrights remain with the authors
`http://lac.linuxaudio.org/2010`

Credits
Cover design: Tijs Ham
Layout: Frank Neumann
Editor: Maurits Lamers
Typesetting: LaTeX and pdfLaTeX

About the cover
The cover text is created using a PureData patch. When you play the patch, you should get an idea of what the penguin at the bottom is singing...

Thanks to:
Martin Monperrus for his webpage "Creating proceedings from PDF files"

Printed in Hilversum by Print service "de Toekomst"
`http://www.detoekomst.nl`

Partners and Sponsors

Hogeschool voor de Kunsten

Loohuis Consulting

Linux Weekly News

Linuxaudio.org

Elastique

Tonehammer

Muziekinstituut MultiMedia

LAC 2010: a Linux Audio Conference

Since 2003 the Linux Audio Conference has become a yearly recurring international event where users and developers of open source music software and Linux Audio in particular get together.

This year the LAC, as it is known by some, is hosted by the Hogeschool voor de Kunsten Utrecht. The HKU offers bachelor and master programmes for various disciplines such as Music, Theatre, Design, Media, Games, Interaction and Music & Technology. Students and alumni of the HKU participate in the organisation and contribute to the conference.

So why is the HKU doing this? And why am I doing this? To answer the first question: because the HKU recognises the importance of Open Source Software for education and, vice versa, by involving the generations that will shape the future we hope to enrich the Linux Audio community.

As for the second question: I'm doing this because I am a long time Linux Audio user, an Open Source advocate, a lecturer at the HKU and I need the freedom that Open Source Software gives me. Oh, and because at last year's LAC in Parma, Italy, I said to someone in the pub how nice it would be to host this conference in a different country or even continent every year. From that moment on my fate was sealed: I was going to get this party to Utrecht.

Originally a Linux Audio developers meeting, the LAC has developed into a conference that comprises not only Linux Audio but also other Open Source Music Software and Open Content. Of course, with Linux being a true Open Source platform, Linux Audio remains the backbone of the venue. The Linux Audio community is more alive than ever, as is demonstrated by this book with its collection of high-standard papers.

The conference offers enough diversity to attract people with different backgrounds and one common denominator: their devotion to open systems. With over 75 scheduled events such as presentations, workshops, discussion panels and concerts, there are lots of opportunities for everyone to participate, learn and find new inspiration, work relations and even friends. My personal hope for the LAC 2010 is that it will contribute to the acceptance and use of open systems for creating, producing, understanding and enjoying music.

To conclude I would like to thank Ad Wisman for giving his approval and Jan IJzermans and Rens Machielse for backing me up. During the entire process, Robin Gareus and Frank Neumann did a tremendous job with advice, support, creating nifty solutions and handling the entire process from papers to proceedings. Thanks guys, I owe you one! Ico Bukvic, thanks for keeping the server up and running. Thank you Farida Ayojil for all your support with respect to the location, security and catering.

And then there are Than van Nispen tot Pannerden and Bertus van Dalen who generated lots of ideas, helped out in numerous situations and gave a whole new dimension to this conference. To all the others I didn't mention: your contributions are highly appreciated.

I wish you all a very good LAC 2010 and hope you enjoy your stay in Utrecht!

Marc Groenewegen

Special thanks to ...

Conference Organisation
Marc Groenewegen
Robin Gareus
Frank Neumann
Marcel Wierckx
Than van Nispen tot Pannerden
Bertus van Dalen

Program
Marc Groenewegen
Than van Nispen tot Pannerden
Bertus van Dalen
Robin Gareus
Frank Neumann

Streaming
Jörn Nettingsmeier
Christian Thäter
Emile Bijk
Herman Robak
Wouter Verwijlen
Raffaella Traniello
Ernst Roza

T-shirt Design
Tijs Ham

Public Relations
Marc Groenewegen
Femke Langebaerd

Conference Website
Marc Groenewegen
Robin Gareus

Logo Design
Niek Kok
Tamara Houtveen

Timetable database and web front-end
Robin Gareus
Marc Groenewegen

Concert Organisation
Marc Groenewegen

Concert Sound
Stefan Janssen

Composition Contest ”150 Years of Music Technology”
Than van Nispen tot Pannerden

Magazine ”The LAC Times”
Niek Kok
Sander Oskamp
Marc Groenewegen
Than van Nispen tot Pannerden
Bertus van Dalen
Maarten Brinkering
Marcel Wierckx

Consulting and Assistance
Luc Nieland
Than van Nispen tot Pannerden
Bertus van Dalen
Danique Kamp
Farida Ayojil
Henk Schraa
Jos Bertens
Roderick van Toor
Sanne Smits
Maurits Lamers
Armijn Hemel
Marije Baalman
Fons Adriaensen

Operational team as known by April 25, 2010

Bertus van Dalen
Ciska Vriezenga
Daan van Hasselt
Danique Kamp
Eric Magnee
Farida Ayojil
Frank Neumann
Giel Dekkers
Harry van Haaren
Henk Schraa
Johan Rietveld
Jos Bertens
Luc Nieland
Marcel Wierckx
Mario van Etten
Martijn van Gessel
Maurits Lamers
Roald van Dillewijn
Roderick van Toor
Sanne Smits
Simon Johanning
Stefan Janssen
Than van Nispen tot Pannerden
Thijs Koerselman
Tijs Ham

Photography

Brendon Heinst

Paper Administration and Proceedings

Frank Neumann

...and to everyone else who helped in numerous places after the editorial deadline of this publication.

Review Committee

Fons Adriaensen	Casa della Musica, Parma, Italy
Marije Baalman	Concordia University, Montreal, Canada
Frank Barknecht	Footils, Cologne, Germany
Ico Bukvic	Virginia Tech, Blacksburg, VA, United States
Götz Dipper	ZKM, Karlsruhe, Germany
Florian Faber	Marburg, Germany
Robin Gareus	University of Paris, France
Marc Groenewegen	Hogeschool voor de Kunsten Utrecht, Utrecht, The Netherlands
Frank Neumann	Harman International, Karlsbad, Germany
Yann Orlarey	GRAME, Lyon, France
Dave Philipps	United States
Martin Rumori	Academy of Media Arts, Cologne, Germany
Pieter Suurmond	Hogeschool voor de Kunsten Utrecht, Utrecht, The Netherlands
Victor Lazzarini	NUI Maynooth, Ireland

Music Jury

Hans Timmermans
Than van Nispen tot Pannerden
Bertus van Dalen
Pieter Suurmond
Marc Groenewegen

Table of Contents

QuteCsound, a Csound Frontend

Andrés CABRERA
Sonic Arts Research Centre
Queen's University Belfast
UK
acabrera01@qub.ac.uk

Abstract

QuteCsound is a front-end application for Csound written using the Qt toolkit. It has been developed since 2008, and is now part of the Csound distribution for Windows and OS X. It is a code editor for Csound, and provides many features for real-time control of the Csound engine, through graphical control interfaces and live score processing.

Keywords

Csound, Front-end, Qt, Widgets, Interface builder

1 Introduction

QuteCsound was born out of the desire to have a cross-platform front-end for Csound [Boulanger, 2000] like MacCsound [Ingalls, 2005] which only runs on OS X. It is based on the idea of having real-time control of Csound through graphical widgets, and making Csound accessible for novice users. It is however designed to be also a powerful editor for advanced users and also offline (non-realtime) work. The most recent version is 0.5.0. It has been tested on Linux, Mac OS X, Windows and Solaris. QuteCsound has been translated to Spanish, French, Italian, German and Portuguese.

One of the main goals was also to make a new interface for Csound which would be comfortable for a musician whose background is not in programming. Existing cross-platform interfaces for Csound were either too basic or somewhat impractical. The Csound Manual [Cabrera, 2010] is very comprehensive, but few front-ends take full advantage of its resources like opcode listing in XML format.

QuteCsound uses the Csound API [ffitch, 2005], which enables tight integration and control of Csound.

It is licensed under the GPLv3 and LGPLv2 for compatibility with the Csound licence to ease distribution alongside Csound. More information on QuteCsound can be found in the

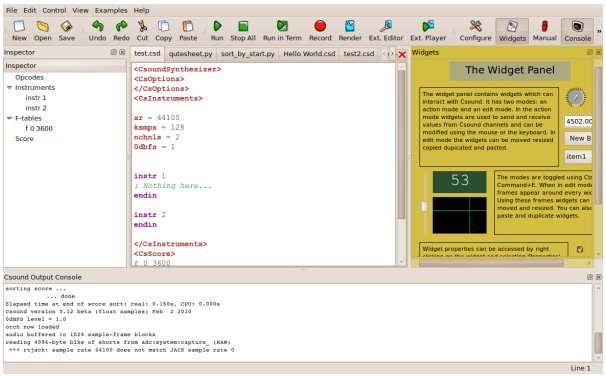


Figure 1: The main QuteCsound window

Figure 2: The transport bar

QuteCsound Front page[1] and the QuteCsound sources and binaries can be found on its Sourceforge page[2].

2 GUI

The main application window is shown in figure 1. The main component is a text editor with syntax highlighting. Documents can be opened as separate tabs in this area. There is a large icon bar with the main actions, which contains the usual open/save and cut/copy/paste action icons, a section with transport controls (see figure 2), and a section for handling visibility of the rest of the application windows and panels.

Around the editor many dockable widgets can be positioned freely. These dockable widgets are:

Widget Panel In the widget panel, control widgets like sliders and knobs can be created. The design and manipulation of the

[1] http://qutecsound.sourceforge.net
[2] http://www.sourceforge.net/projects/qutecsound

widgets is all graphical, and requires no textual programming. See 2.1 below.

Manual Panel The html version of the Csound Manual can be displayed in this panel. The reference for an opcode under the editor cursor can be called with the default short-cut Shift+F1.

Console Panel The output from Csound for the current document is displayed in this panel.

Inspector Panel Shows a dynamically generated hierarchical list of important sections from the current file. It shows instrument definitions and labels, definition of User defined opcodes, f-table definitions, and score sections.

When the current document tab is changed, all the panels which depend on the document like the widget panel and the inspector panel, change to show the current data.

There are three additional windows in the GUI:

Configuration dialog Allows setting options for Csound execution, Environment and interface options.

Utilities dialog Simplifies usage of the Csound utilities -applications which preprocess files for certain opcodes- can be called and controlled through this dialog window.

Live Event Panels These windows which vary in number according to the current document contain a spreadsheet-like interface for manipulating Csound score events which can be sent, manipulated and looped while Csound is running. They can also be processed using the simple python *qutesheet API* (see section 2.3).

2.1 Widgets

The widget panel allows creation of versatile graphical control interfaces. Data widgets provide bi-directional interchange of values with Csound through the API. The values can be updated synchronously or asynchronously depending on the QuteCsound configuration and the usage of *invalue/outvalue* or *chnget/chnset*[3] opcodes in the csd file.

[3]The invalue/outvalue opcodes call a registered callback when they are used, while chnget/chnset only change internal values which must be polled

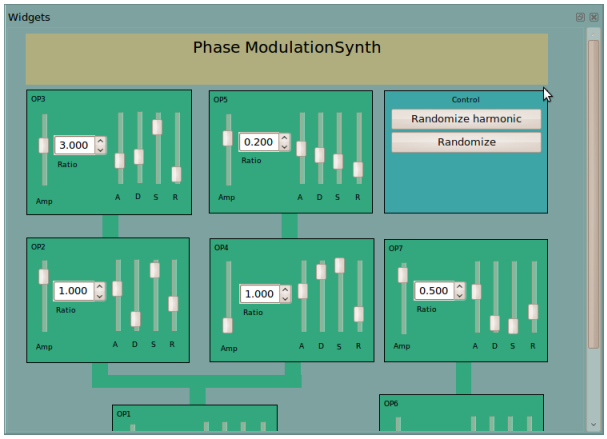


Figure 3: Widget Panel detail

An example of how the widget panel can be used is shown in figure 3.

These are the available data widgets:

Slider Ordinary sliders which become horizontal or vertical depending on the relation between width and height.

Knob Simple rotary knobs.

Labels and Displays Text widgets that display immutable text (labels), or text that can be changed only from Csound, not using the mouse[4].

Text Editor A text entry widget.

Number widgets The ScrollNumber and SpinBox widgets offer number value inputs and outputs with mouse and keyboard control.

Menus The menu widget allows creating a Drop-down or Combo Box for selection from a list.

Controller The controller widget can be a slider, a meter or an XY controller. It can also be used as a "LED" display.

Button Buttons can be data widgets (sending different values depending on whether they are pressed or not) or score event generators. They can also hold images.

CheckBox A simple checkbox which can take a value of 1 or 0.

There are three other widgets which are used to display information from Csound:

[4]While this separation is not really necessary or useful, it follows the MacCsound format choices, and was kept for compatibility purposes only

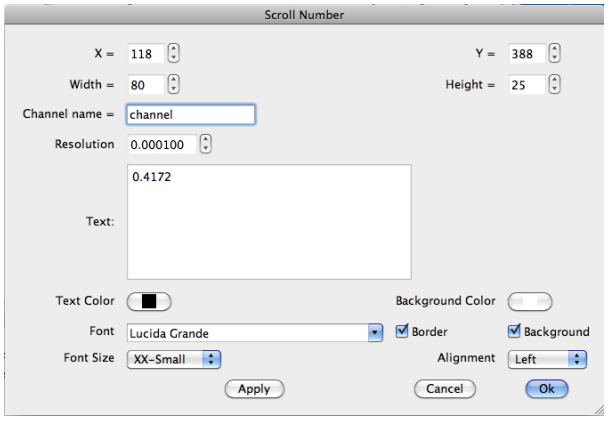


Figure 4: The Widget Preferences dialog

Graphs The Graph Widget displays Csound F-tables, as they are created by Csound, and also displays data from the *dispfft* and *display* opcodes, which allow monitoring any signal or its spectrum. There is a ComboBox which allows selection of the current tables. The table shown can also be selected through a Csound API channel.

Scopes The Scope widget shows visual representations of Csound output buffer. It can act as a traditional Oscilloscope or as a Lissajou, or Poincare display.

Consoles The Console widget shows the Csound console output as a widget in the widget panel.

Widgets can be created by right-clicking on the widget panel, and selecting a type of widget from a list. They can be moved and resized by entering the 'Edit Mode' with the keyboard short-cut Ctrl+E.

All widgets have a configuration dialog which can be shown by right clicking on them and selecting 'Properties' or by double clicking when in edit mode. The Properties dialog for a text widget is shown on figure 4. Properties like size and position can be set in these dialogs. The channel name through which the widget transmits (if it can according to its type) can also be set here.

To receive data from the widgets, they must be assigned to a control variable (*k-rate*) using the *invalue* opcode like this:

```
kval invalue "channel"
```

Conversely, to send data to a widget, the *outvalue* opcode can be used.

```
outvalue "chan", kval
```

There are some handy organization functions for selected widgets like align and distribute to aid in creating better looking widget panels.

The widgets are saved as text in the csd file, but this text is always hidden from the user[5]. Each widget is currently represented by a single line of text, which looks something like this:

```
ioSlider {23, 244} {414, 32} -1.000000
1.000000 -1.000000 amount
```

This follows the original MacCsound format strictly, which enables QuteCsound to open and generate files which are interchangeable with MacCsound. It is somewhat human-readable and editable, but impossible to extend without breaking compatibility. A lot of internal work has already been done to move past the MacCsound widget format to an XML based format which will allow easier extensibility and parsing, and new widget types.

2.2 Live events panel

Each document can have any number of Live Event Panels. A live event panel is a place where Csound score events can be placed for interactive usage during a Csound run. It is shown in figure 5. The live event panel can display the information in the traditional Csound score format or as a spreadsheet with editable cells. It offers a group of common processing functions which can be applied quickly to a group of cells like addition, multiplication, and generation functions like fill linearly or exponentially or random number generation. There is also support for some of the basic Csound Score preprocessor features like tempo control and the carry operator '.'.

Score events can be copied/pasted to/from text view and sheet view seamlessly.

Live Event Panels are also stored as text in the main csd file, and are hidden from the user in the main text editor.

2.3 The QuteSheet Python API

A simple python API has been devised to enable transformation of data from the Live Events Panel using Python. The cells (both the selected and the complete set) are passed to the python script as arrays in any particular organization (by rows, by columns, by individual cells), and can be returned with a single function specifying the new data and where it should

[5]This behavior will probably change in the future to allow text modification of the widgets

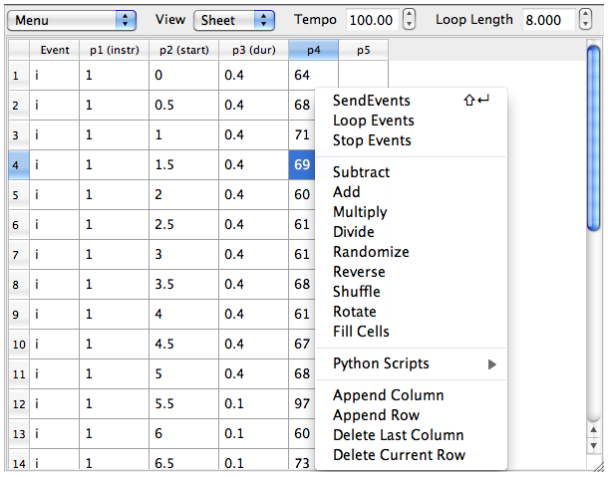


Figure 5: The Live Events Panel

be placed in the sheet. The python scripts can be stored in a directory which is scanned every time the right-click menu in the event sheet is triggered, effectively allowing "live coding" of score transformations while the Csound audio engine is running.

2.4 Code Graph

QuteCsound can parse the currently active document to generate a dot language file, which can be rendered using graphviz[6] . Although complicated files produce diagrams that are too complex, this feature is useful for beginners to see how the variables and opcodes are connected in a visual way. The output of this action can be seen in figure 6.

3 Architecture

QuteCsound requires Qt 4[Nokia, 2010], Csound and libsndfile[de Castro Lopo, 2010] to build.

Qt is a mature cross-platform graphical toolkit, which is used in several well known free-software audio projects like QJackCtl. It also has cross-platform libraries for non GUI stuff like networking, XML parsing, printing and threading which has saved time and avoided the need for additional dependencies. It also has extensive facilities for internationalization and translation. It is distributed in separate dynamic libraries which can be distributed with the executable.

Libsndfile is a well established audiofile

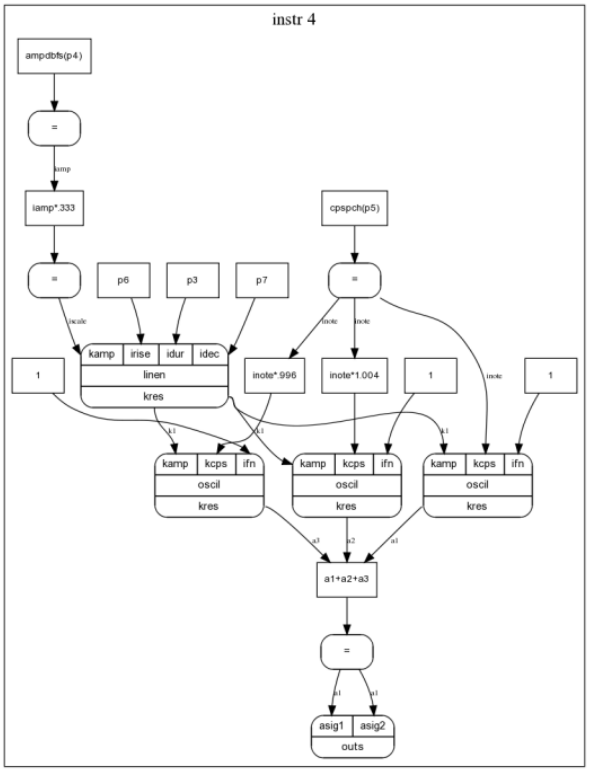


Figure 6: The Code Graph Output

read/write library, which is very well maintained, stable and efficient.

All audio and MIDI I/O is handled by Csound itself, except the Record function, which copies the Csound output buffer after every control block to a ring buffer and writes it to disk from another thread. This means that QuteCsound supports output to Jack, Coreaudio and WinMME as well as generic interfaces like Portaudio.

Csound can be run either as a completely independent process in a separate Terminal application, on the same application thread (usually undesirable) or on a separate thread using the API through the *CsoundPerformanceThread*[7] C++ interface from `interfaces/csPerfThread.hpp` in the Csound sources.

Data update is requested by Csound synchronously through its callback functions (set using the `csoundSetInputValueCallback` and `csoundSetOutputValueCallback` functions) when the *invalue/outvalue* opcodes are used. Data for Csound is polled in that callback, but data for the widgets is queued, to be processed

[6] Graphviz is a package for generating flowchart style graphics using the dot language. More information can be found on www.graphviz.org

[7] this class handles some of the threading issues associated with passing events to Csound and with starting/pausing/stopping the engine

at a slower rate. The values can also be updated asynchronously with the *chnset/chnget* opcodes. The values are then updated from a timer triggered thread in QuteCsound, forcing locking of values.

An interesting feature of MacCsound which has been emulated in QuteCsound is that widgets with the same channel name update each other even when Csound is not running. This is useful to avoid Csound code for common things like showing the numeric value of a slider widget, and it also gives a (false but expected and rewarding) sense of an "always running" engine. In practical terms, this implies that when Csound is running, widgets must first take values generated from Csound, and then propagate their values to other widgets.

3.1 Documentation integration

The Csound documentation is highly integrated in QuteCsound. It can be easily called from the Help menu, or to find the reference for the opcode under the editor cursor. The documentation also contains an XML file of all the opcode definitions organized by category. This file is used for syntax highlight, but also for showing opcode syntax in the status bar, populating an opcode selector organized by categories and building the code graph code (to know the input/output variable names and types).

4 Examples Menu

The Examples menu in QuteCsound includes a large set of introductory examples and tutorials, reference examples for the widgets and a group of 'classic' Csound compositions like Boulanger's *Trapped in Convert*, and Csound realizations of important historical pieces like Chowning's *Stria* and Stockhausen's *Studie II*. The examples menu also contains a set of realtime synths, some useful files for things like I/O monitoring and file processing, and a 'Favorites' menu which shows the csd files from a user specified directory.

5 The future

Plans for the future include completion of the new widget format, and development of new widgets like a table widget. It would be nice to have a synchronization option to make loops sync to a master loop.

A lot of work has been done to enable the exporting of stand alone applications, so this will hopefully be finished soon.

And of course fix some of the bugs along the way.

6 Acknowledgements

Many thanks to the Csound developers, specially John ffitch and Victor Lazzarini for their assistance with usage of the API during the development of QuteCsound, and their quick response to issues and particular needs. Also, many thanks go to the users who have contributed many ideas, translations, bug reports and testing. Thanks to people like Joachim Heintz, Andy Fillebrown and François Pinot, for their contribution to development and packaging.

References

Richard Boulanger, editor. 2000. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing and Programming*. MIT Press.

Andres Cabrera, editor. 2010. *The Csound 5.12 Manual*.

Erik de Castro Lopo. 2010. Libsndfile. http://www.mega-nerd.com/libsndfile/api.html.

John ffitch. 2005. On the design of csound 5. In *Proceedings of the 2005 Linux Audio Conference*, ZKM, Karlsruhe, Germany.

Matt Ingalls. 2005. Maccsound. http://www.csounds.com/matt/MacCsound/.

Nokia. 2010. Qt – cross-platform application and ui framework. http://qt.nokia.com/.

Implementing a Polyphonic MIDI Software Synthesizer using Coroutines, Realtime Garbage Collection, Closures, Auto-Allocated Variables, Dynamic Scoping, and Continuation Passing Style Programming

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
k.s.matheussen@notam02.no

Abstract

This paper demonstrates a few programming techniques for low-latency sample-by-sample audio programming. Some of them may not have been used for this purpose before. The demonstrated techniques are: Realtime memory allocation, realtime garbage collector, storing instrument data implicitly in closures, auto-allocated variables, handling signal buses using dynamic scoping, and continuation passing style programming.

Keywords

Audio programming, realtime garbage collection, coroutines, dynamic scoping, Continuation Passing Style.

1 Introduction

This paper demonstrates how to implement a MIDI software synthesizer (MIDI soft synth) using some unusual audio programming techniques. The examples are written for *Snd-RT* [Matheussen, 2008], an experimental audio programming system supporting these techniques. The techniques firstly emphasize convenience (i.e. few lines of code, and easy to read and modify), and not performance. Snd-RT[1] runs on top of Snd[2] which again runs on top of the Scheme interpreter Guile.[3] Guile helps gluing all parts together.

It is common in music programming only to compute the sounds themselves in a realtime priority thread. Scheduling new notes, allocation of data, data initialization, etc. are usually performed in a thread which has a lower priority than the audio thread. Doing it this way helps to ensure constant and predictable CPU usage for the audio thread. But writing code that way is also more complicated. At least, when all samples are calculated one by one. If however the programming only concerns handling blocks of samples where we only control a signal graph, there are several high level alternatives which makes it relatively easy to do a straightforward implementation of a MIDI soft synth. Examples of such high level music programming systems are SuperCollider [McCartney, 2002], Pd [Puckette, 2002], CSound[4] and many others.

But this paper does not describe use of block processing. In this paper, all samples are individually calculated. The paper also explores possible advantages of doing everything, allocation, initialization, scheduling, etc., from inside the realtime audio thread.

At least it looks like everything is performed inside the realtime audio thread. The underlying implementation is free to reorganize the code any way it wants, although such reorganizing is not performed in Snd-RT yet.

Future work is making code using these techniques perform equally, or perhaps even better, than code where allocation and initialization of data is explicitly written not to run in the realtime audio thread.

2 MIDI software synthesizer

The reason for demonstrating a MIDI soft synth instead of other types of music programs such as a granular synthesis generator or a reverb, is that the behavior of a MIDI soft synth is well known, plus that a MIDI soft synth contains many common challenges in audio and music programming:

1. Generating samples. To hear sound, we need to generate samples at the *Audio Rate*.
2. Handling Events. MIDI data are read at a rate lower than the audio rate. This rate is commonly called the *Control Rate*.

[1] http://archive.notam02.no/arkiv/doc/snd-rt/
[2] http://ccrma.stanford.edu/software/snd/
[3] http://www.gnu.org/software/guile/guile.html
[4] http://www.csound.com

3. Variable polyphony. Sometimes no notes are playing, sometimes maybe 30 notes are playing.

4. Data allocation. Each playing note requires some data to keep track of frequency, phase, envelope position, volume, etc. The challenges are; How do we allocate memory for the data? When do we allocate memory for the data? How do we store the memory holding the data? When do we initialize the data?

5. Bus routing. The sound coming from the tone generators is commonly routed both through an envelope and a reverb. In addition, the tones may be autopanned, i.e. panned differently between two loudspeakers depending on the note height (similar to the direction of the sound coming from a piano or a pipe organ).

3 Common Syntax for the Examples

The examples are written for a variant of the programming language Scheme [Steele and Sussman, 1978]. Scheme is a functional language with imperative operators and static scoping.

A number of additional macros and special operators have been added to the language, and some of them are documented here because of the examples later in the paper.

(<rt-stalin>...) is a macro which first transforms the code inside the block into clean R4RS code [Clinger and Rees, 1991] understood by the Stalin Scheme compiler.[5] (Stalin Scheme is an R4RS compiler). After Stalin is finished compiling the code, the produced object file is dynamically linked into Snd-RT and scheduled to immediately run inside the realtime audio thread.

(define-stalin *signature* **...)** defines variables and functions which are automatically inserted into the generated Stalin scheme code if needed. The syntax is similar to *define*.

(spawn ...**)** spawns a new coroutine [Conway, 1963; Dahl and Nygaard, 1966]. Coroutines are stored in a priority queue and it is not necessary to explicitly call the spawned coroutine to make it run. The spawned coroutine will run automatically as soon[6] as the current coroutine yields (by calling ***yield*** or ***wait***), or the current coroutine ends.

Coroutines are convenient in music programming since it often turns out practical to let one dedicated coroutine handle only one voice, instead of mixing the voices manually. Furthermore, arbitrarily placed pauses and breaks are relatively easy to implement when using coroutines, and therefore, supporting dynamic control rate similar to ChucK [Wang and Cook, 2003] comes for free.

(wait n**)** waits n number of frames before continuing the execution of the current coroutine.

(sound ...**)** spawns a special kind of coroutine where the code inside ***sound*** is called one time per sample. (***sound*** coroutines are stored in a tree and not in a priority queue since the order of execution for ***sound*** coroutines depends on the bus system and not when they are scheduled to wake up.)

A simple version of the ***sound*** macro, called *my-sound*, can be implemented like this:

```
(define-stalin-macro (my-sound . body)
  `(spawn
     (while #t
       ,@body
       (wait 1))))
```

However, *my-sound* is inefficient compared to ***sound*** since *my-sound* is likely to do a coroutine context switch at every call to ***wait***.[7] ***sound*** doesn't suffer from this problem since it is run in a special mode. This mode makes it possible to run tight loops which does not cause a context switch until the next scheduled event.

(out <*channel***>** *sample***)** sends out data to the *current bus* at the *current time*. (the *current bus* and the *current time* can be thought of as global variables which are implicitly read from and written to by many

[5] Stalin - a STAtic Language ImplementatioN, http://cobweb.ecn.purdue.edu/ qobi/software.html.

[6] Unless other coroutines are placed earlier in the queue.

[7] I.e. if two or more *my-sound* blocks or ***sound*** blocks run simultaneously, and at least one of them is a *my-sound* block, there will be at least two coroutine context switches at every sound frame.

operators in the system)[8] By default, the *current bus* is connected to the sound card, but this can be overridden by using the ***in*** macro which is explained in more detail later.

If the *channel* argument is omitted, the *sample* is written both to channel 0 and 1.

It makes sense only to use ***out*** inside a ***sound*** block. The following example plays a 400Hz sine sound to the sound card:

```
(<rt-stalin>
  (let ((phase 0.0))
    (sound
      (out (sin phase))
      (inc! phase (hz->radians 400)))))
```

(range *varname start end ...***)** is a simple loop iterator macro which can be implemented like this:[9]

```
(define-macro (range varname start end . body)
  (define loop (gensym))
  `(let ,loop ((,varname ,start))
     (cond ((<,var ,end)
            ,@body
            (,loop (+ ,varname 1))))))
```

(wait-midi :*options ...***)** waits until MIDI data is received, either from an external interface, or from another program.

wait-midi has a few options to specify the kind of MIDI data it is waiting for. In the examples in this paper, the following options for ***wait-midi*** are used:

:***command note-on***
: Only wait for a *note on* MIDI message.

:***command note-off***
: Only wait for a *note off* MIDI message.

:***note*** *number*
: Only wait for a note which has MIDI note number *number*.

Inside the ***wait-midi*** block we also have access to data created from the incoming midi event. In this paper we use ***(midi-vol)*** for getting the velocity (converted to a number between 0.0 and 1.0), and ***(midi-note)*** for getting the MIDI note number.

:where is just another way to declare local variables. For example,

```
(+ 2 b
  :where b 50)
```

is another way of writing

```
(let ((b 50))
  (+ 2 b))
```

There are three reason for using ***:where*** instead of ***let***. The first reason is that the use of ***:where*** requires less parenthesis. The second reason is that reading the code sometimes sounds more natural this way. (I.e "add 2 and b, where b is 50" instead of "let b be 50, add 2 and b".) The third reason is that it's sometimes easier to understand the code if you know what you want to do with a variable, before it is defined.

4 Basic MIDI Soft Synth

We start by showing what is probably the simplest way to implement a MIDI soft synth:

```
(range note-num 0 128
  (<rt-stalin>
    (define phase 0.0)
    (define volume 0.0)
    (sound
      (out (* volume (sin phase))))
      (inc! phase (midi->radians note-num)))
    (while #t
      (wait-midi :command note-on :note note-num
        (set! volume (midi-vol)))
      (wait-midi :command note-off :note note-num
        (set! volume 0.0)))))
```

This program runs 128 instruments simultaneously. Each instrument is responsible for playing one tone. 128 variables holding volume are also used for communicating between the parts of the code which plays sound (running at the *audio rate*), and the parts of the code which reads MIDI information (running at the *control rate*[10]).

There are several things in this version which are not optimal. Most important is that you

[8] Internally, the *current bus* is a coroutine-local variable, while the *current time* is a global variable.

[9] The actual implementation used in *Snd-RT* also makes sure that "end" is always evaluated only one time.

[10] Note that the *control rate* in Snd-RT is *dynamic*, similar to the music programming system *ChucK*. *Dynamic control rate* means that the smallest available time-difference between events is not set to a fixed number, but can vary. In ChucK, control rate events are measured in floating numbers, while in Snd-RT the measurement is in frames. So In Chuck, the time difference can be very small, while in Snd-RT, it can not be smaller than 1 frame.

would normally not let all instruments play all the time, causing unnecessary CPU usage. You would also normally limit the polyphony to a fixed number, for instance 32 or 64 simultaneously sounds, and then immediately schedule new notes to a free instrument, if there is one.

5 Realtime Memory Allocation

As mentioned, everything inside ***<rt-stalin>*** runs in the audio realtime thread. Allocating memory inside the audio thread using the OS allocation function may cause surprising glitches in sound since it is not guaranteed to be an $O(1)$ allocator, meaning that it may not always spend the same amount of time. Therefore, *Snd-RT* allocates memory using the Rollendurchmesserzeitsammler [Matheussen, 2009] garbage collector instead. The memory allocator in Rollendurchmesserzeitsammler is not only running in $O(1)$, but it also allocates memory extremely efficiently. [Matheussen, 2009]

In the following example, it's clearer that instrument data are actually stored in closures which are allocated during runtime.[11] In addition, the 128 spawned coroutines themselves require some memory which also needs to be allocated:

```
(<rt-stalin>
  (range note-num 0 128
   (spawn
     (define phase 0.0)
     (define volume 0.0)
     (sound
       (out (* volume (sin phase))))
       (inc! phase (midi->radians note-num)))
     (while #t
       (wait-midi :command note-on :note note-num
         (set! volume (midi-vol)))
       (wait-midi :command note-off :note note-num
         (set! volume 0.0)))))
```

6 Realtime Garbage Collection. (Creating new instruments only when needed)

The previous version of the MIDI soft synth did allocate some memory. However, since all memory required for the lifetime of the program were allocated during startup, it was not necessary to free any memory during runtime.

But in the following example, we simplify the code further by creating new tones only when they are needed. And to do that, it is necessary to free memory used by sounds not playing anymore to avoid running out of memory. Luckily though, freeing memory is taken care of automatically by the Rollendurchmesserzeitsammler garbage collector, so we don't have to do anything special:

```
 1| (define-stalin (softsynth)
 2|   (while #t
 3|     (wait-midi :command note-on
 4|       (define osc  (make-oscil :freq (midi->hz (midi-note))))
 5|       (define tone (sound (out (* (midi-vol) (oscil osc)))))
 6|       (spawn
 7|          (wait-midi :command note-off :note (midi-note)
 8|            (stop tone))))))
 9|
10| (<rt-stalin>
11|   (softsynth))
```

In this program, when a *note-on* message is received at line 3, two coroutines are scheduled:

1. A ***sound*** coroutine at line 5.
2. A regular coroutine at line 6.

Afterwards, the execution immediately jumps back to line 3 again, ready to schedule new notes.

So the MIDI soft synth is still polyphonic, and contrary to the earlier versions, the CPU is now the only factor limiting the number of simultaneously playing sounds.[12]

7 Auto-Allocated Variables

In the following modification, the CLM [Schottstaedt, 1994] oscillator *oscil* will be implicitly and automatically allocated first time the function *oscil* is called. After the generator is allocated, a pointer to it is stored in a special memory slot in the current coroutine.

Since *oscil* is called from inside a ***sound*** coroutine, it is natural to store the generator in the coroutine itself to avoid all tones using the same oscillator, which would happen if the auto-allocated variable had been stored in a global variable. The new definition of *softsynth* now looks like this:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))
```

[11] Note that memory allocation performed before any *sound* block can easily be run in a non-realtime thread before scheduling the rest of the code to run in realtime. But that is just an optimization.

[12] Letting the CPU be the only factor to limit polyphony is not necessarily a good thing, but doing so in this case makes the example very simple.

The difference between this version and the previous one is subtle. But if we instead look at the reverb instrument in the next section, it would span twice as many lines of code, and the code using the reverb would require additional logic to create the instrument.

8 Adding Reverb. (Introducing signal buses)

A MIDI soft synth might sound unprofessional or unnatural without some reverb. In this example we implement John Chowning's reverb[13] and connect it to the output of the MIDI soft synth by using the built-in signal bus system:

```
(define-stalin (reverb input)
  (delay :size (* .013 (mus-srate))
    (+ (comb :scaler 0.742 :size  9601 allpass-composed)
       (comb :scaler 0.733 :size 10007 allpass-composed)
       (comb :scaler 0.715 :size 10799 allpass-composed)
       (comb :scaler 0.697 :size 11597 allpass-composed)
       :where allpass-composed
         (send input :through
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)))))

(define-stalin bus (make-bus))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (write-bus bus
                     (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (fx-ctrl input clean wet processor)
  (+ (* clean input)
     (* wet (processor input))))

(<rt-stalin>
  (spawn
    (softsynth))
  (sound
    (out (fx-ctrl (read-bus bus)
                  0.5 0.09
                  reverb))))
```

Signal buses are far from being an "unusual technique", but in text based languages they are in disadvantage compared to graphical music languages such as Max [Puckette, 2002] or Pd. In text based languages it's inconvenient to write to buses, read from buses, and most importantly; it's hard to see the signal flow. However, signal buses (or something which provides similar functionality) are necessary, so it would be nice to have a better way to handle them.

[13] as implemented by Bill Schottstaedt in the file "jc-reverb.scm" included with Snd. The *fx-ctrl* function is a copy of the function *fxctrl* implemented in Faust's Freeverb example.

9 Routing Signals with Dynamic Scoping. (Getting rid of manually handling sound buses)

A slightly less verbose way to create, read and write signal buses is to use dynamic scoping to route signals. The bus itself is stored in a coroutine-local variable and created using the ***in*** macro.

Dynamic scoping comes from the fact that ***out*** writes to the bus which was last set up by ***in***. In other words, the scope for the *current bus* (the bus used by ***out***) follows the execution of the program. If ***out*** isn't (somehow) called from ***in***, it will instead write to the bus connected to the soundcard.

For instance, instead of writing:

```
(define-stalin bus (make-bus))

(define-stalin (instr1)
  (sound (write-bus bus 0.5)))

(define-stalin (instr2)
  (sound (write-bus bus -0.5)))

(<rt-stalin>
  (instr1)
  (instr2)
  (sound
    (out (read-bus bus))))
```

we can write:

```
(define-stalin (instr1)
  (sound (out 0.5)))

(define-stalin (instr2)
  (sound (out -0.5)))

(<rt-stalin>
  (sound
    (out (in (instr1)
             (instr2)))))
```

What happened here was that the first time ***in*** was called in the main block, it spawned a new coroutine and created a new bus. The new coroutine then ran immediately, and the first thing it did was to change the *current bus* to the newly created bus. The ***in*** macro also made sure that all ***sound*** blocks called from within the ***in*** macro (i.e. the ones created in *instr1* and *instr2*) is going to run before the main ***sound*** block. (That's how ***sound*** coroutines are stored in a tree)

When transforming the MIDI soft synth to use ***in*** instead of manually handling buses, it will look like this:

```
;; <The reverb instrument is unchanged>

;; Don't need the bus anymore:
(define-stalin bus (make-bus))

;; softsynth reverted back to the previous version:
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

;; A simpler main block:
(<rt-stalin>
  (sound
    (out (fx-ctrl (in (softsynth))
                  0.5 0.09
                 reverb))))
```

10 CPS Sound Generators. (Adding stereo reverb and autopanning)

Using coroutine-local variables was convenient in the previous examples. But what happens if we want to implement autopanning and (a very simple) stereo reverb, as illustrated by the graph below?

```
              +-- reverb -> out ch 0
             /
softsynth--<
             \
              +-- reverb -> out ch 1
```

First, lets try with the tools we have used so far:

```
(define-stalin (stereo-pan input c)
  (let* ((sqrt2/2 (/ (sqrt 2) 2))
         (angle (- pi/4 (* c pi/2)))
         (left  (* sqrt2/2 (+ (cos angle) (sin angle))))
         (right (* sqrt2/2 (- (cos angle) (sin angle)))))
    (out 0 (* input left))
    (out 1 (* input right))))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (stereo-pan (* (midi-vol)
                         (oscil :freq (midi->hz (midi-note))))
                      (/ (midi-note) 127.0))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(<rt-stalin>
  (sound
    (in (softsynth)
        (lambda (sound-left sound-right)
          (out 0 (fx-ctrl sound-left  0.5 0.09 reverb))
          (out 1 (fx-ctrl sound-right 0.5 0.09 reverb))))))
```

At first glance, it may look okay. But the reverb will not work properly. The reason is that auto-generated variables used for coroutine-local variables are identified by their position in the source. And since the code for the reverb is written only one place in the source, but used two times from the same coroutine, both channels will use the same coroutine-local variables used by the reverb; a delay, four comb filters and four all-pass filters.

There are a few ways to work around this problem. The quickest work-around is to recode 'reverb' into a macro instead of a function. However, since neither the problem nor any solution to the problem are very obvious, plus that it is slower to use coroutine-local variables than manually allocating them (it requires extra instructions to check whether the data has been allocated[14]), it's tempting not to use coroutine-local variables at all.

Instead we introduce a new concept called CPS Sound Generators, where CPS stands for Continuation Passing Style. [Sussman and Steele, 1975]

10.1 How it works

Working with CPS Sound Generators are similar to Faust's Block Diagrams composition. [Orlarey et al., 2004] A CPS Sound Generator can also be seen as similar to a Block Diagram in Faust, and connecting the CPS Sound Generators is quite similar to Faust's Block Diagram Algebra (BDA).

CPS Sound Generators are CPS functions which are able to connect to other CPS Sound Generators in order to build up a larger function for processing samples. The advantage of building up a program this way is that we know what data is needed before starting to process samples. This means that auto-allocated variables don't have to be stored in coroutines, but can be allocated before running the *sound* block.

For instance, the following code is written in generator-style and plays a 400Hz sine sound to the sound card:

```
(let ((Generator (let ((osc (make-oscillator :freq 400)))
                   (lambda (kont)
                     (kont (oscil osc))))))
  (sound
    (Generator (lambda (sample)
                 (out sample)))))
```

The variable *kont* in the function *Generator* is the continuation, and it is always the last argument in a CPS Sound Generator. A continuation is a function containing the rest of the program. In other words, a continuation function

[14] It is possible to optimize away these checks, but doing so either requires restricting the liberty of the programmer, some kind of JIT-compilation, or doing a whole-program analysis.

will never return. The main reason for programming this way is for generators to easily return more than one sample, i.e have more than one output.[15]

Programming directly this way, as shown above, is not convenient, so in order to make programming simpler, some additional syntax have been added. The two most common operators are Seq and Par, which behave similar to the '**:**' and ',' infix operators in Faust.[16]

Seq creates a new generator by connecting generators in sequence. In case an argument is not a generator, a generator will automatically be created from the argument.

For instance, (Seq (+ 2)) is the same as writing

```
(let ((generator0 (lambda (arg1 kont0)
                    (kont0 (+ 2 arg1)))))
  (lambda (input0 kont1)
    (generator0 input0 kont1)))
```

and (Seq (+ (random 1.0)) (+ 1)) is the same as writing

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input kont2)
    (generator0 input (lambda (result0)
                        (generator1 result0 kont2)))))

;; Evaluating ((Seq (+ 2) (+ 1)) 3 display)
;; will print 6!
```

Par creates a new generator by connecting generators in parallel. Similar to *Seq*, if an argument is not a generator, a generator using the argument will be created automatically.

For instance, (Par (+ (random 1.0)) (+ 1)) is the same as writing:

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input2 input3 kont1)
    (generator0 input2
      (lambda (result0)
        (generator1 input3
          (lambda (result1)
            (kont1 result0 result1)))))))

;; Evaluating ((Par (+ 2)(+ 1)) 3 4 +) will return 10!
```

(gen-sound :*options* **generator)** is the same as writing

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0)
           (out 0 result0)))))
```

...when the generator has one output. If the generator has two outputs, it will look like this:

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0 result1)
           (out 0 result0)
           (out 1 result1)))))
```

...and so forth.

The Snd-RT preprocessor knows if a variable or expression is a CPS Sound Generator by looking at whether the first character is capitol. For instance, (Seq (Process 2)) is equal to (Process 2), while (Seq (process 2)) is equal to (lambda (input kont) (kont (process 2 input))), regardless of how 'Process' and 'process' are defined.

10.2 Handling auto-allocated variables

oscil and the other CLM generators are macros, and the expanded code for *(oscil :freq 440)* looks like this:

```
(oscil_ (autovar (make_oscil_ 440 0.0)) 0 0)
```

Normally, *autovar* variables are translated into coroutine-local variables in a separate step performed after macro expansion. However, when an auto-allocated variable is an argument for a generator, the *autovar* surrounding is removed. And, similar to other arguments which are normal function calls, the initialization code is placed before the generator function. For example, (Seq (oscil :freq 440)) is expanded into:[17]

```
(let ((generator0 (let ((var0 (make_oscil_ 440 0.0)))
                    (lambda (kont)
                      (kont (oscil_ var0 0 0))))))
  (lambda (kont)
    (generator0 kont)))
```

[15] Also note that by inlining functions, the Stalin scheme compiler is able to optimize away the extra syntax necessary for the CPS style.

[16] Several other special operators are available as well, but this paper is too short to document all of them.

[17] Since the Snd-RT preprocessor doesn't know the number of arguments for normal functions such as *oscil_*, this expansion requires the preprocessor to know that this particular *Seq* block has 0 inputs. The preprocessor should usually get this information from the code calling *Seq*, but it can also be defined explicitly, for example like this: (Seq 0 Cut (oscil :freq 440)).

10.3 The Soft Synth using CPS Sound Generators

```
(define-stalin (Reverb)
  (Seq (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (Sum (comb :scaler 0.742 :size 9601)
            (comb :scaler 0.733 :size 10007)
            (comb :scaler 0.715 :size 10799)
            (comb :scaler 0.697 :size 11597))
       (delay :size (* .013 (mus-srate)))))

(define-stalin (Stereo-pan c)
  (Split Identity
         (* left)
         (* right)
    :where left  (* sqrt2/2 (+ (cos angle) (sin angle)))
    :where right (* sqrt2/2 (- (cos angle) (sin angle)))
    :where angle (- pi/4 (* c pi/2))
    :where sqrt2/2 (/ (sqrt 2) 2)))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (gen-sound
          (Seq (oscil :freq (midi->hz (midi-note)))
               (* (midi-vol))
               (Stereo-pan (/ (midi-note) 127)))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (Fx-ctrl clean wet Fx)
  (Sum (* clean)
       (Seq Fx
            (* wet))))

(<rt-stalin>
  (gen-sound
    (Seq (In (softsynth))
         (Par (Fx-ctrl 0.5 0.09 (Reverb))
              (Fx-ctrl 0.5 0.09 (Reverb))))))
```

11 Adding an ADSR Envelope

And finally, to make the MIDI soft synth sound decent, we need to avoid clicks caused by suddenly starting and stopping sounds. To do this, we use a built-in ADSR envelope generator (entirely written in Scheme) for ramping up and down the volume. Only the function *softsynth* needs to be changed:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (gen-sound :while (-> adsr is-running)
        (Seq (Prod (oscil :freq (midi->hz (midi-note)))
                   (midi-vol)
                   (-> adsr next))
             (Stereo-pan (/ (midi-note) 127))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (-> adsr stop)))
      :where adsr (make-adsr :a 20:-ms
                             :d 30:-ms
                             :s 0.2
                             :r 70:-ms))))
```

12 Conclusion

This paper has shown a few techniques for doing low-latency sample-by-sample audio programming.

13 Acknowledgments

Thanks to the anonymous reviewers and Anders Vinjar for comments and suggestions.

References

William Clinger and Jonathan Rees. 1991. Revised Report (4) On The Algorithmic Language Scheme.

Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408.

O.-J. Dahl and K. Nygaard. 1966. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678.

Kjetil Matheussen. 2008. Realtime Music Programming Using Snd-RT. In *Proceedings of the International Computer Music Conference.*

Kjetil Matheussen. 2009. Conservative Gargage Collectors for Realtime Audio Processing. In *Proceedings of the International Computer Music Conference*, pages 359–366 (online erratum).

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(2):61–68.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust, *soft computing.*

Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

W. Schottstaedt. 1994. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):30–37.

Jr. Steele, Guy Lewis and Gerald Jay Sussman. 1978. The Revised Report on SCHEME: A Dialect of LISP. *Technical Report 452, MIT.*

Gerald Jay Sussman and Jr. Steele, Guy Lewis. 1975. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab.*

Ge Wang and Perry Cook. 2003. ChucK: a Concurrent and On-the-fly Audio Programming Language, *Proceedings of the ICMC.*

Writing Audio Applications using GStreamer

Stefan KOST
GStreamer community, Nokia/Meego
Majurinkatu 12 B 43
Espoo, Finland, 02600
ensonic@sonicpulse.de

Abstract

GStreamer is mostly known for its use in media players. Although the API and the plugin collection has much to offer for audio composing and editing applications as well. This paper introduces the framework, focusing on features interesting for audio processing applications. The author reports about practical experience of using GStreamer inside the Buzztard project.

Keywords

GStreamer, Framework, Composer, Audio.

1 GStreamer framework

GStreamer [1] is an open source, cross platform, graph based, multimedia framework. Development takes place on freedesktop.org. GStreamer is written in C, but in an object oriented fashion using the GObject framework. Numerous language bindings exist, e.g. for Python, C#, Java and Perl. GStreamer has been ported to a wide variety of platforms such as Linux, Windows, MacOS, BSD and Solaris. The framework and most plugins are released under LGPL license. The project is over 10 years old now and has many contributors. The current 0.10 series is API and ABI stable and being actively developed since about 5 years.

A great variety of applications is using GStreamer already today. To give some examples, then GNOME desktop is using it for its Mediaplayers Totem and Rhythmbox and the Chat&Voip client Empathy, platforms like Nokias Maemo and Intels Moblin use GStreamer for all multimedia tasks. In the audio domain two applications to mention are Jokosher (a multitrack audio editor) and Buzztard [2] (a tracker like musical composer). The latter is the pet project of the article's author and will be used later on as an example case and thus deserves a little introduction. A song in Buzztard is a network of sound generators, effects and an audio sink. All parameters on each of the elements can be controlled interactively or by the sequencer. All audio is rendered on the fly by algorithms and by optionally accessing a wavetable of sampled sounds. [3] gives a nice overview of live audio effects in Buzztard.

1.1 Multimedia processing graph

A GStreamer application constructs its processing graph in a GstPipeline object. This is a container for actual elements (source, effect and sinks). A container element is an element itself, supporting the same API as the real elements it contains. A pipeline is built by adding more elements/containers and linking them through their “pads”. A pipeline can be anything from a simple linear sequence to a complex hierarchical directed graph (see Illustration 1).

As mentioned above, elements are linked by connecting two pads – a source pad of the source element and a sink pad from the target element. GStreamer knows about several types of pads – always-, sometimes- and request-pads:

- always-pads are static (always available)
- sometimes-pads appear and disappear

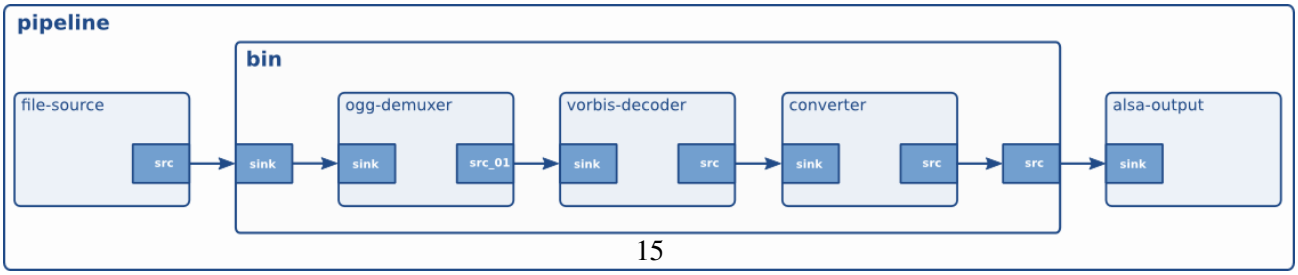


Illustration 1: A hierarchical GStreamer pipeline

according to data-flow (e.g. new stream on multiplexed content)
- request-pads are a template that can be instantiated multiple times (e.g. inputs for a mixer element)

Besides generic containers (containers to structure a pipeline like GstBin and GstPipeline) several specialized containers exist:

- GstPlaybin2: builds a media playback pipeline based on the media content, e.g. used in Totem
- GstCamerabin: abstract still image/video capture workflow, e.g. used in Nokia N900
- Gnonlin: multitrack audio/video editor, e.g. used in Jokosher and Pitivi

1.2 Data formats

The GStreamer core is totally data format agnostic. Formats like audio and video (and their properties) are only introduced with the gst-plugins-base module. New formats can be added without modifying the core (consider adding spectral audio processing plugins like VAMP [4]). Data formats are described by GstCaps which is an object containing a media-type and a list of {key, value} pairs. Values can be fixed or variable (a range or list of possible values). The GstCaps class implements operations on caps (union, intersection, iterator,...). This is quite different to e.g. ladspa where the standard specifies a fix format like float-audio with 1 channel.

Elements register template caps for their pads. This describes the data formats that they can accept. When starting the pipeline, elements agree on the formats that they will use and store this in their pads. Different pads can use different formats and formats can also change while the pipeline is running.

GStreamer plugin packages provide various data format conversion plugins such as e.g. audioconvert and audioresample. Such plugins switch to pass-through mode if no conversion is needed. For performance and quality reason it is of course best to avoid conversion.

1.3 Data Flow

The framework provides various communication and I/O mechanisms between elements and the application. The main data-flow is done by transferring buffers from pad to pad. A pipeline can operate “push based”, “pull based” and in “mixed mode”. In push mode, source elements are active and deliver data downstream. In pull mode sink elements request data form upstream elements when needed. Pure pull based mode is interesting for audio applications, but not yet fully supported by all elements. Regardless of the scheduling, buffers always travel downstream (from sources to sinks).

Events are used for control flow between elements and from the application to the pipeline. Events inside the pipeline are bidirectional – some go upstream, some go downstream. E.g. seeking is done by using events.

Messages are used to inform the application from the inside of the pipeline. Messages are sent to a bus. There they are marshalled to the application thread. The application can subscribe to interesting messages. E.g. multimedia metadata (such as id3 tags) are send as a tag-message.

Queries are mostly used by the application to check the status of the pipeline. E.g. one can use a position-query to check the current playback position. Queries can also be used by elements, both up- and downstream.

Type	Direction
Buffer	Between elements
Event	From application to elements and between elements
Message	From elements to application via the message bus
Query	From applications to elements and between elements.

Table 1: Communication primitives

All the communication object are lightweight object (GstMiniObject). They support subclassing, but no properties and signals.

1.4 Multi threading

GStreamer pipelines always use at least one thread distinct from the main ui thread[1]. Depending on the scheduling model and pipeline structure more threads are used. The message bus described in the previous chapter is used to transfer messages to the applications main thread.

[1]This also applies to commandline applications

Together these mechanisms decouple the UI from the media handling.

Sink elements usually run a rendering thread. In pull based scheduling also all the upstream data processing would run in this thread.

In push based scheduling (which is the default for most elements), sources start a thread. Whenever data-flow branches out, one would add queue elements which again start new threads for their source pad (see Illustration 2 for an example).

Threads are taken from a thread-pool. GStreamer provides a default thread-pool, but the application can also provide its own. This allows (some) processing to run under different scheduling modes. The threads from the default pool are run with default scheduling (inherited from parent and thus usually SCHED_OTHER).

1.5 Plugin/Element API

GStreamer elements are GObjects subclassing GstElement or any other base-class build on top of that. Also structural components like GstBin and GstPipeline are subclassed from GstElement. GStreamer has many specialized base-classes, covering audio-sinks and -sources, filters and many other use cases.

Elements are normally provided by installed plugins. Plugins can be installed system wide or locally. Besides applications can also register elements directly from their code. This is useful for application specific elements. Buzztard uses this for loading sounds through a memory-buffer sink.

The use of the GObject paradigm provides full introspection of an element's capabilities. This allows applications to use elements in a generic way. New elements can be used without that the application needs special knowledge about them.

1.6 Plugin wrappers

The GStreamer plugin packages provide bridge plugins that integrate other plugin standards. In the audio area, GStreamer applications can use ladspa, lv2 and buzzmachine plugins. The bridge plugins register elements for each of the wrapped plugins. This eases application development, as one does not have to deal with the different APIs on that level any more.

1.7 Input and Outputs

GStreamer supports sources and sinks for various platforms and APIs. For audio on Linux these include Alsa, OSS, OSS4, Jack, PulseAudio, ESound, SDL as well as some esoteric options (e.g. an Apple Airport Express Sink). Likewise there are video sinks for XVideo, SDL, OpenGL and others.

1.8 Audio plugins

Besides the plugin wrappers described in section 1.6, the input/output elements mentioned in 1.7 and generic data flow and tool elements (tee, input/output-selector, queue, adder, audioconvert), GStreamer has a couple of native audio elements already. There is an equalizer, a spectrum analyser, a level meter, some filters and some effects in the gst-plugins-good module. The Buzztard project has a simple monophonic synthesizer (simsyn) and a fluidsynth wrapper in the gst-buzztard module. Especially simsyn is a good starting point for an instrument plugin (audio generator).

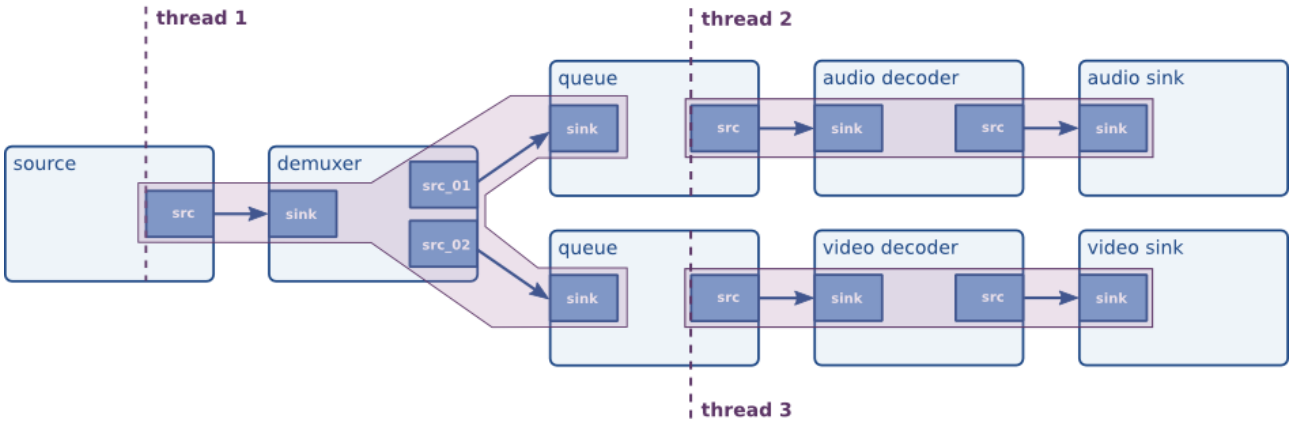


Illustration 2: Thread boundaries in a pipeline

1.9 A/V Sync and Clocks

Time is important in multimedia application. A common timebase is needed for mixing and synchronisation. A GStreamer pipeline contains a clock object. This clock can be provided by an element inside the pipeline (e.g. the audio clock from the audio sink) or a system clock is used as a fall back. The system clock is based on POSIX monotonic timers if available. Elements tag buffers with timestamps. This is the stream-time for that particular media object (see Illustration 3 for the relation of different timebases). Sink elements can now sync received buffers to the clock. If multiple sinks are in use, they all sync against the same clock source.

GStreamer also provides a network clock. This allows to construct pipelines that span multiple computers.

One can use seek-events to configure from which time to play (and until what time or the end). Seek-events are also used to set the playback speed to achieve fast forward or playing media backwards.

1.10 Sequencer

A sequencer records events over time. Usually the sequence is split into multiple tracks too.

GStreamer elements are easy targets for event automation. Each element comes with a number of GObject properties as part of its interface. The properties can be enumerated by the application. One can query data types, ranges and display texts. Changing the value of an element's property has the desired effect almost immediately. Audio elements update the processing parameters at least once per buffer. In video elements a buffers is one frame and thus parameter changes are always immediate.

The GObject properties are a convenient interface for live control. Besides multimedia applications usually also need sequencer capabilities. A Sequencer would control element properties based on time. GStreamer provides such a mechanism with the GstController subsystem. The application can create control-source objects and attach them to element properties. Then the application would program the controller and at runtime the element fetches parameter changes by timestamps.

The control-source functionality comes as a base-class with a few implementations in GStreamer core: an interpolation control source and an lfo control source. The former takes a series of {timestamp, value} pairs and can provide intermediate values by applying various smoothing functions (trigger, step, linear, quadratic and cubic). The latter supports a few waveforms (sine, square, saw, reverse-saw and triangle) plus amplitude, frequency and phase control.

The GObject properties are annotated to help the application to assign control-sources only to meaningful parameters.

[5] has a short example demonstrating the interpolation control source.

The sequencer in Buzztard completely relies on this mechanism. All events recorded in the timeline are available as interpolation control sources on the parameters of the audio generators and effects.

1.11 Preset handling

The GStreamer framework defines several interfaces that can be implemented by elements. One that is useful for music applications is the preset interface. It defines the API for applications to browse and activate element presets. A preset itself is a named parameter set. The interface also defines where those are stored in the file system and how to merge system wide and user local files.

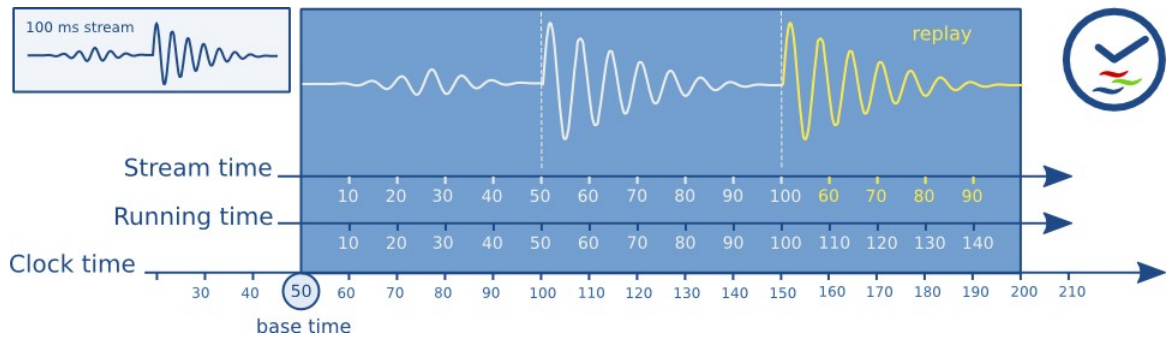


Illustration 3: Clock times in a pipeline

In Buzztard the preset mechanism is used to defines sound preset on sound generators and effects. Another use of presets are encoding profiles for rendering of content so that it is suitable for specific devices.

2 Development support

Previous chapters introduced the major framework features. One nice side effect of a widely used framework is that people write tools to support the development.

The GStreamer core comes with a powerful logging framework. Logs can be filtered to many criteria on the fly or analysed off-line using the gst-debug-viewer tool. Another useful feature is the ability to dump pipeline layouts with all kinds of details as graphviz dot graphs. This is how Illustration 1 was generated.

Small pipelines can by tested with gst-launch on the command-line. The needed elements can be found with gst-inspect or by browsing the installed plugins with its graphical counterpart gst-inspector.

Gst-tracelib can provide statistics and performance monitoring of any GStreamer based application.

3 Performance

One often asked questions is regarding to the framework overhead. This is discussed in the sections below, looking at it from different angles.

3.1 Startup time

When initializing the gstreamer library, it load and verifies the registry cache. The cache is a dictionary of all known plugins and the features they provide. The cache allows applications to lookup elements by features, even before their module is loaded.

The cache is built automatically if it is not present. This spawns a separate process that will load and introspect each plugin. Crashing plugins are blacklisted. Obviously this takes some time, especially if a large amount of plugins is installed.

This of course depends at lot on the amount of installed plugins (in my case: 235 plugins, 1633 features) and the system[2]. As can been seen from the above example, the 2nd round is quick.

[2] Lenovo T60 using a Intel® CPU T2400@ 1.83GHz (dual core)

```
> sync; echo 3 > /proc/sys/vm/drop_caches
> time gst-inspect >/dev/null 2>&1
real    0m2.710s
user    0m0.084s
sys     0m0.060s
> time gst-inspect >/dev/null 2>&1
real    0m0.074s
user    0m0.036s
sys     0m0.040s
```

Example 1: Registry initialisation times

3.2 Pipeline construction

Initial format negotiation can take some time in large pipelines. This gets especially worse if a lot of conversion elements are plugged (they will increase the number of possible formats). One way to address this, is to define a desired default format and plug conversion elements only as needed.

```
> export GST_DEBUG="bt-cmd:3"
> ./buzztard-cmd 2>&1 -c p -i Aehnatron.bmw |
grep start
0:00:00.038075793 first lifesign from the app
0:00:02.583626430 song has been loaded
0:00:03.388285111 song is playing
```

Example 2: Playback start time for a Buzztard song

Example 2 is the authors poor mans approach to meassure the time to go to playing. The timings already use the optimization suggested in the previous paragraph. The song used in the example uses 44 threads and consists of 341 GStreamer elements structured into 62 GstBins.

3.3 Running time

The actual run time overhead is quite low. Pushing a buffer, will essentially just call the process function on the peer pad. Likewise it happens in pull mode. There is some overhead for sanity checking and renegotiation testing.

```
> time ./buzztard-cmd -c e -i Aehnatron.bmw -o
/tmp/Aehnatron.wav
11:48.480 / 11:48.480

real    0m29.687s
user    0m41.615s
sys     0m4.524s
```

Example 3: Render a Buzztard song

Example 3 show how long it takes to render a quite big song to a pcm wav file. Processing nicely saturates both cpu cores. This benchmark uses the same song as in Example 2.

4 Conclusion

GStreamer is more that just a plugin API. It implements a great detail of the plugin-host

functionality. To compare GStreamer with some technologies more known in the Linux Audio ecosystem - one could understand GStreamer as a mix of the functionality that e.g. Jack + lv2 provide together. GStreamer manages the whole processing filter graph, but in one process, while jack would do something similar across multiple processes. GStreamer also describes how plugins are run together, while APIs like ladspa or lv2 leave that freedom to the host applications.

The GStreamer framework comes with a lot of features needed to write a multimedia application. It abstracts handling of various media formats, provides sequencer functionality, integrates with the different media APIs on platforms like Linux, Windows and MacOS. The multithreaded design helps to write applications that make use of modern multicore CPU architectures.

The media agnostic design is ideal for application that like to use other media besides audio as well.

As a downside all the functionality also brings quite some complexity. Pure audio projects still need to deal with some aspects irrelevant to their area. Having support for arbitrary formats makes plugins more complex. Pull based processing remains an area that needs more work, especially if it should scale as well as the push based processing on multicore CPUs.

If there is more interest in writing audio plugins as GStreamer elements, it would also be a good idea to introduce new base-classes. While there is already one for audio filters, there is none yet for audio generators (there is a GstBaseAudioSrc, but that is meant for a sound card source). Thus simsyn introduced in chapter 1.8 is build from 1428 lines of C, while the audiodelay effect in the same package only needs 620.

5 Acknowledgements

Thanks go the the GStreamer community for their great passionate work on the framework and my family for their patience regarding my hobby.

References

[1] GStreamer Project. 1999-2010. *GStreamer: Documentation.* ww.freedesktop.org.

[2] Stefan Kost. 2003-2010. *The Buzztard project*. www.buzztard.org

[3] Stefan Kost. 24 June 2007 . *Fun with GStreamer Audio Effects*. The Gnome Journal.

[4] VAMP Project. 1999-2010. *The Vamp audio analysis plugin system*. http://vamp-plugins.org

[5] GStreamer Project. 1999-2010. *GstController audio example*. http://cgit.freedesktop.org/gstreamer/gstreamer/tree/tests/examples/controller/audio-example.c

Emulating a Combo Organ Using Faust

Sampo SAVOLAINEN
Foo-plugins
http://foo-plugins.googlecode.com/
sampo.savolainen@gmail.com

Abstract

This paper describes the working principles of a 40 year old transistor organ and how it is emulated with software. The emulation presented in this paper is open source and written in a functional language called Faust. The architecture of the organ proved to be challenging for Faust. The process of writing this emulation highlighted some of Faust's strengths and helped identify ways to improve the language.

Keywords

Faust, synthesis, emulation

1 Introduction

Faust[Yann Orlarey, 2009b] stands for Functional AUdio STream and as the name implies it is a functional language designed for audio processing. The Faust compiler is an intermediary compiler, which produces source code for a C++ signal processor class which is integrated into a chosen C++ architecture. This architecture file provides a run-time environment, or wrapper, for the processor. This wrapper can be for example a stand-alone Linux Jack application or an audio processing plug-in.

This paper describes how an emulation of a 1970's combo-organ was written in Faust. The Yamaha YC-20 is a fairly typical organ of its time, a transistor based relatively light instrument meant for musicians on the road. The organ is a divide-down design and its working principles are discussed in detail in section 2. The emulation is released as open source under the GNU General Public License (v3). In the spirit of open source, the working principles and decisions taken during writing the emulation are described in this paper for all to read.

The YC-20 was chosen to be emulated as its architecture is very different from typical software and virtual analog synthesizers. Instead of the complex controllability and routability of typical synthesizers, this organ requires a large number of fixed parallel processes and components. This makes for a good test of Faust's performance and parallelization capabilities. Access to an operational organ was also a factor in the choice, as it makes it possible to match the emulation quality against the original.

The contents of this paper is organized into four sections. The first section covers how the real organ operates. This is followed by a description of how the organ is emulated. The third section covers the performance aspects of the emulation. The last section offers analysis of Faust's strengths and gives proposals on how to improve the language.

2 YC-20

This section covers the operations of the organ in detail[Yamaha, 1969]. The information in this is section is later referred to and detailed further when discussing the emulated parts. Figure 1 shows a block diagram of the device.

2.1 Features

The organ has one physical manual with 61 keys with a switchable 17 key bass manual and two voice sections. Instead of drawbars like Hammond organs, the voices are controlled by a lever system used in Yamaha organs of the time. As the word drawbar is commonplace when discussing organ voicing, the word is used in place of lever for the voice controls. While the controls are potentiometers, they have notches to help the user achieve repeatable settings. Each drawbar lever in the organ has four positions (end stops and two notches) from off to full volume.

Section I has drawbars 16', 8', 4', 2-2/3', 2', 1-3/5' and 1' and section II has drawbars 16', 8', 4', 2' and a continuous brightness lever. The sections can be selected or mixed together using a continuous balance lever. Enabling the bass manual switches the lowest 17 keys (white on black) to bass section sounds with drawbars 16'

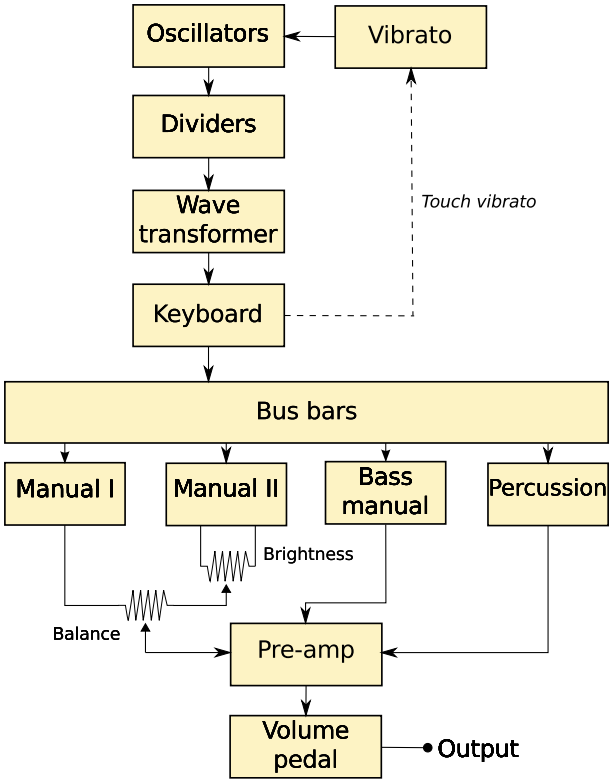


Figure 1: Block diagram of the YC-20

and 8'. There are controls for pitch, overall volume, bass volume, vibrato depth, vibrato speed and an obscure "touch vibrato" feature. The organ also has a single percussion drawbar and an integrated volume pedal. Like the drawbars, all vibrato controls have four notches.

2.2 Tone generation

The synthesizer architecture comprises of 12 oscillators, one per each note in a twelve-tone equal temperament (12-TET) scale. Each oscillator produces a sawtooth wave. For each oscillator, there are 7 frequency divider stages so each oscillator produces 8 voices totalling 96 voices. The dividers produce square waves.

These voices are next fed through a passive filter bank. The divided voices have both a high pass and low pass filters while the oscillator voices are only high pass filtered to remove bias. This totals 180 resistor-capacitor (RC) filter networks. Filtered voices are connected to switches on the actual keys of the keyboard. The keyboard is thus in fact a matrix mixer connecting the voices to bus bars. A key feeds each bus bar with the appropriate voice matching that key and octave, or the matching harmonic voice in case of the 2-2/3' and 1-3/5' bus bars.

2.3 Sections

The drawbar controls are potentiometers which control how much of each bus bar is mixed to each section of the organ. While in section I, drawbars are simply mixed together using resistors, Section II drawbars have more complex filtering. Each section II drawbar is divided into two streams, each filtered separately with RC networks. One stream is low pass filtered while the other is high pass filtered. The low passed and high passed signals are mixed through resistors to combine into bright and dull signals which are then buffered. The brightness potentiometer controls a mix of the two which, after further buffering, is the output of section II.

The key switches on the lowest 17 keys feed separate bass bus bars. The bass manual switch controls whether these bus bars are kept separate or mixed with the main bus bars. Thus, with the bass feature disabled, the bass keys work exactly as the rest of the manual. With the switch enabled, the bass bus bars are separate and only feed the bass section. It is worth noting that when enabled the bass section effectively discards the signal of bass bus bars 2-2/3', 2', 1-3/5' and 1'.

Bass section differs from the two main sections in that both bass manual drawbars (16' and 8') use a mix of multiple bass bus bars. The 16' drawbar is a mix of bass bus bars 16', 8' and 4'. The bus bars are mixed through different value resistors. 16' has the least resistance and 4' the highest resistance. The 8' drawbar is a mix of the 8' bass bus bar with lower resistance and 4' bass bus bar with higher resistance. While section I and section II drawbar control potentiometers are wired in a standard volume configuration, the bass manual drawbar controls are wired center-tap to source with the other tip grounded and the other tip as the output. This configuration makes the drawbar control not only affect how much of the drawbar is mixed in, but it also varies the impedance of the signal source. The different mixing resistors and the varying impedance of the drawbar controls, combined with a fixed capacitor to ground (after summing the drawbar signals) makes the network act as a fairly complex RC filter. This filter has a varying cutoff point and mix amount per bass bus bar.

2.4 Percussion

Percussion manual sounds are created by mixing together bus bars 1', 2-2/3' and 16' via resis-

tors. There is a substantially larger resistance on the 1' bus bar leading to less 1' voices in the percussion section. The volume of this signal is controlled by a simple envelope generator. This envelope generator is triggered by activity on the 1' bus bar. The envelope attack is instantaneous and the release time is fast while the sustain level is zero. This causes a fast attack sound, but the effect only works when no keys are pressed down before. In other words, the percussion effect is not heard, for example, when playing legato. However there is a substantial amount of bleed from the percussion section which is audible when all other drawbars are off.

If the bass manual is enabled, it disengages the bass bus bars from the main bus bars. Thus the bass keys will not mix sounds onto the 1' bus bar. Therefore playing the bass manual will not trigger the percussion and percussion sounds will trigger even when bass keys are held down.

2.5 Main output

The main output of the device is a mix of sections I and II, the percussion part and the bass manual. The mix between sections I and II is controlled by the balance lever (a potentiometer). The amount of percussion mixed in is controlled by the percussion drawbar. Bass manual is summed into the main output via a potentiometer controlling the bass volume. This combined signal is then preamplified. The preamplified output can then be attenuated by the main volume potentiometer of the device and the volume pedal. The volume pedal action controls a mechanical shutter between a small light bulb and a light dependent resistor.

3 The emulation

The emphasis was on creating a playable instrument which sounds like the original organ. A playable emulation needs to be able to work at low latencies and it needs to be efficient enough to be ran on commodity hardware. The emulation tries not only to emulate the ideal circuit but also some of the inaccuracies in the real instrument.

3.1 Why Faust?

Instead of taking an approach where sounds are synthesized only when needed, this emulation keeps all oscillators, dividers and filters running all the time – just like the real organ. Faust was chosen as the programming language to emulate the organ with, as Faust's functional semantics fit well with having all processing running at all times. Faust also makes it trivial to have a large amount of streams flowing from one circuit to another. This emulation was also intended as a test of Faust in a real world use.

3.2 Tone generation

The 12 main oscillators produce sawtooth waves. They share a common, varying bias voltage (see section 3.3) which affects the oscillator frequencies. The main oscillator voices are divided by an array of flip flop circuits each dividing the frequency in half and the next one dividing the previously divided voice. The flip flops produce square wave signals. This means each oscillator produces a total of 8 phase-synchronized voices, each one an octave down from the previous voice.

As the main oscillator frequencies are high (4–8kHz), a naive oscillator implementation would suffer massively from aliasing when using typical sampling rates (44.1–96kHz). Naturally, also the dividers would suffer from aliasing as well. After evaluating different methods to band-limit the signals, the PolyBLEP[1][Välimäki and Huovilainen, 2007] method was chosen. While BLEP[Brandt, 2001] would produce less aliasing components, it is computationally more expensive. More importantly, the BLEP step function is currently impossible to calculate in Faust as it requires using Fourier and inverse Fourier transfer functions. The quality of BLIT-SWS[2][Stilson, 1996] is good at high frequencies, but it produces aliasing components below the fundamental frequency[Välimäki and Huovilainen, 2007]. Also, BLIT-SWS is problematic when it comes to band-limiting hard synchronized oscillators[Brandt, 2001] which is one strategy to emulate the divider circuits.

Second, third, and fourth order polynomial residual functions were evaluated for the PolyBLEP. The higher order residual functions reduced high frequency aliasing only slightly more compared to the second order function. Furthermore a band-limited signal using the second order function has considerably less aliasing components below the fundamental frequency when compared to the third and fourth or-

[1]BLEP = band-limited step function. PolyBLEP uses a step function derived from a simple polynomial

[2]BLIT = band-limited impulse train, SWS refers to the use of windowed sinc functions

der functions. This confirmed previous findings[Pekonen, 2007]. The chosen second order polynomial residual function *r(t)* is shown as equation 1.

$$r(t) = \begin{cases} t^2/2 + t + 1/2, & \text{if } -1 \leq t \leq 0 \\ -t^2/2 + t - 1/2, & \text{if } 0 < t \leq 1 \end{cases} \quad (1)$$

The divider circuits are emulated as slaved oscillators. The main oscillator function outputs both the signal and phase information. One divisor function divides the phase and feeds this to a slave oscillator. The slave again produces both a signal and phase information. The complete divider circuit for one oscillator is seven such divisor functions piggybacked.

PolyBLEP works by adding the polynomial band-limited step function to two samples: the sample before and after a discontinuity in the signal. To achieve this, the implementation delays its output by one sample. At each frame, the phase is inspected. If the phase passed a discontinuity in the waveform, the PolyBLEP function is evaluated for and added (rising wave) or subtracted (falling wave) for the previous sample and the current sample. As Faust lacks true branches, all possible branches of conditional statements are always calculated. Table 1 shows the amount of residual function evaluations required under different conditions. The table shows that branching would be far superior to any non-branching solution. This is because without branching the amount of PolyBLEP evaluations done is purely a function of the sampling rate while the amount of required evaluations is a function of the signal frequency.

To be able to run the emulation in real time, a truly branching solution had to be developed as an external C++ function. This was however easy to integrate with the Faust processing as the *ffunction* operator lets one use externals just as native functions.

The filter bank contains tailored filtering for each of the 96 voices produced by the oscillators and dividers. The main oscillators only have a single capacitor in series. This is emulated as a high-pass RC filter using an approximation of the next stage impedance as the load resistor value. The first four divided signals have a trivial RC low-pass filter before a single capacitor in series similar to the the main oscillator filter. The lowest three voices are filtered like the previous four, except that there is a resistor to ground after the series capacitor. This alters the next stage impedance compared to the other filters.

Frequency	(A)	(B)	(C)
5kHz	352 800	44 100	10 000
1kHz	352 800	44 100	2 000
500Hz	352 800	44 100	1 000

Table 1: Amount of PolyBLEP calculations per second for a square wave at Fs = 44.1kHz. (A) an implementation where per each frame the PolyBLEP is evaluated for both the previous and current sample for both discontinuities. The branched nature of the residual function multiplies this number further by a factor of two. (B) an ideal implementation without branching where only one PolyBLEP is evaluated per frame. (C) is the number of required PolyBLEP calculations.

Filtering is very complex to emulate exactly right, as one voice might be connected to multiple bus bars. This varies the high-pass filter load resistance and therefore affects the filter cutoff frequency depending on what keys are depressed. This is not emulated. Instead, the high-pass filter load is estimated based on the expectation that there is only one connection to a bus bar.

3.3 Oscillator bias

Each oscillator produces a saw wave at a different frequency. The frequencies are chosen from a 12-TET scale. However all oscillators share a single bias voltage affecting their frequency. This bias voltage is controlled by the vibrato circuit, touch vibrato and the master pitch potentiometer. The master pitch potentiometer is emulated by a simple slider widget with a DC output. Touch vibrato would be controlled by horizontal movement of the keys. As such MIDI keyboards are extremely rare[3] this feature was left out of the emulation.

Vibrato control voltage is created by a simple sine wave oscillator. The vibrato speed controls the speed of this oscillator. The vibrato speed range (5-8Hz) was measured from the real device. The vibrato depth is simply an attenuation control for the vibrato oscillator output. Vibrato depth range was empirically selected. The depth control deliberately never goes to

[3]Only one keyboard was found claiming such capability: the Yamaha STAGEA ELS-01C.

zero, thus the vibrato has a miniscule effect on the bias voltage even when turned down.

3.4 Keyboard matrix mixer

The keyboard matrix mixing, while a relatively simple part, contains many separate operations. Each key is a Faust button[4] connecting multiple voices to different bus bars. This means each key button is used as a multiplier for 7 different voices (one per bus bar). So the signals on the emulated bus bars are the outcome of $61 * 7 = 427$ discrete multiplications summed to appropriate busses. There is also added logic for the bass manual where the resulting downmixes from the 17 bottom keys are fed to either the main bus bars or the bass bus bars.

The matrix mixer could be written as sets of floating point tables multiplied together. In Faust however, there are no such array operators. This results in a number of separate multiplications and sums which are difficult for the Faust compiler to optimize, as vector operations are unusable if the data is not in ordered arrays.

The key action is not band-limited. The key switches on the real organ are simple switches connecting voices to bus bars. This operation causes clicks in the real organ as switches open and close. The naive non-band-limited key action of the emulation matches the sound of the real organ surprisingly well. However, this is something that could be improved at a later stage.

Writing the keyboard matrix mixer also revealed an issue with the service manual. The service manual indicates wrongly which voices are connected to the harmonic bus bars (2-2/3' and 1-3/5'). The manual states that 2-2/3' is connected to a voice five semitones higher than the voice connected to 4' and 1-3/5' with a voice eight semitones higher than 2'. The real organ connects voices seven and three semitones above respectively. The emulation follows the real organ instead of the service manual.

3.5 Section I

Section I is a mix of the main bus bars mixed together through the drawbar potentiometers and additional resistors. There is no extra filtering applied to the signal. Thus the only part left to emulate for section I sounds is the drawbar controls.

The drawbar potentiometers do not follow the typical linear or logarithmic tapers. The real organ was measured while playing a stable note with all drawbars off except one drawbar which was tested in all three on positions. The off position is expected to be -inf. Compared to full volume, the two middle positions were measured at -12dB and -18dB.

[4]Button output signal is 1 when it is being pressed down and otherwise 0.

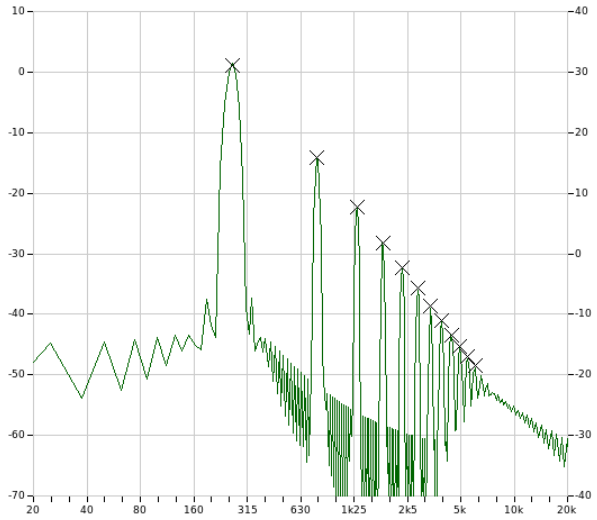


Figure 2: Comparison between emulated and real manual I drawbar 8' sounds.

This can be translated into a continuous transfer function (see equation 2). p is the position of the lever from 0 (off) to 1 (full volume). The function returns a gain coefficient usable in the emulation. The function emulates the taper and can be used with slider controls provided by the Faust architecture. Further work on a graphical user interface should provide four position levers.

$$\begin{aligned} \text{coeff}(p) &= 2.81p^3 - 2.81p^2 + p \\ \text{coeff}(0) &= -\text{inf dB} \\ \text{coeff}(1/3) &\approx -18.05 \text{ dB} \\ \text{coeff}(2/3) &\approx -12.03 \text{ dB} \\ \text{coeff}(1) &= +0.00 \text{ dB} \end{aligned} \tag{2}$$

Figure 2 shows C2 played on section I with only the 8' drawbar engaged. The line depicts the frequency spectrum of the emulation output. The crosses show peaks measured from a YC-20 organ using the same settings.

3.6 Section II

As described earlier in section 2.3, Section II has a controllable brightness feature. The variable brightness is done by dividing each bus bar into two streams, one of which is high-passed and the other low-passed. The streams derived

from different bus bars are then mixed together into bright and dull streams. The brightness control is a simple balance control between the two streams.

The high-pass filtering is done by a two stage RC filter with a resistor in parallel with the capacitor of the first filter. Thus the filter is effectively a shelving high pass filter. The shelf is emulated by calculating a voltage divider between the parallel resistor and the resistor in the RC high pass filter. The voltage divider gives a gain coefficient C which is used to mix together the high-passed and unfiltered signals. Unfiltered signal is multiplied by C while the filtered is multiplied by $1\text{-}C$ and the results are added together. This keeps gain at high frequencies at 0dB as with the original passive filter. The two passive filters also affect each other and could not be emulated by simply chaining two digital filters. The load applied by two filter stages is emulated by dividing the resistance of the second filter by two. This doubles the cutoff frequency of the filter. This method was found by trial and error. Figure 4 shows the block diagram of the complete filter.

The low-pass stage is much more straightforward. Filtering consists of two chained RC low-pass filters. However, the best match with the original organ was achieved by not compensating for load posed by the chained filters. This filter stage is emulated by two digital RC low pass filters in series.

Figure 3 shows C2 played on a fully bright section II with only the 8' drawbar engaged. This measurement was done at the same overall level as measurements in figure 2. The measurement shows a slight overall volume difference and that the section II high-pass filters are not perfect. However, while harmonics for this particular sound do not line up exactly, the balance between harmonics is good enough. The harmonics of many other notes (for example C3) line up perfectly.

3.7 Percussion

The percussion uses the signal on the 1' bus bar to trigger an envelope generator. There is a root-mean-square envelope follower on the 1' bus bar and the envelope is triggered when the follower exceeds a set threshold. The envelope generator starts off with a signal where a one frame unit impulse is created at the trigger point. This impulse signal is low-pass filtered to match the measured percussion envelope. This envelope is then used to control the volume of a mix of the 1', 2-2/3' and 16' bus bars. Some bleed is added to the envelope generator output to emulate the bleed exhibited by the real organ.

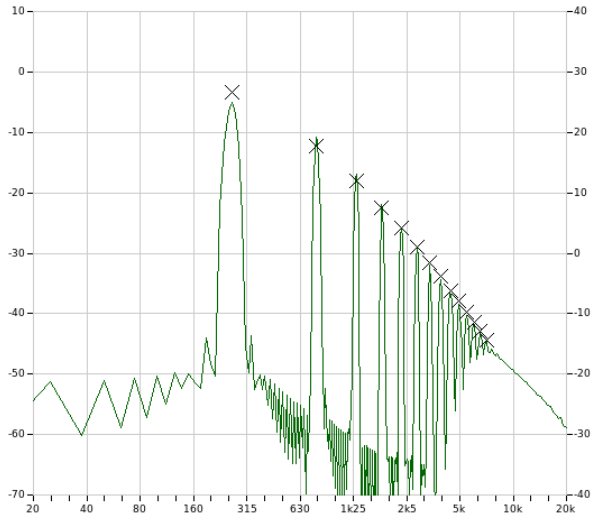


Figure 3: Comparison between emulated and real bright section II drawbar 8' sounds.

3.8 Bass manual

Emulating the bass section perfectly is a complex task due to how mixing resistors and the variable impedances of the potentiometers affect the RC filter. Measurements from the real organ however suggest that the low frequency fundamentals dominate the signal and an acceptable emulation is achieved by carefully controlling the mix of the bass bus bars and by using a single fixed RC low-pass filter. This one filter is applied to a mix of both bass drawbars.

4 Performance

Running this emulation requires a substantial amount of processing power. There are 96 oscillators working all of the time and there are hundreds of filters being applied at many stages of the design. Faust is designed to easily harness multiple processing units in parallel. The design of the emulation has a lot of potential for parallelization and as such, it should work as a good real world test for Faust's SMP features.

The performance numbers shown here are measured using a modified version of the Faust benchmarking suite. These numbers are estimates of achieved memory bandwidth in megabytes of audio data generated per second. The Faust benchmark uses this measurement, as memory bandwidth on SMP systems is a pre-

cious commodity. In the case of the YC-20, the processing creates much more data than what comes out of its single mono output. Thus these figures can only be compared to each other.

Tests were done on a Dell D820 laptop with a Core 2 Duo T7400 (2.16GHz) processor using the internal sound interface. The operating system was 32 bit Ubuntu 9.10. The processor frequency governor was switched to performance on both cores before running the tests. The kernel was a standard Ubuntu package, 2.6.31-19-generic and the used gcc version was 4.4.1-4ubuntu9. Benchmarks were done on YC-20 code revision 227[5]. Figures in column C are from a slightly modified version of the YC-20 code and it shows the performance impact of non-branching PolyBLEP calculations. The results in MB/s are shown in table 2.

Faust benchmark	(A)	(B)	(C)
scalar (galsascal)	0.40	0.43	0.28
vectorized (galsavec)	0.58	0.59	0.35
vectorized 2 (galsavec2)	0.62	0.62	0.34
OpenMP (galsomp2)	0.51	0.55	0.29
scheduler (galsasch)	0.90	0.88	N/A
scheduler 2 (galsasch2)	0.93	0.89	N/A

Table 2: The tests were ran with different sets of gcc parameters:
(A) Branching C++ PolyBLEP evaluations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize
(B) Branching C++ PolyBLEP evaluations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize -fgcse-sm -funsafe-math-optimizations
(C) Divider slave oscillators use non-branched PolyBLEP operations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize -fgcse-sm -funsafe-math-optimizations
GCC was unable to compile the scheduler versions of the emulation with non-branching PolyBLEP operations.

The multi-threaded scheduler tests were notably faster than other benchmarks. There is an increase of 50% in memory bandwidth when compared to vectorized single thread tests. This data is backed by tests done by measuring the performance of the YC-20 emulation running as a jack application. The measurements shown in table 3 were done by running the emulation compiled with different flags and observing the DSP percentage meter in qjackctl. This meter tells how much of the time between jack engine callbacks is spent doing processing inside jack clients.

Tests	Faust flags	Load
scalar	none	$\sim$ 49%
vectorized	-vec -vs 32	$\sim$ 34%
scheduler	-sch -g -vs 256	$\sim$ 29%

Table 3: All tests were compiled with the following gcc flags:
-O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize.
Jack 1 was running real-time with a dummy back-end simulating a 48kHz sample rate with 512 frame buffers.

5 Conclusions

The software presented in this paper emulates the YC-20 organ well. While some parts of the emulation could benefit from more polish, other parts of the emulation can sound very much like a real YC-20. A better quality anti-aliasing method could be beneficial, although these methods would come at the expense of CPU cycles. The real organ has considerable bleed and inconsistencies which might also be worth emulating.

5.1 How to improve Faust

As discussed in section 3.2, and shown in the benchmarks in section 4, there is a strong case for a truly branching select operation. However, the semantics of Faust requires all of the processing graph to be evaluated for every frame. Truly branching operations would cause some function evaluations to be skipped. The operating semantics are however compatible with branching operations if the skipped functions are stateless. This is because if the function has no state, the next evaluation of the function does not depend on previous evaluations.

There are two ways truly branching operations could be added to Faust: either select2 and select3 calls with no stateful function branches are compiled automatically as truly branching or a new branching select call is created. Both cases require the specification of rules on what a stateless function is. Also, the compiler must

[5]Subversion repository available at http://foo-plugins.googlecode.com/svn/trunk/

check whether the functions being skipped over comply to these specifications before allowing compilation of such code.

This logic could also be generalized to multiplication functions. Function $f(x)$ in equation 3 is an example of a function where evaluating $t2(x)$ can be skipped if $x \leq 0$ as $t1(x)$ evaluates to 0 for $x \leq 0$ and $t2$ is stateless.

$$\begin{aligned} t1(x) &= (x > 0) \\ t2(x) &= x^2 - fmod(x, 1.0) \\ f(x) &= t1(x) * t2(x) \end{aligned} \tag{3}$$

The compiler error messages are often unhelpful. They do not always specify in which file the compilation error occurred. Optimally the compiler should identify the file name, line number and if possible, the name of the function the error occurred in. One specific issue with the compiler messages is worth giving special attention to. It is cases where the number of inputs and outputs of functions in a sequential composition do not match. In such cases, instead of printing the names of the offending functions, Faust outputs a quite exhaustive description of both functions. For example, if the YC-20 emulation would fail to cut the phase output of the last divider (dividers would then output a total of 108 instead of 96 streams), the resulting error message is 67 kilobytes.

For signal processing, the lack of support for Fourier transforms restricts what problems can be solved. As mentioned in section 3.2, the better band-limiting method (BLEP) is not possible to implement in pure Faust. Fourier transforms would naturally be useful for a variety of other tasks as well. Fortunately there has been promising news of multi-rate support for Faust which would allow processing such as Fourier transforms.

As procedural programming is the prevailing model of programming, it is safe to say many potential users of Faust are familiar with only that model. This applies pressure to the quality of documentation. The current documentation should abbreviate the definitions of the more advanced features of the language. Especially recursion and the rdtable and rwtable functions could benefit from better explanations. Currently the features are almost side-stepped and almost nothing but their syntax is described.

Implementing the matrix mixer required a lot of hand-crafting. Having a concept of indexable arrays would help such operations. If the 96 inputs of the keyboard mixer and the buttons representing the keys of the organ could be arranged into arrays, the mixing could be done algorithmically using the aggregate functions such as sum and vector multiply. Such operations also open up ways to further optimize the created C++ code. As it stands, the keyboard mixing is compiled into a large amount of separate discrete multiplication and addition operations which can not be vectorized.

The Faust code shown as equation 4 is ambiguous, but can be compiled without any warning. This can lead to anywhere from unexpected results to complete failure. It would be good form for the Faust compiler to at least warn the user of the situation.

$$\begin{aligned} f(x, y) &= x + y \\ with\{x &= y * y;\}; \end{aligned} \tag{4}$$

5.2 Benefits of Faust

Functional thinking suits audio applications very well as solutions to problems can often be expressed as mathematical equations. This model sidesteps many of the issues procedural programs face, as the processes can be written at a higher abstraction level. Unlike with procedural languages, Faust allows signal processing programmers to not worry about buffers, their lengths or loop structures or real-time thread constraints. This also saves time for the programmer as less time is spent debugging issues not directly connected with the signal processing task at hand.

Faust code is easy to keep readable as the syntax offers tools, such as sequential composition, which help keep functions simple and concise. See listings 1, 2 and 3 for an example. While listing 2 is pretty compact, one should note that functions are presented to the reader in reverse order and it is also difficult to see to which function return value the multiplication is applied to.

```
float dostuff(float f) {
  f = func1(f);
  f = func2(f);
  f = f * 2.5;
  f = func3(f);
  return f;
}
```

Listing 1: Consecutive calls

```
float dostuff(float f) {
  return func3( func2( func1(f)) * 2.5 );
```

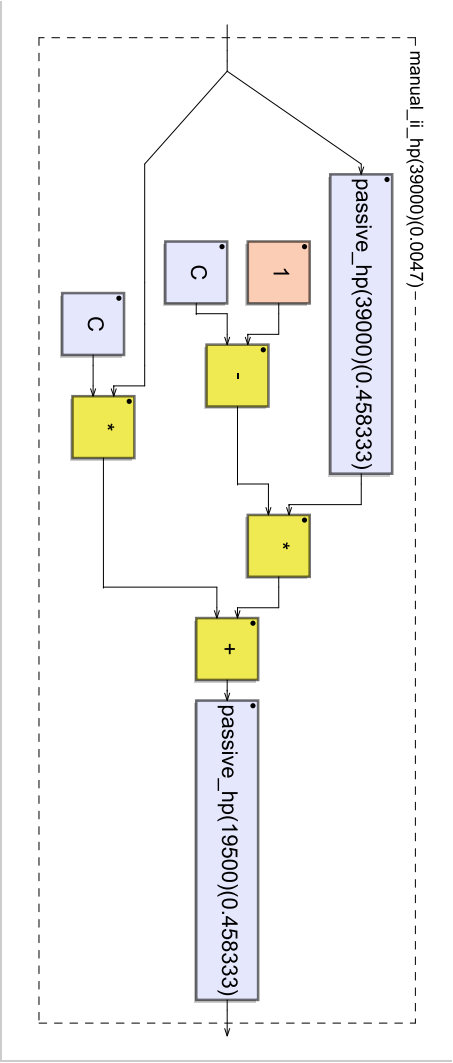


Figure 4: Faust SVG output of section II high pass filtering for a single drawbar. The parameters for the filters are resistance in ohms and capacitance in microfarads.

```
}
```

Listing 2: Enclosed statements

```
dostuff = func1 : func2 : *(2.5) : func3;
```

Listing 3: Faust sequential composition

The SVG output of the Faust compiler is also worth mentioning. The compiler can create hyper-linked SVG documents which depict the process function. This document can be an excellent learning and debugging tool as it shows how the compiler understood the source code. Figure 4 shows an example of this output.

The stream concept also makes code reusable. When combining procedural code from multiple sources you often need to either convert data types between the different modules or refactor the modules to match. Faust processors naturally fit with each other. This model works very well with open source as it makes it easier to spread good ideas and implementations. It might even lead to a resource library of truly reusable components usable with any Faust project – as long as the licenses are compatible.

Faust also provides automatic parallelization and vectorization. This allows all Faust programs to benefit from these advanced optimization methods which usually require expert programming skills to implement. These capabilities have been discussed in detail by Orlarey, Fober and Letz[Yann Orlarey, 2009a]. Faust also allows the developer to easily test and compare different optimization methods without refactoring their code.

6 Acknowledgements

Thanks to Petri Junno for support and all the help with analyzing the original organ. I would also like to thank Sakari Bergen, Torben Hohn and Yann Orlarey for providing advice. Thanks also to Fons Adriaensen for Japa which was an excellent tool for comparing the emulation to the real organ.

References

Eli Brandt. 2001. Hard sync without aliasing. In *Proceedings of the International Computer Music Conference (ICMC)*, Havana, Cuba, September.

Jussi Pekonen. 2007. Computationally efficient music synthesis – methods and sound design. Master's thesis, TKK Helsinki University of Technology.

Tim Stilson. 1996. Alias-free digital synthesis of classic analog waveforms.

Vesa Välimäki and Antti Huovilainen. 2007. Antialiasing oscillators in subtractive synthesis. *Signal Processing Magazine, IEEE*, 24(2):116–125.

Yamaha, 1969. *YC-20 Service Manual*. Yamaha Corporation.

Stephane Letz Yann Orlarey, Dominique Fober. 2009a. Adding automatic parallelization to faust. In Grame, editor, *Linux Audio Conference 2009*.

Stephane Letz Yann Orlarey, Dominique Fober. 2009b. *FAUST : an Efficient Functional Approach to DSP Programming*.

LuaAV: Extensibility and Heterogeneity for Audiovisual Computing

Graham WAKEFIELD and **Wesley SMITH** and **Charles ROBERTS**
Media Arts and Technology, University of California Santa Barbara
Santa Barbara, CA 93110,
USA,
{wakefield, whsmith, c.roberts}@mat.ucsb.edu

Abstract

We describe LuaAV, a runtime library and application which extends the Lua programming language to support computational composition of temporal, sound, visual, spatial and other elements. In this paper we document how we have attempted to maintain several core principles of Lua itself - extensibility, meta-mechanisms, efficiency, portability - while providing the flexibility and temporal accuracy demanded by interactive audio-visual media. Code generation is noted as a recurrent strategy for increasingly dynamic and extensible environments.

Keywords

Audio-visual, composition, Lua, scripting language

1 LuaAV

LuaAV is an integrated programming environment based upon extensions to the Lua programming language enabling the tight real-time integration of computation, time, sound and space.

LuaAV has grown from the needs of students and researchers in the Media Arts & Technology program at the University of California Santa Barbara; its origins lie in earlier Lua-based audio and visual tools [20] [15] [22]. More recently it has formed a central component of media software infrastructure for the AlloSphere [1] research space (a 3-storey immersive spherical cave-like environment with stereographic projection and spatial audio).

Various projects built using LuaAV have been performed, exhibited or installed internationally, for scientific visualization [1], data visualization [13], immersive generative art [21], game development[1], live-coding (Figure 1) and audiovisual performance[2].

LuaAV is available under a UC Regents license similar in nature to the BSD license[3].

[1] http://www.charlie-roberts.com/projects/circles/
[2] http://www.mat.ucsb.edu/~whsmith/Synecdoche/
[3] http://lua-av.mat.ucsb.edu/

Figure 1: LuaAV in a live-coding performance (Asterisk: performed by A McLeran and G Wakefield, 2009).

2 LuaAV philosophy

The broad attitude taken in the development of LuaAV draws inspiration from the Lua programming language itself: extensibility, meta-mechanisms, efficiency, and portability[4].

2.1 Extensibility

Lua is described as an 'extensible extension language' [8]: a configuration language to embed within and extend the control of kernel application or library code (which is typically written in a statically compiled language such as C). LuaAV follows this methodology directly: the core engine of LuaAV is a library of code (*libluaav*) intended to be embeddable within applications and application plugins, embedding code to manage instances of Lua interpreters, schedulers, an audio driver, and basic communication protocols (MIDI and OSC). Most aspects of the core library have both C and Lua programming interfaces.

[4] The reasoning behind the choice of Lua has been documented in prior publications, particularly [17].

The LuaAV application embeds *libluaav* and adds to it a GUI for managing active script states, a CodePad for adding code to script states at run-time, and a Windowing/Menu system. Most of these components are also scriptable; indeed much of the application logic of LuaAV itself is written using embedded scripts.

2.2 Meta-mechanisms

A fundamental Lua concept is the provision of low-level meta-mechanisms for implementing features, rather than a built-in fixed set. Lua's meta-mechanisms bring an economy of concepts while allowing the semantics to be extended in unconventional ways. We find this open-ended philosophy appropriate to our research domain of computational composition.

We attempt re-use the mechanisms provided by Lua itself in a consistent and predictable manner. As examples: the addition of temporal scheduling in LuaAV is implemented as an extension of the existing coroutine construct in Lua; new functionality is added to the runtime environment using the existing Lua module system; and many of these capabilities are specified using the existing data description and metamethod features of Lua.

2.3 Efficiency

Lua is widely acknowledged as amongst the most efficient of interpreted or scripting languages, however there is still an order of magnitude of performance cost relative to statically compiled code: the price paid for dynamic flexibility that interpreters offer. For this reason, the core scheduler, drivers and other elements of the *libluaav* runtime library are coded in C/C++. Nevertheless, in order to overcome the usual trade-off between flexibility and efficiency, we have begun to leverage of run-time compilation to machine code within LuaAV.

On i386 platforms, the Lua core of libluaav is replaced with the LuaJIT [11] interpreter and trace compiler, allowing performance approximating C for certain algorithms, and significant performance boost over regular Lua even in interpreted mode.

LuaJIT grants efficiency for pure Lua source code, however it cannot optimize over the C/Lua boundary. For those aspects of an application that talk to existing C code, we have been investigating the potential of LLVM/Clang [9]. We have developed a binding to a large proportion of the LLVM/Clang API from within Lua[5], such that abstract syntax trees as Lua data-structures, or pure C code strings, can be JIT-compiled and linked back into the application as it proceeds, under the direction of an executing Lua script.

2.4 Portability

It is near impossible to portably support the complex demands of multimedia applications because of the diversity of platforms and their dependencies. LuaAV currently targets recent Linux and OSX platforms, however we strive to use established, stable cross-platform libraries where possible (such as PortAudio/JACK, the Apache Portable Runtime, etc), or else provide abstraction layers for platform-specific code as appropriate (for example, the LuaAV application Windowing and GUI is written using a common abstraction layer over Qt for Linux and Cocoa for OSX).[6]

3 Related work

LuaAV is one of a family of audio/visual applications in which the primary interface is an embedded programming language, including SuperCollider [10], Impromptu [2], Fluxus [7], ChucK [23], and many more. All of these applications can all be used within a performative context, such as live-coding [3].

Impromptu, based on the Scheme programming language, is an OS X only environment that was originally created for audio manipulation; it has been extended to also include visual programming. One element of Impromptu that is of particular interest is its multi-user runtime; a single networked Impromptu environment can be accessed and manipulated by multiple users concurrently. We are actively developing a similar capability in LuaAV to satisfy multi-user demands in the AlloSphere.

Fluxus is another Scheme based platform, however it is primarily geared towards visual composition and is cross-platform. The Fluxa add-on module adds basic audio synthesis and playback capabilities to the Fluxus environment; however it is limited in terms of the number and breadth of unit generators that it provides.

[5] http://code.google.com/p/luaclang

[6] Nevertheless, the broad scope of the LuaAV application implies many non-trivial dependencies. We currently include a Lua-based build tool and command-line scripts to try to make installation on Linux more fluid.

ChucK is a live coding language geared towards audio with fine scheduling between synthesis and control ('strongly timed'). LuaAV's scheduling system offers similar control, which will be described in detail below. ChucK's visual capabilities only extend to one of its development environments, the Audicle, and primarily revolve around visualizing currently running audio processes.

SuperCollider is also predominantly a system for audio synthesis and scheduling. Its language is strongly inspired by Smalltalk, whose dynamism provides many possibilities for modifying running programs in real time. SuperCollider can be extended with downloadable modules ('quarks'), some including graphical capabilities[7].

4 LuaAV Implementation

4.1 Script states

Opening a script in the LuaAV application creates a *luaav_state* object, a *libluaav* wrapper around a Lua interpreter state with additional components:

- a logical clock
- a scheduler queue with pending events and coroutines
- a graph of audio processes (*Synths*)
- a bidirectional message queue (for communication with the audio graph)
- a memory pool for C-allocated objects associated with the lifetime of the *luaav_state*

An opened script can be closed or reloaded from within the LuaAV application, and its source can be viewed by opening the default external editor. The file modification date of the script file is monitored by LuaAV and the script automatically reloaded if changed; thus users can edit scripts in their preferred editor and see the results updated in LuaAV immediately.

Scripts can also be edited via the integrated CodePad (Figure 2) which was added to incorporate support for live coding practices into LuaAV. Code entered into the CodePad does not reload the script; rather it is injected into the *luaav_state* without interruption. Important features of the CodePad include:

- syntax highlighting via the Leg[8] parsing expression grammar
- the ability to edit multiple scripts concurrently in tabs or multiple windows
- the ability to execute a selected portion of a script
- basic visual error reporting

[7] e.g. http://sourceforge.net/projects/scgraph/

[8] http://leg.luaforge.net/

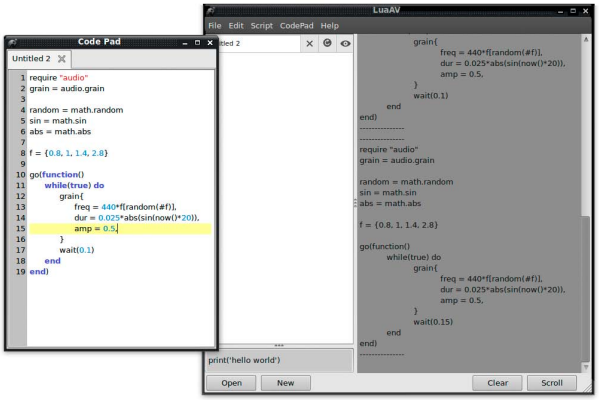


Figure 2: The LuaAV application running on Ubuntu 9.10, with the CodePad view open.

4.2 Scripting in real-time

A central problem of interactive computing applications is the translation from the abstract temporality of programming to the concrete and often unpredictable behavior of real-time behavior and interaction. The articulation of structure goes beyond matters of efficiency to demand:

- capacity to maintain required state until the appropriate moment (dynamic memory management)
- flexibility to re-activate maintained state at unpredictable moments (re-entrancy)
- ability to delay activity until a chosen or appropriate time (event ordering, scheduling)
- ability to specify events in measures of real-time (clocks, event spacing) as well as logical/causal relations (event handling)

Lua already offers excellent re-entrancy and dynamic memory management: user-driven calls and library callbacks can be made into an interpreter instance after a script has executed, and variables will remain alive so long as they are accessible. This re-entrancy extends to the

implementation of coroutines[9] in Lua in which lexically scoped local variables will remain 'alive' for as long as the coroutine block runs or awaits to be resumed.

4.2.1 Metrics, scheduling and concurrency

While Lua ensures deterministic sequencing of instructions, it lacks is a sense of temporal metric. Adding this metric is one of the roles of *libluaav*. Within a script in LuaAV, we can ask for the current time using the *now()* function. The time returned is logical time for the *luaav_state* scheduler, which is anchored in real-time by reference to the audio sample counter.

To grant explicit scriptable control over the scheduling capabilities of a *luaav_state*, we have extended the coroutine mechanism to allow yielding control to the scheduler by means of a *wait()* function. The arguments to *wait* can be a duration (after which the coroutine will resume)[10] or an arbitrary string (the name of an event trigger to wait for). A scheduled coroutine can be directly created using the *go* function, whose arguments can specify a duration or event to wait for before starting the coroutine, and a function and arguments to form the body of the coroutine. Of course, any valid Lua code can be placed within a coroutine; in fact the entire script itself is also a coroutine and can *wait()* and check *now()* as needed.

A code example may speak a thousand words:

```
-- define a function to print a message
-- repeatedly, every 1 second
function printer(message)
  while true do
    print(message)
    wait(1) -- wait 1 second
  end
end
-- start ticking:
go(printer, "tick")
-- start tocking after 0.5 seconds
go(0.5, printer, "tock")
```

This relatively simple interface is a low-level meta-mechanism from which more complex temporal patterns and semantics can be constructed. For example, a coroutine which returns a function will continue execution in that function's body (as a tail call in Lua), and a coroutine which returns a call to its own function will implement temporal recursion [19].

Furthermore, it is possible to create new schedulers whose metrics are driven by events within the script itself; this can be used to create a tempo clock for example.

4.3 Multi-threading and audio

Given the power of coroutines to deterministically model concurrent activities the decision by the Lua authors to shun multi-threading is easier to understand[11]. Our own approach is to maintain this single-threaded nature for the Lua interpreter instances: it is consistent with the recommended manner to interact with OpenGL contexts and GPU resources, and its deterministic assurances greatly simplify the code within *libluaav*.

Unfortunately however, audio processing is better placed in a dedicated independent high-priority thread, in which unbounded calls (such as memory allocations, garbage collections and so on) are avoided [4]. The natural result is two threads: one for the interpreter and graphics, one for the audio processing, and the problem of synchronization between them.

Our solution is to mirror state between the interpreter thread and the audio thread by means of time-stamped synchronization messages along a pair of single-reader/single-writer FIFO (first-in, first-out) message queues (built upon the JACK ringbuffer[12]). Memory allocation/disposal and initialization of audio objects occurs in the main thread, but subsequent state changes triggered from Lua code are serialized and dispatched to audio thread via the message queue. The audio thread can then retrieve these messages (up to the appropriate timestamp) and apply the state changes in the context of signal processing directly.

[9] Coroutines are subroutines that act as master programs (Conway, 1963). A coroutine in Lua is a concurrent asynchronous state with its own instruction pointer, stack and local variables, but with access to shared globals. A coroutine is constructed from a Lua function, which can explicitly yield execution and be resumed later.

[10] Durations are measured in seconds by default, however LuaAV supports the creation of arbitrary user clocks and schedulers, with which concepts of tempo and beats can be easily constructed.

[11] Introducing threading into standard Lua can be done, however the granularity is so high as to make this feature nearly useless and execution effectively single-threaded anyway.

There is necessarily a latency between the Lua thread time and the audio thread time, which is bounded by the update period and jitter of both. So long as actual latency remains below a (user-specifiable) ideal limit, fully sample-accurate temporal determinism can be achieved[12]. If ideal latency cannot be kept, events will fire late, but the order of events remains determinate.

The main drawback of this approach is that audio state cannot be immediately retrieved in the Lua script: method calls on audio objects are asynchronous and cannot return concrete values. Similarly any messages sent from the audio thread to the main thread are also latent, preventing temporally accurate triggering of Lua code in response to audio analysis for example.

4.4 Audio engine

The audio engine within LuaAV acts upon time-stamped messages received on the message queue from the Lua thread, and triggers any process calls in active audio objects (from here on denoted *Synths*[13]). Synths have notions of signal input and output ports of various kinds which can be connected to each other[14]. The connections and disconnections of ports are specified by messages from the main thread.

4.4.1 Dynamic graphs with sample accuracy

For efficiency (and to achieve real-time guarantees) signal processing graph nodes are typically computed over blocks of N sample frames with buffered input and output signal streams. The audio engine has the responsibility to ensure that buffers of data for a Synth's input ports are properly filled and the output ports properly prepared before the Synth's signal processing function is executed.

Since the block-rate is purely an implementation detail and carries no musical or aesthetic significance, we aim to hide it completely from the Lua interface; users should be able to code with state changes at any arbitrary time ('strongly timed'). In order to achieve sample-accuracy in graph dynamics while maintaining determinism in the signals, LuaAV traverses sub-sections of the graph for sample-accurate state changes. For CPU efficiency, only the upstream dependencies of the changing node(s) must be computed, and only up to the sub-block timestamp of the scheduled change.

For memory efficiency, it is better to minimize the number of buffers allocated, and re-use existing memory when it can be safely done. A good strategy would maintain a memory pool of re-usable buffers with a lazy allocation, eager recycling policy, under the control of a coloring algorithm akin to register allocation. However this strategy becomes quite complex with the combination of multiple references (buffers with multiple readers and/or multiple writers), feedback connections and sub-block size traversals. We currently only optimize for single-use non-feedback connections and defer recycling until the end of the block, but are researching more optimal algorithms.

4.5 Signal Processing

Toward efficiency and extensibility, LuaAV's Synths are built according to a low-level C API. The API provides as much functionality as possible (such as automatic bindings to Lua) without compromising flexibility. Synth code can be written in C or C++, does not need to inherit or compose any pre-existing objects, nor conform to a particular data layout.

4.5.1 Standard signal processing units

For the purposes of rapid testing and minimal dependency, a concise set of standard Synths are provided within the *libluaav* library. These units wrap low-level synthesis routines (using code from the Gamma [14] library) within abstract definitions of ports, methods and process routines. Static code-generation in the *libluaav* build process uses these definitions to automatically create bindings to the LuaAV audio API, Lua bindings, and documentation.

The following example code plays a series of sine bleeps of random frequency, whose durations progressively shorten from 1 second to 1 millisecond:

```
local outs = Outs() -- stereo output bus
for i = 1, 1000 do
  local dur = 1/i
  local f = 100 * math.random(10)
```

[12] Clock drift is not an issue per se, since we derive our source of real time from the audio sample clock itself.

[13] The term 'synth' is used rather than 'unit generator' to indicate a coarser granularity in the graph. Finer granularities are better handled by JIT compilation of synths from sub-components which better deserve to be named unit generators.

[14] Feedback between Synths necessarily incurs a block of delay. Feedback within synth implementations incurs a single-sample delay.

```
  local synth = Sine{ freq = f }
  outs:play(synth, dur)
end
```

4.5.2 Embedding CSound

Audio synthesis specification is a complex domain with a long history. We considered it practical to re-use existing interfaces and frameworks if possible. CSound in particular has a long heritage and a huge collection of signal processing primitives ('opcodes').

Embedding CSound within LuaAV was a remarkably straightforward process, thanks to the design of the CSound API [5]. CSound instances can be created in a Lua script as LuaAV *synth* objects using CSound's host-implemented audio option bound to the *libluaav* audio API. CSound synths can thus be connected with other LuaAV synths. Typically CSound instances in LuaAV have only minimal score specification, turning over the responsibility of the generation of score events to the Lua script. For example, the following code snippet plays an ascending harmonic scale on instrument 1 from the orchestra defined in "demo.csd":

```
require "csound"
local cs = csound.create("demo.csd")
play(outs, cs, 4)
for h = 1, 16 do
  cs:scoreevent(
    'i',   -- event type
    1,     -- p1 (instrument)
    now(), -- p2 (start)
    1,     -- p3 (duration)
    1.0,   -- p4 (amplitude)
    100*h  -- p5 (frequency)
  )
  wait(0.25)
end
```

CSound instances can also be created from strings of valid CSound code, opening up interesting possibilities of code-generating CSound orchestras and scores at run-time.

4.5.3 Run-time generation of signal processing code

Our primary focus for audio synthesis in LuaAV however is the generation and compilation of synthesis code from definitions specified at run-time, leveraging the the JIT capabilities of LLVM via *luaclang*. It is our view that runtime code-generation best serves the goals of extensibility and efficiency (and to a certain degree also portability [6]).

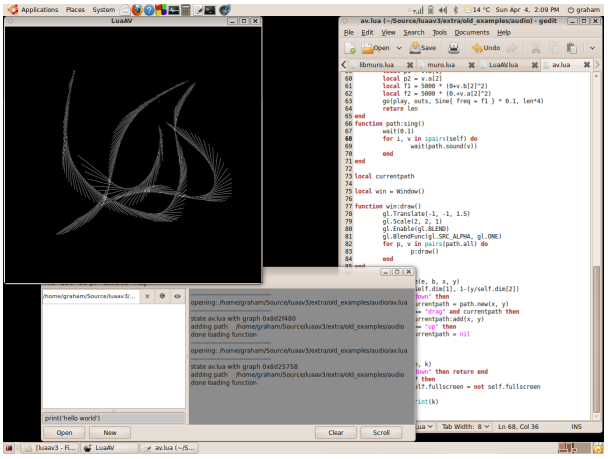


Figure 3: An audio-visual interactive canvas (mouse-paths converted to hyperbolic lines, rendered with OpenGL and sonified as grain chirps). The script itself is being edited in Gedit.

A more detailed description of our investigations can be found in [18]; here we will provide a summary for the reader's convenience. Initial experiments constructing abstract syntax trees (ASTs) of complex expressions from elementary nodes were very promising: expression trees have a natural corollary to data-flow networks typical in signal processing, and also to the static single-assignment (SSA) form of LLVM's intermediate representation (IR).

Expression graphs however are limited because they have no internal state. Extending our model to support stateful objects such as filters and variable oscillators called for the run-time generation of data-structures to maintain state across function calls, and associated routines to allocate and free memory and connect to the LuaAV audio system appropriately. We have successfully implemented such a model, and are continuing to pursue this line of development and hope that a user programming interface will stabilize soon.

The performance of the JIT compiled code is close to that of a native static compiler (GCC). The time to JIT a simple Synth can fit within the acceptable latency window between the Lua thread and the audio thread.

4.6 Beyond Audio

We have concentrated on audio in this paper, but it is important to note that LuaAV has very strong capabilities in the visual (2D and 3D) domain (see Figure 3). A near-complete binding of the OpenGL standard is included as a dy-

namically loadable Lua module, along with the Muro module which uses a generic Matrix data format to connect image, video files/cameras, GPU textures, shaders and slabs, matrix data processing and analysis, 3D mesh drawing, and supporting utilities for vectors, quaternions and other common 3D tasks.

LuaAV has MIDI and OSC built-in, and extension libraries for numerous devices and systems. And of course, anything that can be loaded in standard Lua can also be used in LuaAV, such as existing SQL database bindings, networking code, Cairo 2D drawing, PEG text parsing, and so on.

5 Future directions

It is notable that code generation appears in all three strategies to embed signal processing within LuaAV: static generation of synthesis units within the library, the potential to generate CSound orchestras programmatically at runtime, and to code-generate entire synthesis routines to machine code using LLVM/Clang. We believe it is a natural consequence of increasingly dynamic and extensible programming interfaces and environments, and which will continue to grow.

Embedded scripting language bindings to efficient library code still carry a divide between static and dynamic code which remains immutable during runtime. The incorporation of runtime JIT compilation (such as LLVM/Clang in LuaAV) adds the capacity to dynamically generate new bindings into the environment and the augmentation of existing bindings and binaries on the fly.

For example, computer vision video filters in LuaAV are code generated by bringing together functionality from OpenCV and the Muro Matrix specification to generate a new video filter with full Lua bindings which is properly adaptive to the heterogeneous nature of computer vision data. By compiling these filters at runtime, it's possible to dynamically alter how filters mix and combine with further processing elements in ways that would otherwise be preconceived and fixed.

As we have explored this area of software design, it has become apparent that the trend is toward heterogeneous computational environments that freely mix paradigms be they typed versus untyped, dynamically compiled versus statically compiled, and so on. What we are working to achieve is a continuum of paradigms as opposed to simply concatenating them together. Currently in LuaAV it is possible to mix Lua code and C code. As we develop the system further and add intermediate languages to generate new code between the paradigms, the boundary will only become more blurred.

6 Acknowledgements

With thanks for the support of the AlloSphere Research Group, University of California Santa Barbara.

References

[1] X. Amatriain, J. Kuchera-Morin, T. Hollerer, and S. T. Pope, "The allosphere: Immersive multimedia for scientific discovery and artistic exploration," *IEEE MultiMedia*, vol. 16, pp. 64–75, 2009.

[2] A. Brown and A. Sorensen, "Dynamic media arts programming in impromptu," *Proceedings of the 6th ACM SIGCHI conference on Creativity & ...*, Jan 2007.

[3] N. Collins, A. Mclean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organized Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[4] R. B. Dannenberg and R. Bencina, "Design patterns for real-time computer music systems," ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music, 2005.

[5] J. Ffitch, "On the design of csound5," in *Proceedings of the 3rd Linux Audio Developers Conference*, ZKM, Karlsruhe, Germany, 2004.

[6] M. S. O. Franz, "Code–generation on–the–fly: A key to portable software," 1994.

[7] D. Griffiths, "Fluxus," http://www.pawfal.org/Software/fluxus/, 2007.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua — an extensible extension language," *Software Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[9] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, 2004.

[10] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[11] M. Pall, "LuaJIT," http://luajit.org/, 2007.

[12] Paul Davis, "Jack — connecting a world of audio," http://www.jackaudio.org/, 2010.

[13] M. Peljhan, "Common data processing and display unit-tokyo system prototype," http://www.ntticc.or.jp/ Exhibition/2009/Openspace2009/Works/ commondataprocessinganddisplayunit.html, 2009.

[14] L. Putnam, "Gamma - generic synthesis c++ library," http://mat.ucsb.edu/gamma/, 2009.

[15] W. Smith, "Abelian: A visual and spatial platform for computational audiovisual performance," Master's thesis, University of California Santa Barbara, 2007.

[16] W. Smith and G. Wakefield, "Synecdoche," http://www.mat.ucsb.edu/ whsmith/Synecdoche/, 2007.

[17] ——, "Computational audiovisual composition using lua," *Communications in Computer and Information Science*, vol. 7, pp. 213–228, 2008.

[18] ——, "Augmenting computer music with just-in-time compilation," *Proceedings of the International Computer Music Conference*, 2009.

[19] A. Sorensen and A. Brown, "Aa-cell in practice: an approach to musical live coding," in *Proceedings of the 2007 International Computer Music Conference*, 2007.

[20] G. Wakefield, "Vessel: A platform for computer music composition, interleaving sample-accurate synthesis and control," Master's thesis, University of California Santa Barbara, 2007.

[21] G. Wakefield and H. Ji, "Artificial nature: Immersive world making," in *EvoWorkshops*, 2009, pp. 597–602.

[22] G. Wakefield and W. Smith, "Using lua for audiovisual composition," in *Proceedings of the 2007 International Computer Music Conference*. International Computer Music Association, 2007.

[23] G. Wang and P. Cook, "Chuck: A programming language for on-the-fy, real-time audio synthesis and multimedia," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*. New York, NY, USA: ACM, 2004, pp. 812–815.

The WFS system at La Casa del Suono, Parma

Fons Adriaensen
Casa della Musica
Pzle. San Francesco
43000 Parma (PR),
Italy,
fons@kokkinizita.net

Abstract

At the start of 2009 a 189-channel Wave Field Synthesis system was installed at the Casa del Suono in Parma, Italy. All audio and control processing required to run the system is performed by four Linux machines. The software used is a mix of standard Linux audio applications and some new ones developed specially for this installation. This paper discusses the main technical issues involved, including sections on the audio hardware, the digital signal processing algorithms, and the software used to control and manage the system.

Keywords

Wave field synthesis, spatial audio, distributed audio systems

1 Introduction

La Casa del Suono in Parma, Italy, is one of the cultural entities managed by the city of Parma, located in a small restored church and open to the general public. It is in the first place a museum dedicated to the history of audio technology, showing a collection of vintage audio equipment. It also aims to provide to its visitors a view on current developments in this area.

One of the installations designed for that purpose is the Wave Field Synthesis system installed in the *Sala Bianca*, shown in fig. 1. This is a room of around 7.5 by 4.5m meters and 4.5m high. When the doors (seen in the back) are closed, there is a continuous ring of 189 small speakers running along the complete inner circumference of the room, one every 12cm (with some small exceptions due to constructional constraints). The speakers are hidden behind a white tissue covering all the walls, and are constructed in 'blocks' of 10, 15, or 17 units. They are a bass-reflex design consisting of a small four inch bass and midrange unit (from Ciare) and a tweeter, driven by a passive crossover network. The height of the ring is a compromise between typical ear height for a seated and a standing audience. The usable frequency range is 50Hz to 20kHz, and sound quality is remarkably good for a design of this size. Except for the space taken by the speakers the walls are completely covered by sound absorbing material, leading to reasonably good 'dead' acoustics.

Figure 1: La Sala Bianca

As a project the Sala Bianca is the result of a collaboration between the city of Parma and the Engineering Department of the University of Parma. The university in turn contracted the author to specify the audio hardware and design the complete software.

The system is intended to be used both as a public demonstration of WFS technology, and as a scientific research instrument. This has re-

sulted in some quite peculiar and contradictory requirements. In the first role the system is operated by the museum staff, and it has to run fully automatically every day and without any technical support. As a scientific instrument on the other hand it has to be reconfigurable without any artificial limits, allowing the researchers to e.g. easily replace part of the software with experimental versions, or use it as a listening room for psycho-acoustic experiments using entirely different rendering algorithms. After the system was constructed a third way of using it was added as the direction of La Casa della Musica (who are managing the installation) also expressed their desire to use the Sala Bianca as an instrument for electro-acoustic music.

This is of course not the first large-scale WFS installation using Linux. A much larger one was constructed in 2007 at the Technical University of Berlin, see [Baalman et al., 2007]. The two systems are however quite different both in the hardware and software solutions that were adopted.

2 A short introduction to WFS

Wave Field Synthesis operates by reconstructing the wavefront that would be generated by a real sound source, called the *primary source,* by a large number of real *secondary sources*. The method used is based on the Huygens principle, and formally expressed by the Kirchoff integral [Verheyen, 1998]. It only works up to a frequency determined by the distance between the secondary sources, which should be less than half the wavelenght. Above this frequency, spatial aliasing will lead to false images of the primary source being generated as well as the correct one. For this reason most WFS systems use linear arrays of closely spaced secondary sources, as filling a plane would increase the required number above practical limits. Reducing a WFS system to 2-D operation has some consequences: it complicates the signal processing, and it also leads to an approximate solution that however still works very well in practice. Marije Baalman's recently published book [Baalman, 2008] provides a good overview of the state of WFS technology today, including some features not covered by the system discussed in this paper.

A WFS system can generate virtual sound sources either behind or in front of the secondary sources. The latter, in a complete surround setup as the Sala Bianca or the Berlin system, means virtual sources that appear to be *inside* the room.

Each virtual source is in principle reproduced by all speakers (in practice about half of them). For each pair of primary and secondary sources different gain factors, delays and filtering are required. So the complexity of the DSP software is proportional to $N_{primary\_sources} \times N_{secondary\_sources}$

The system installed in the Sala Bianca is designed to create up to 48 moving virtual sources in real time. Movement of the primary sources is continuous - it is not implemented by crossfading between fixed points but by adjusting the synthesis parameters at the full sample rate. This means that also the Doppler effect of a moving source is reproduced faithfully.

3 The audio system

Figure 2 shows the structure of the audio system. There are four PCs, all of them from the Siemens/Fujitsu Celsius range of workstations. This choice was influenced largely by the IT department of the city of Parma which imposed its requirements regarding suppliers, warranty conditions etc., but it turned out to be a very good one. At the time of writing all these machines are running Fedora 8 (installed two years ago), but they will all be converted to ArchLinux in the coming weeks. This distro makes it easy to use a much leaner system (e.g. without a fat desktop such as Gnome or KDE).

All machines are fitted with an RME HDSP-MADI interface providing 64 channels of digital audio input and output. RME ADI648 units are used to convert between MADI and ADAT, the latter being the format required by all AD/DA units used in the system. All machines are on a local gigabit LAN, and for audio they are connected to each other using an RME MADI matrix. The matrix has actually 8 inputs and 8 outputs, the remaining ones being used to connect to the 'Lampadario Acustico', another audio system installed at the CdS, and to provide connections for external PCs of researchers and musicians. All units share the same sample clock, transmitted either by the MADI links or by explicit word clock connections.

The **wfsmaster** machine coordinates the operation of the the WFS system, and provides all the access points to external applications using it. It is also the only machine having a human interface.

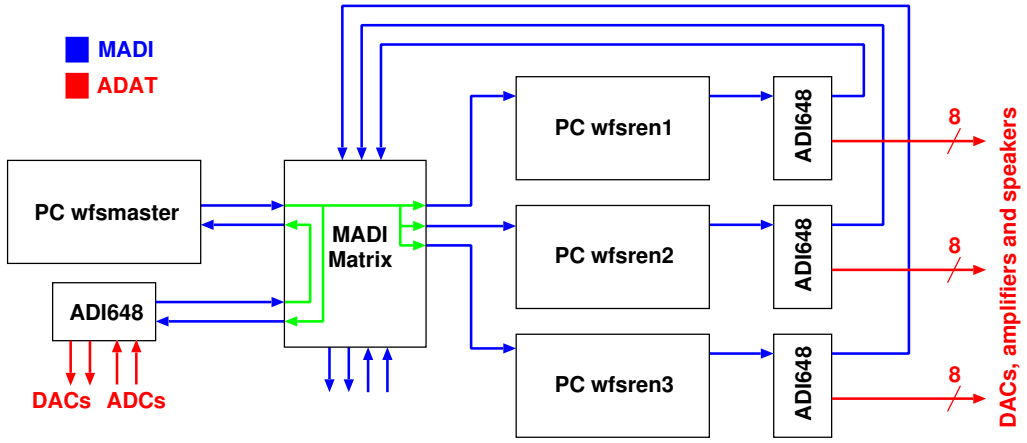


Figure 2: The audio system

The three **wfsrender** machines each take care of 64, 61, and 64 speaker outputs respectively, for a total of 189. They run 'headless' in Unix runlevel 3, and are fully remotely controlled.

All the equipment shown in fig 2, except the ADI648 units of the rendering machines is located in a small control room adjacent to the Sala Bianca. The three ADI648 connected to the rendering machines and all DA convertors and amplifiers are located two large racks in a separate technical room. The entire system is powered by a large UPS.

The normal MADI connections are as shown in fig 2: the 64 outputs of the the master machine are connected to the inputs of all three rendering machines. The hardware ensures that audio is synchronised system-wide - if the rendering machines all use the same Jack period size and would just copy one input to all outputs then an audio signal provided by the master machine would appear on all 192 outputs at exactly the same time. But even if the Jack period sizes are the same, the period boundaries on each machine are of course not synchronised, and this has to be taken into account in the processing code. This is discussed in section 4.1.

The MADI connections from the rendering machines back to the master can be used to monitor and measure the signals generated by the rendering machines. They have proven to be very useful for verifying the correct operation of the signal processing algorithms running on the rendering machines, but are not used during normal operation.

3.1 Using network connections for audio

At the time the system was designed (end of 2007 and early 2008), the use of a dedicated gigabit network to transfer audio between the master and rendering machines was considered, but in the end this solution was rejected for several reasons.

- It would require some solution to ensure that audio signals would remain synchronised. This was actually a minor problem. By providing audio feedback paths for the synchronisation signal described in section 4.1 from the renderers back to the master the delays could be easily measured and automatically adjusted to be equal.
- While the capacity of a gigabit ethernet would be sufficient for the amount of data to be transferred, there were some serious doubts regarding the performance of the network adaptor drivers in a real-time system.
- Using network connections for the audio links would not permit significant savings in the audio hardware. The MADI cards and ADI648 units for the renderers would still be required, as would be those for the master machine (which has to provide a multichannel audio access point). So the only item saved would be the MADI matrix, and that is actually one of the less expensive units in the system.

In the best case the use of network connections for audio signals would have increased the latency of the system by an unpredictable

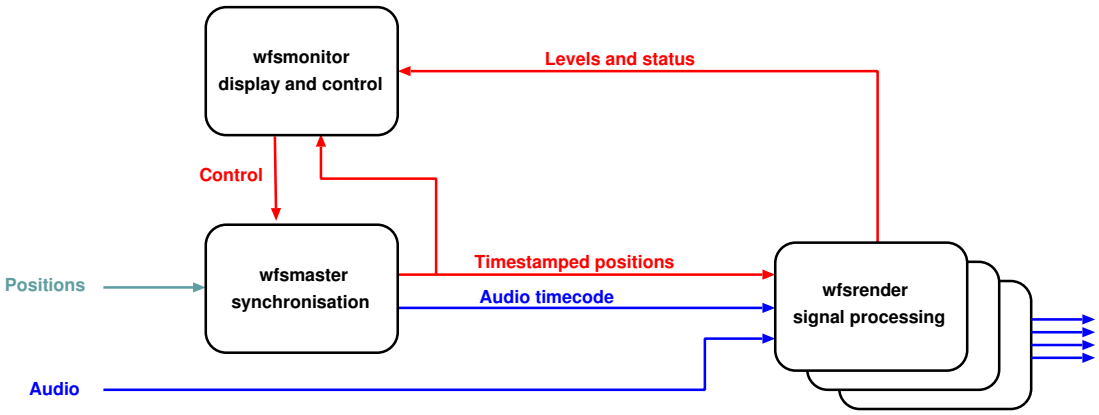


Figure 3: WFS processing structure

amount. The risk of doing this was deemed to be too high at the time. Today the picture could look different, but at least the third point mentioned above remains valid - there would not be any significant savings.

4 The WFS processing architecture

Figure 3 shows the basic WFS processing system which consists of 3 applications which communicate with each other using network messages. All of these command and monitoring messages use the OSC format.

The main function of the **wfsmaster** is to provide a single access point for controlling the WFS rendering system. Apart from generating the synchronisation signal (discussed in section 4.1), this program does not perform any audio processing. It receives commands from external programs specifying the position and movements of each virtual source (more on this in section 5), and transforms these in the format required by the rendering engines.

One instance of **wfsrender** runs on each of the rendering machines. It performs all the signal processing required for the set of speakers controlled by its host. Its inputs are the audio signals - one for each virtual source - from the MADI interface, and the processing parameters transmitted by the wfsmaster. The details of the DSP algorithms used are discussed in section 6. It also transmits monitoring messages containing e.g. the levels of all its output signals along with its internal state and any errors.

The **wfsmonitor** is the only program having a graphical user interface. It shows the current position of all virtual sources, the levels of all 189 speaker outputs, and error and status info from the two others. It can also be used to 'solo' a single speaker, or to send signals directly to a speaker for testing and alignment. There can be any number of instances of *wfsmonitor* - it can be run e.g. on portable PC inside the Sala Bianca, but the system will also run perfectly without it.

4.1 Synchronisation issues

As mentioned already in section 3 the basic audio architecture ensures that all outputs of the rendering machines will be exactly synchronised. If the system would provide only *static* virtual sources that would be all that is required. But to support *moving* sources also the application of the position parameters must be synchronised to sample accuracy on all outputs.

With its standard configuration *wfsmaster* will update and transmit all the source positions every 1024 frames. This is synchronised to (but independent of) the Jack period on the master machine. To allow the renderers to apply this information to the correct audio samples a separate synchronisation signal is generated by the master, and transmitted to the renderers following the same path as the source audio signals. It is similar to the one used by *jack_delay* to measure latency, and consists of a mix of six sine tones with frequencies that are chosen such that their phases encode the current position in a sequence that repeats exactly every 2^{20} samples (around 22s). So every sample has an implicit timestamp in the range $0 \ldots 2^{20} - 1$ that can be recovered easily by any application receiving the signal.

The timestamp corresponding to the first frame of the current position update period is included in the messages transmitted by the master program. A fixed offset (currently 800

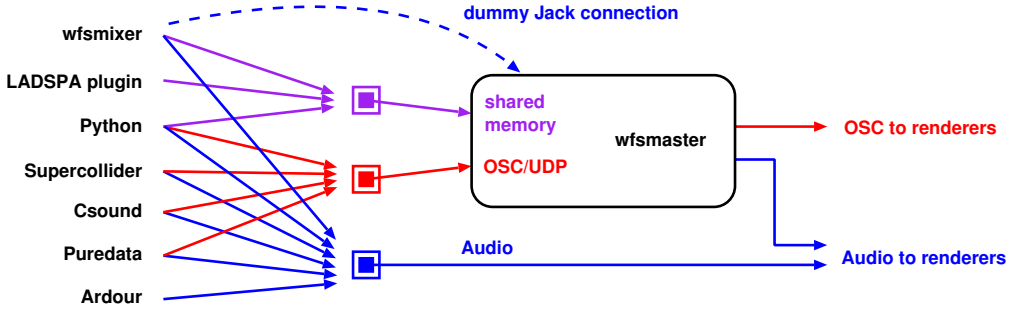


Figure 4: External application interfaces

frames) is added to allow for the worst case expected network latency. The rendering applications use the timestamp and the decoded synchronisation signal to re-align audio and data. They also check the arrival time and integrity of the position messages and the presence of a valid audio time code, and report these in their monitoring messages. An occasional 'late' position message is reported but ignored, as is a short interruption of the time code. Consistent errors will make the renderers mute their output until the situation returns to normal.

4.2 The structure of wfsrender

The *wfsrender* application is the most complicated one in the system. It has two basic functions: compute the internal WFS processing parameters in function of source position and movement, and perform the actual DSP work.

In order to make this application more easily adaptable it uses two plugins to perform most of this work. Both are loaded at runtime (as specified by a global configuration file), and can be replaced without recompiling the whole application.

The **layout** plugin defines the complete geometry of the installation. It provides methods to e.g. find out the exact position of each speaker, to which channel on which host it is connected, etc. It also performs some specific calculations, such as determining if a given x, y coordinate is internal or external and finding its distance to the line of speakers.

The **engine** plugin performs the computation of all required internal parameters and the actual DSP work, i.e. it defines the WFS algorithm used.

The interfaces to these plugins are defined as abstract C++ classes. The 'host' program takes care of the audio and network interfaces, OSC encoding and decoding, the synchronisation signal, status reporting etc. In principle any C++ programmer who understands the required algorithms could create the plugins without being distracted by the system level aspects.

Another feature added recently is multithreaded audio processing in order to use SMP systems more efficiently (the machines used are dual-core). This divides the DSP work over a configurable number of threads, each of them handling a subset of the output channels. This multithreading is transparent to the plugins.

5 External application interfaces

The system described so far just implements the basic WFS algorithm for up to 48 moving sources. To create content, spatialisation, room simulation etc. it depends on external applications supplying both the audio and source position and/or movement information. Figure 4 shows some possibilities.

Audio signals can originate or be processed on the master machine which has almost no CPU load due the WFS system itself. External sources can be connected via MADI, ADAT, or analog inputs. In all cases the signals to be used as WFS sources are just sent to the first 48 outputs of the MADI interface. Time code and test signals typically use channels 49-56, while the last eight channels are normally used for control room monitoring or for providing external analog signals (e.g. for driving subwoofers).

The *wfsmaster* program has two interfaces for source position data.

The **shared memory** interface can be used locally only, but permits sample-accurate control if the position data is generated by a Jack client. A dummy Jack connection to the master program will ensure correct order of execution in that case. This interface is also used by

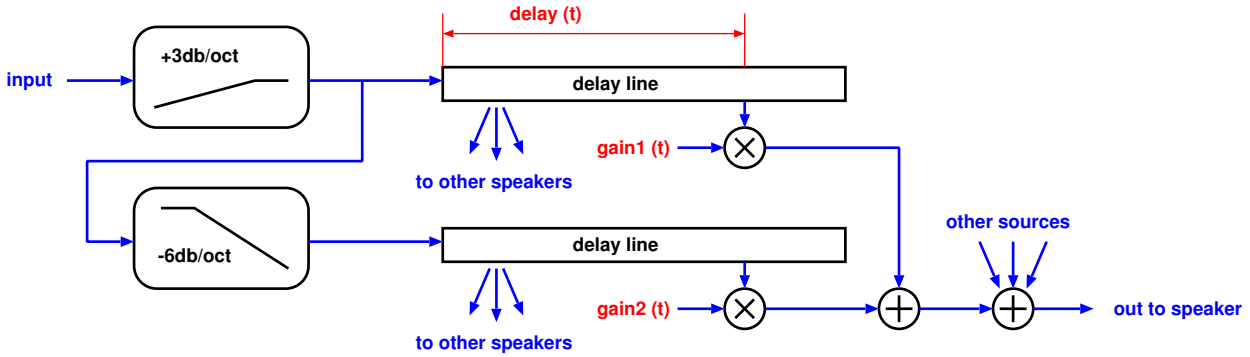


Figure 5: The basic DSP operation

some LADSPA plugins that allow source positions and movements to be encoded as automation data in Ardour. The *wfsmixer* application provides a graphical tool to control source positions. It also allows basic room simulation using Ambisonic convolution reverbs.

The **OSC/UDP** interface can be used from all the standard synthesis programs, from Python (scripts or interactive), or other languages and applications, and from any PC on the local network. The position commands allow a source to move at a controlled speed, with the master program interpolating the movement if it takes longer than one update period.

Both interfaces also include a per-source gain factor which is applied at the end of the DSP processing. This is necessary to still allow full level for a source positioned at a large distance, for example when simulating virtual speaker sets for Ambisonics.

6 The DSP algorithms

The processing required for WFS is not really very complex, but it has to be repeated $N_{in} \times N_{out}$ times, or around half that number for a 'surround' layout such as the Sala Bianca.

6.1 The basic real-time processing

The form used in the *wfsrender* application is based on equation (29) in Sascha Spor's very practical summary of WFS theory [Spors et al., 2008], which shows the driving function for a spherical wave reproduced by a linear array. This translates in to the diagram shown in fig. 5. For each single source, and for each speaker reproducing that source, the two filtered signals are delayed by a time *delay* and then combined using gain factors *gain*1 and *gain*2. Using two delay lines allows the second filter to be shared for all outputs, as are the delay lines themselves. The three parameters are computed from the source position and the system geometry. The *wfsrender* code performs this computation at every position update (i.e. every 1024 frames) and interpolates them linearly in between. Some optimisations are applied in the actual code for cases where one or more of the three parameters remain fixed.

Since the system allows for moving sources and uses the the actual delays corresponding to the postion of a primary source (which means it will reproduce the Doppler effect exactly), it can't handle the case of plane waves, as these would correspond to infinite distance and delay. It would be easy to add these as a special case, but so far there has been no need to do this.

6.2 Handling sources near the speakers

The equation cited above is an approximation that is valid only for primary sources that are not too close to the line of speakers. This is because its derivation assumes a continuous distribution of the secondary sources. Since the system supports arbitraty movements and virtual sources can cross the line of speakers without restriction these cases have to be handled separately.

The exact solution becomes quite complex, and it can't be derived as a limit case of the standard equation. To find it one has to go back to the full three-dimensional case, apply the secondary source quantisation there, and then reduce the result to a linear array.

The *wfsrender* code uses a pragmatic approach to handle this situation. If a primary source comes too close to the secondary ones, two parameter sets are computed: one for a source at a fixed distance (for which the normal equation is still valid) behind the speakers,

and one for the same distance in front. The two parameter sets are then combined according to the actual source position. The result is an approximation of the exact solution, but works well in practice.

7 System organisation and use

All source code, scripts, data files etc. required to install the system are kept on an NFS share available on all machines. A single click on a desktop menu will install or update all necessary components system-wide. Compilation is always on the target machines, i.e. everything gets installed from source.

Binaries, plugins and most scripts are kept in the standard places under /usr/local. The only exceptions would be experimental versions that would be installed per-user.

The basic setup of *wfsmaster, wfsrender* and *wfsmonitor* is defined in a global configuration file. 'Technical' users would normally just run *wfsmonitor* which then permits to run and control the others without having to use remote logins, set up Jack servers, etc. They would then add anything else they require on the master machine manually. Alternatively the Python components described below provide an easy way to create sessions that need to be recreated automatically many times, and have in fact become the preferred way to use the system.

7.1 Automatic configuration

For the 'museum' user the situation is quite different. A single click on a desktop icon launches the complete system under the control of a Python program that configures everything and that acts as a server to a remote control application. The remote control looks like a typical desktop audio player, with stop/start/loop buttons, volume control, a playlist etc. It runs locally on the master machine and also on one or more 'EEE' notebooks with wireless network access, used by the museum staff. Selecting an item from the playlist reconfigures the complete system according to the requirements for that item.

Apart from the components that implement the remote control server and the playlist, the Python program has classes for all required audio applications, each new instance of them becoming a separate Jack client. These include *py_jackctl, py_jplayer, py_ambdec, py_jconvolver* and some others. More will be added in the future.

8 Acknowledgements

I'd like to thank Prof. Angelo. Farina (University of Parma) and the direction of La Casa della Musica for having provided the opportunity to create the system described in this paper. A warm thanks also to Stefano Cantadori and to all my former colleagues at Audio Link and AIDA.

The realisation of this project would not have been possible without the work of all the developers who have created Linux and its audio system. An uncountable number of people on the Linux Audio mailing lists have provided valuable and often essential help and hints. My sincere thanks go to all of them.

References

Marije Baalman, Torben Hohn, Simon Schampijer, and Thilo Koch. 2007. Renewed architecture of the swonder software for wave field synthesis on large scale systems. In *Proceedings of the 6th Linux Audio Convention*, Berlin, Germany.

Marije Baalman. 2008. *On Wave Field Synthesis and Electro-acoustic Music.* VDM Verlag Dr. Mueller AG, Saarbruecken, Germany.

Sacha Spors, Rudolf Rabenstein, and Jens Ahrens. 2008. The theory of wave field synthesis revisited. In *Proceedings of the 124th AES Convention*, Amsterdam, The Netherlands. Audio Engineering Society.

Edwin Verheyen. 1998. *Sound Reproduction by Wave Field Synthesis.* Technische Universiteit Delft, Delft, The Netherlands. Doctoral thesis.

General-purpose Ambisonic playback systems for electroacoustic concerts - a practical approach

LAC 2010, Utrecht

Jörn NETTINGSMEIER
Freelance audio engineer and qualified event technician
Lortzingstr. 11
Essen, Germany, 45128
nettings@stackingdwarves.net

Abstract

Concerts of electro-acoustic music regularly feature works written for various speaker layouts. Except in the most luxurious of circumstances, this implies compromises in placement, frequent interruptions of the concert experience to relocate speakers, and/or error-prone equipment rewiring or reconfiguration during the concert.

To overcome this, an Ambisonic higher-order playback system can be used to create virtual speakers at any position as mandated by the compositions. As a bonus, the performer can then be given realtime control of the source positions in addition to their levels, increasing the creative freedom of live sound diffusion.

Deployments at LAC 2009 and the 2009 DEGEM concert at musikFabrik in Cologne have yielded very good results and were met with general approval. Additionally, two informal listening tests with a group of film sound artists and electro-acoustic composers were conducted to gather expert opinions on advantages and shortcomings of the proposed virtual speaker system.

This paper was originally published at the Ambisonics Symposium 2010, hosted by IRCAM in Paris, France.

Keywords

Ambisonics, surround sound, live sound reeinforcement, electro-acoustic music, tape music

Introduction

This paper describes a practical Ambisonic concert system, as well as a setup procedure suitable for live situations. The system consists of a Linux-based rendering machine, and otherwise standard sound reinforcement equipment that should be available at electro-acoustic music facilities or is readily obtained from rental companies.

All software parts of the production toolchain are free and open-source. Hence, they do not incur licensing costs, are reasonably easy to port to and interface with existing Mac OS X, Solaris and (with certain limitations) Microsoft Windows environments [Sle2010], and are easily customized to accommodate special requirements that frequently arise in any experimental arts context.

The sound engineer will appreciate the ease of creating different speaker layouts or trying small layout variations to better convey the composer's intentions in a given venue, and the ability to compensate for non-optimal speaker positioning due to space constraints, escape routes and other venue safety requirements.

Concert curators will like the added flexibility in accommodating non-standard spatial configurations (whose variety grows even more quickly when height reproduction is desired) and the option of commissioning or accepting native Ambisonic compositions without extra cost or error-prone pre-rendering attempts by composers. With changeover times and distracting stage work minimized, it becomes much easier to create a coherent and focused concert experience.

Overview

A minimal Ambisonic live playback rig consists of a multichannel playback source with appropriate encoders, a decoder, digital-to-analogue converters and a number of speakers and amplification.

For pre-produced tape works, the source, the encoders (i.e. the panners) and the decoder will usually run on a single PC. If the work features live electronics or sound is rendered in real time on a machine provided by the artist, it will be necessary to accommodate external audio sources. By providing analog line-ins as well as multichannel digital inputs, all situations should be covered.

While the decoding machine can drive the converters and amps directly, it is highly desirable to have a physical master volume control outside the computer: for convenience, and as an emergency breaker in case something goes horribly wrong (as happens with complex digital systems from time to time).

A digital mixer with digital I/O is the obvious choice. It also handles analog signals from an announcer's microphone or traditional instruments, if necessary.

For flexibility, the mixer's inputs should be routable through the encoding machine and back into the mixer, so that external signals fed into the mixer can benefit from Ambisonic panning as well. In such a setup, it is important to consider (and minimize) the system latency[1].

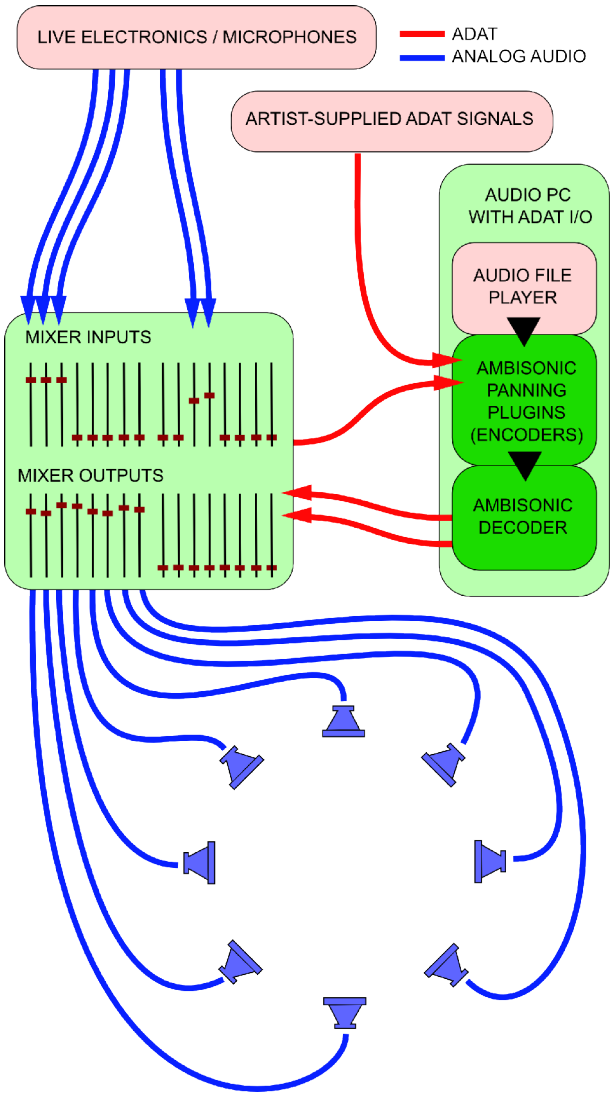


Illustration 1: Signal flow overview of an Ambisonic concert playback system, using a digital mixer. Mixer output faders used for speaker gain compensation.

1 External signal transmission

Currently, there are two suitable multichannel digital interface standards that are well-supported by free software.

ADAT lightpipe connections provide eight channels of 24 bit audio at 48 kHz over a cheap optical link using polymer fibre with plastic toslink connectors. Their maximum transmission distance is specified as 10 metres; under optimum conditions, 20 m are possible. ADAT has become a commodity standard that is available on many consumer and professional computer interfaces, many of which have production-quality Linux drivers. Most live mixing desks support ADAT natively or through optional interface cards. The channel count is usually limited by the number of available extension bays. 32 inputs and outputs (i.e. four ADAT pairs) are a common upper bound, which is suitable for Ambisonics up to fourth order plus room for subwoofers and auxiliary feeds.

MADI (Multichannel Audio Digital Interface) connections carry 56 channels of 24bit audio at 48kHz (or 64 if the optional varispeed capability is not needed) over either coaxial 75 ohm cables terminating in BNC connectors (which are cheap and reliable), or optical fibre using ST plugs (slightly more expensive, but an ace in the hole over long distances where hum currents might be

[1]The ffado.org website provides a good starting point for latency tuning. The focus is on firewire devices, but most of the tricks are applicable to all audio systems: http://subversion.ffado.org/wiki/LatencyTuning

an issue). Default MADI sockets are rare on mixing desks, but most vendors offer optional expansion cards.

On the computer side, as of this writing, the choice is currently limited to the RME MADIface (available in PCIe and ExpressCard flavours), for which complete and reliable Linux support is available.

ADAT-to-MADI bridges and vice versa are available from several vendors for more complex setups.

Both ADAT and MADI can accommodate double or quad sample rates by demultiplexing the signals onto two or four physical channels, at the cost of reducing the number of transmission channels accordingly. It should be noted that the only tangible benefit of higher sample rates in a live situation is a slightly reduced latency (since most devices have a constant number of samples of buffering, regardless of the rate), but at a significant cost in bandwidth, CPU and storage.

2 Software components and internal signal flow

The decoding PC should be equipped with at least two ADAT I/O connectors, i.e. 16 channels in each direction. Tape pieces can be played back from the PC itself, and live electronics, microphones or instrument signals will be coming in via the ADAT inputs.

Inside the PC, the audio signals are handled by the **JACK Audio Connection Kit** [Dav10], a low-latency sound server that allows the user to route audio between different applications and have them co-operate in real time.

In the use case at hand, these are:

- a *playback engine* for tape pieces,
- a *virtual mixer/signal router* for manipulation and to host the panning plugins, and
- an *Ambisonic decoder* to generate the speaker feeds.

This author prefers **Ardour** [Dav10-2] (a very flexible, free digital audio workstation) as the signal handling hub inside the PC, since it can double as a playback engine for complex multichannel pieces and offers parameter automation. If only routing and panning are desired, a more lightweight mixer application such as the Non-Mixer [Lil10] could be considered.

All incoming signals are patched into Ardour buses (or tracks, if recording capability is necessary), tape pieces are imported as single-channel audio regions into mono Ardour tracks, and a master bus suitable for the desired target format is created. In the case of 3rd-order periphonic playback, it is 16 channels wide.

Note that Ardour itself is agnostic to surround formats – it is only the panner that cares about (and defines) the meaning of each channel in the master bus.

The default panners in all tracks and buses must be bypassed. Instead, an Ambisonic panning plugin is inserted as the last post-fader plugin of the channel strip. Panners up to third order are freely available as part of the **AMB plugin set** [Adr10]. An extension to higher orders is trivial.

For tape pieces, the panners can be set in advance to produce the required virtual speaker positions, either statically or with automation, if in-flight adjustments are required.

If necessary, the performer can be given real time control of rotation, azimuths and elevations via MIDI controllers. The mapping of parameters in Ardour is as simple as a shift+control middle-click on a fader widget to initiate MIDI learn mode, followed by a quick wiggle of the desired hardware controller, which is now bound to the widget.

Optionally, an Ambisonic rotator can be inserted into the master bus to correct for angular offsets of the rig (if, for example, a two-in-front octagonal preset is being used for an octagon with a centre speaker), or as an additional control parameter for creative live sound diffusion.

From Ardour's master bus, the signal is then fed to an appropriate JACK-aware decoding software. The author recommends **AmbDec** [Adr10-2], a state-of-the-art decoder capable of up to 3rd-order horizontal and first-order vertical playback. It supports dual-band decoding and optional but highly recommended near-field compensation.

After decoding, the signals can be patched to the ADAT outs, or optionally run through a convolver for additional FIR filtering, and then back into the physical mixing desk, which in turn feeds the speakers.

Equipment requirements and setup

As noted above, a silent PC with sufficient ADAT I/O is essential. Multi-core CPUs are convenient to ensure a responsive user interface when audio computation load is high, but they are not mandatory. Anything like a 1.0 GHz Pentium or better should be more than adequate for simple playback and decoding.

Budget permitting, a digital mixer is highly recommendable. It should have as many analogue outs as there are speakers to drive.

Next, decide on the make and number of speakers. In Ambisonic systems, all speakers will contribute to any one virtual source to some extent, giving you a slightly higher SPL efficiency over the entire listening area. On the other hand, electronic music does require rather more headroom than your average mastered pop material. For a small audience (50-80 people), 8 or more large active near-field monitors might be adequate in a medium-sized room if supported by subwoofers, but for anything larger, P.A.-grade material should be used to avoid sonic compromises and burnt drivers. All speakers and amplification should be identical, to ensure consistent phase response.

Six speakers will give you second-order horizontal coverage with a usable listening area of perhaps one-third the radius of the speaker circle. Third-order operation is recommended for an extended listening area of one-half to two thirds radius, with improved source sharpness that is closer to (but not quite on par with) discrete speaker panning. The minimum number of speakers for 3rd-order horizontal is eight.

1 Selection of a suitable base layout

Assuming that the reader does not have the means of computing her/his own decoding coefficients (much like this author), it is highly advisable to select a speaker layout that can be derived from one of the presets shipped with AmbDec. As of this writing, higher-order capable presets include a number of regular polygons, and regular polyhedra as well as stacked rings for full 3D reproduction[2].

Generally, the angles of incidence should be kept as dictated by the preset. The speaker distances can then be varied within reasonable limits to accomodate the venue, as they only require delay and gain compensation without affecting the decoding coefficients as such.

Extreme distance differences should be avoided, because the maximum sound level will be bounded by what the farthest speaker can safely deliver at the sweet spot. Moreover, distant speakers will have more room reverb at the listening position, which will reduce the sharpness of sources in that direction, create uneven tone color across the rig and might cause closer speakers to dominate the directional perception.

In practice, the usual starting point will be a circle, which can be elongated to an ellipse or even made into a rectangle if necessary.

2 Speaker positioning

The first step is to mark a point of reference on the floor in the centre of the listening area. This will be the "sweet spot" to which the setup is being calibrated.

Now set up the speakers at the correct angles (minor deviations are acceptable). A laser angle gauge will speed up this process considerably - adequate specimens can sometimes be obtained cheaply at your local do-it-yourself store.

3 Distance measurement

Mount a laser range finder at ear height on a tripod over the point of reference. Then measure the distance to each of your speakers precisely (to within 1-3 centimetres). Reproduction in the sweet spot breaks down as the error approaches $\lambda/2$ (around 6 kHz at 3 cm). Outside the strict sweet spot and in a diffuse field, the distinction is largely academic, but it is suggested that this precision be maintained to get predictable results.

Aim at the tweeter or some other easily identifiable flat surface, but avoid shooting at the front cloth or mesh grilles, as these may introduce errors depending on whether your beam lands on the surface or goes through the material. If there is a bass port behind your measuring spot, the error may be substantial. Similarly, avoid cheap range

[2]For special setups such as hemispheres, the author has found that people who are researching decoders are generally very interested in getting field testing and user feedback. Which means you might get a custom-tailored matrix from a friendly Ambisonics research facility near you if you ask nicely. The AmbDec author has also made a standing offer to try his skills on any interesting speaker layout you might care to throw at him.

finders that do the actual measurement with ultrasound, since you can never be sure what you are measuring. To mislead users, those cheap units usually feature a laser as well, but it is only used for aiming, and the actual measurement beam is much wider and more diffuse.

4 Distance compensation and level adjustment

Now edit the AmbDec preset file you have based your speaker setup on. Leave the angle values alone (they are only there for clarity - changing them will not automagically adjust the coefficients), but do enter your measurement results into the speaker distance column. These values govern the distance correction functions, which you must activate in the setup dialog: be sure to tick both "delay compensation" and "near field compensation". You can also activate gain compensation based on distance, but that assumes that all your speakers are precisely level-matched to begin with. As this is usually not the case, leave that option off and proceed as follows:

On your point of reference, set up a good SPL metre (pointing straight upwards, to avoid measurement errors due to shadowing effects, at least in the horizontal plane) and calibrate all speakers to within +/- 0.1 dB using pink noise. The calibration gains can be applied in the external mixer or, if available, in the software mixer of your audio interface. If neither of these options is available, use an additional Ardour bus for each speaker send.

As an alternative calibration method, use an omni-directional measurement microphone. A cheap one, like the Behringer ECM 8000, will do fine. Again, point it straight upward or downward. Install the JAPA realtime analyzer [Adr10-3] on your rendering machine and feed the microphone signal into it. Conveniently, JAPA provides a pink noise test signal, which you can now route to each speaker in turn, using AmbDec's external test signal input. Use the most distant speaker as your reference, bring it up to the desired level, and make sure it is not clipping or limiting. Make a snapshot of the steady-state curve in noise mode (i.e. with slowest response time) with the “-> X” button once the system has stabilized. Keep it on the display by selecting the “X” curve in the memory section. Now iterate over your speakers, adjusting each gain until the resulting measurements match the reference curve as closely as possible. We are only interested in overall loudness, so ignore any minor deviations in frequency response.

A crude level calibration can also be accomplished with a simple metre and a 1 kHz test tone, but then you are susceptible to large adjustment errors if something funny happens to be going on in that narrow test band, like destructive interference with a floor reflection.

5 Aside: speaker equalisation

You should avoid 30-band EQs per output to compensate for acoustic deficiencies of room and speakers. The phase deviations introduced by filter banks at different settings would severely degrade the recreated sound field.

Hence, the best approach is to either do without equalisation altogether, or to ensure matching phase responses in all channels. It is a good (and time-saving) compromise to use the same EQ curve (and algorithm!) for all speakers, which is easily achieved by ganging the EQs on a physical (digital) mixer, or by inserting a mono EQ plugin into Ardour's master bus, which will be replicated automatically for all channels. In theory, it is possible to use custom phase-matched FIR filters for each speaker[3]. However, the benefit will be marginal in most cases, at the cost of additional latency and a setup time and effort that is prohibitive in all but permanent installations.

6 Adding optional subwoofers

Depending on the number of woofers available, different optimised driving signals can be used.

In the case of just one woofer, the obvious choice is to feed it the W channel and to place the speaker it in the centre front (where any “residual” localisation cues will be the least distracting). If two are available, they can either be placed centre front and back, or left and right, and be driven with W +/- X, or W +/- Y, respectively. In the ideal case of four woofers, a standard first-order square decoder can be used. If eight units (and safe rigging anchors) are available, place them in a cube and drive them in first-order periphonic. Higher orders have no advantages at low frequencies.

[3]Another free software package is available for this job: DRC-FIR [Sbr09]. [Net08] shows how DRC can be applied to an Ambisonic listening rig.

The user should experiment with different ratios of W to the directional components. More W will increase the efficiency and overall "boom" that the system can deliver. If emphasis is placed on the directional components, a nice dry trouser-flapping effect can be observed, without the obnoxious pressure on the ears that is often associated with deep bass in enclosed spaces. Obviously, the latter option reduces the maximum sound pressure that can be delivered.

To derive the signals, you have two options: either run two instances of AmbDec in parallel, one for the tops and one for the subs, or create a custom configuration that includes both. The latter is generally more convenient to use, while the former is less work to set up.

Subwoofers should be aligned and calibrated using the procedure described for the tops. With respect to equalisation, the same rules apply. For band separation, use low- and highpass plugins[4].

TESTING AND FINE-TUNING

After the system has been set up and calibrated, it is important to check for defects with well-known program material.

As a first step, a mono signal should be routed to each adjacent pair of speakers, one after the other, to check for polarity errors. If the polarity is correct, a stable phantom image should appear in the middle of the speaker pair under test.

Next, pan the same signal slowly across the entire rig and ensure there are no major changes in loudness.

After any problems have been corrected, send an ambisonic test signal with sustained HF content over the system. Depending on the room acoustics, you will experience irritating phasing effects when moving around near the centre spot. They will be quite pronounced in very dry spaces and inconspicuous in reverberant rooms. Decide whether that phasiness is an issue with the program material at hand – actual audibility will depend on HF content and on whether the audience is seated or invited to move around.

You can trade in some of the phasiness in the centre for a slight loss of localisation precision by "de-tuning" your carefully measured delays. As a quick A/B check, temporarily disable delay compensation in AmbDec to hear the effect.

Next, experiment with the crossover frequency between the LF and Mid/HF decoder[5]. Start with the default (which is always very conservative), and move it upwards a hundred hertz or so, to see if localisation clarity improves or degrades.

If you have the chance, try and compare a discrete stereo signal sent to two speakers with the same signal panned in Ambisonics, and listen for coloration. Likely, you will observe a slight damping in the treble range, which can be corrected with one or two dB boost at 5-6 kHz with a bandwidth of two octaves.

Sometimes, venue constraints such as escape routes may force you to move a speaker away from its optimum angle. Sources will now be drawn to where two speakers are lumped together, and a "hole" will open up on the other side. In this case, try to raise the level of the lone speaker gently, and bring down the level of the other two speakers accordingly. Test the effect with a pink noise signal panned across the problematic area, and aim for constant loudness rather than smooth movement.

Listening Tests

Composers performing their works on a 12-speaker 3rd-order horizontal Ambisonic system have reported a very pleasing sense of space and envelopment, free from timbral defects or obvious imaging deficiencies even at the peripheral areas of the auditorium. However, it should be noted that due to time and equipment limitations, no A/B comparison with traditional discrete loudspeaker replay could be offered to them.

To corroborate the author's hypothesis that virtual Ambisonic sources are indeed a workable approach to multichannel playback, two informal listening tests with different target groups were scheduled, this time including A/B comparisons.

1 Film sound people

The first test session was conducted at the Kunsthochschule für Medien in Cologne, in a Dolby-certified film mixing room with very dry acoustics. The audience consisted of an

[4] 6 and 12dB/oct variants are available as part of the CALF plugin set [Fol10].

[5] For a detailed discussion of dual-band decoding, see [Ger74].

experienced film mixing engineer, a film composer, two students of media arts, a film projectionist and a media arts Ph.D. student. Due to the limited number and at the same time high expertise of the participants, it was decided not to attempt a statistically valid formalized test, but to collect opinions and discuss impressions instead.

Two rigs of identical loudspeakers (K+H O108TV 2-way 8" systems) were set up for direct comparison: an Ambisonic octagon driven in third order, and an ITU 5.0 setup. Both rigs shared a common center speaker. For the test, 5.0 content (both film and music) was reproduced either natively or as 3rd-order virtual sources on the Ambisonic rig. Obviously, the Ambisonic system could not be expected to perform better than the ITU rig, as it had to suffer from the inherent limitations of the 5.0 source material, and then added its own problems. Rather, the goal was to find out whether Ambisonic reproduction could be considered a workable compromise, and to learn more about the perception of its shortcomings.

Participants were encouraged to move around while listening, explore the usable listening area and watch out for position-dependent artifacts.

In the first listening phase (which featured a short animated film), the most strongly evident problem of the rig under test was the tendency of sources to move along with the listener, i.e. to maintain relative direction but not absolute position. This was unanimously deemed unacceptable for the centre channel in film sound reproduction.

Listeners also reported a much wider frontal sound stage, which some found pleasant, and others criticized as "loss of frontal focus".

All participants expressed a clear preference for native reproduction in the film sound use case. The mixing engineer remarked that she could imagine working with an Ambisonic system, but that the increased stage width would have affected her mixing decisions.

In a second listening phase, various 5.0 music snippets were presented: a Händel aria, a piece for organ and percussion, some lounge jazz and an electric reggae track.

The initial reaction to Ambisonic playback was positive – the listeners reported better envelopment and pleasant ambience reproduction. One participant perceived an irritating amount of phasiness and jumping sources depending on pitch (specifically that transient attacks would seem to come from a different direction than the sustained sound). In direct comparison, two test subjects reported audible coloration.

Half of the participants still expressed a clear preference for 5.0, while the other half was undecided. General consensus was that the Ambisonic system was far better suited to music reproduction, where its characteristics would be less obtrusive or even beneficial.

A couple of tentative conclusions can be drawn:

1. **For cinema use, absolute localisation of the center channel is mandatory.** This requirement cannot be met by a third-order Ambisonic system. In a subsequent test run, the centre channel was routed around the Ambisonic encoder and directly to the front speaker, with a gain boost of 4dB to maintain balance (the exact amount will likely vary with the number of speakers and Ambisonic order). This hybrid setup was found acceptable. Since that speaker remained part of the Ambisonic rig, it now carried both the direct C signal and components of L/R/SL/SR. The test content did not show obvious artefacts, but further experiments with LCR-panned material will be necessary to check this method for phasing or comb filter effects.

2. **Focus and stability of localisation were considered critical**, while holes in the side and rear sound stages or distorted ambience reproduction or localisable speakers were not. Even when listening to native B-format recordings, the test subjects did not express much appreciation for the advantages of Ambisonic reproduction. Instead, they reported localisation ambiguities and coloration (which were clearly present, but not particularly disturbing to this author, compared to the benefits).

This experience suggests that people who have had prior exposure to Ambisonics have learned to "decode" a great amount of spatial detail from Ambisonic playback and will usually prefer it, while subjects without such listening experience tend to be irritated by ambiguities and diffuseness. It might well be that Ambisonic listening takes some getting used to, and that the listeners' verdict might improve over time. But then it also implies that Ambi "professionals" tend to overestimate the

impact on (and quality perceived by) “laypersons”, i.e. casual listeners.

3. **The widening of the listening area** during Ambisonic playback postulated by the author **was not substantiated by listener remarks.** Interestingly, the collapsing of the image into a side or rear speaker during 5.0 playback at peripheral positions was not commented on either. Apparently it was considered natural and not perceived as a problem.

4. **The dry acoustics of typical cinema rooms make phasing artifacts very evident.** Additional measures to reduce phasiness (such as de-correlators for the HF band) should be explored under such circumstances.

2 Electro-acoustic musicians

A second listening test was conducted at the Institut für Computermusik und Elektronische Medien (ICEM) at Folkwang Hochschule Essen. During a 2-day workshop, one sound engineer, five students and one professor of electro-acoustic composition experimented with an octagonal setup of Apogee AE-8 15/2” speakers in a moderately ambient room. On the first day, the students set up their own quadrophonic compositions for playback and evaluated both native and Ambisonic renderings. On the second day, a short survey was taken.

For this survey, the author had selected seven excerpts, five from student's works and two from other compositions. Each excerpt was first played back natively, then via Ambisonics. Participants were encouraged to move around to be able to assess the usable listening area. After some time for note-taking, each excerpt was repeated, again on both systems. The subjects were asked to compare the Ambisonic playback to the discrete reference, specifically coloration, stability of localisation, angular fidelity, area of very good resp. usable reproduction, sound of phantom sources, smoothness of movement, ambience, envelopment, source distance and size, personal preference, and appropriateness of the Ambisonic reproduction for the artistic work.

Subjects reported minor shortcomings throughout, but they were mostly deemed inobtrusive. Only for one piece was the Ambisonic system considered unsuitable: it treated the speakers as individual sources, without any correlated information or intent for phantom imaging. Here, any diffuseness or spatial widening was clearly detrimental.

On average, the subjects expressed “no preference” with a slight overall tendency towards native reproduction, and **assessed the Ambisonic system to be “very usable”** for conveying the artistic intention. However, **individual judgements deviated considerably** – in the most extreme case, one person expressed a strong preference for Ambisonics while another considered it clearly unusable for the same excerpt.

Again, the subjects did not share the author's impression of a slightly enlarged sweet spot. On average, the listening area for “good” imaging was found to be 0.5 times the array radius for Ambisonics, and 0.6 for discrete; “usable” imaging was perceived up to 0.7 and 0.8, respectively.

In comments it became evident that **subjects wanted to be able to pinpoint speaker positions** – much to the author's surprise, the ability to do so was considered a matter of reproduction fidelity.

After the survey, some music DVDs were played back in both modes, which provided further interesting insights. One specimen in particular, a recent live production of the Eagles' “Hotel California” showed severe comb filtering of frontal sources, to the point where the result was unusable. Detailed inspection showed negative correlation between the C and L/R channels, a production trick that widens frontal sources over discrete speakers, but lets Ambisonics fall flat on its face. A Pink Floyd Live DVD had the main instruments mixed to L/R exclusively, without significant centre channel content. When switched to Ambisonic playback, the intentionally airy and diffuse sound stage would gel into a very focused centre image, certainly “correct” but very audibly different.

It could be argued that these highly media-specific hacks are not valid testing material, but then again composers will always try to stretch the limits of any given playback system. Any engineer trying to employ Ambisonic techniques will have to be aware of such corner cases.

Most DVD content showed a slight damping of the treble when played on Ambisonics. This effect usually caused a clear preference for discrete rendering, which disappeared after applying some corrective EQ.

One participant nicely summarized the overall consensus in remarking that in tape music, any form of playback is “interpretation”, and that he considered Ambisonic playback to be a different but valid interpretation except in special cases.

Conclusion

Informal listening tests have provided interesting insights into the potential and limitations of virtual speaker sources using Ambisonic panning. While acutely aware of the shortcomings, this author remains convinced that such systems are a very useful tool for electro-acoustic music facilities and can provide a very pleasing and successful concert experience. Since they can be built from commodity hardware and free software, the entrance barrier to a successful deployment is low, in theory. It is the author's hope that this paper contributes to lowering that barrier in actual practice.

References

[Adr10] Fons Adriaensen, the AMB plugin set, containing Ambisonic panners up to third order: http://kokkinizita.net/linuxaudio/downloads/index.html, 2010

[Adr10-2] Fons Adriaensen, AmbDec, a state-of-the-art Ambisonic decoder, l.c., 2010

[Adr10-3] Fons Adriaensen, JAPA, the JACK audio perceptual analyser: l.c., 2010

[Dav10] Paul Davis et al., Ardour digital audio workstation: http://ardour.org, 2010

[Dav10-2] Paul Davis, Torben Hohn, et al., The JACK Audio Connection Kit: http://jackaudio.org/, 2010

[Fol10] The CALF plugin collection, by Krzysztof Foltman and Thor Harald Johansen: http://calf.sourceforge.net, 2010

[Ger74] Michael A. Gerzon: Surround Sound Psychoacoustics, in: Wireless World 12/1974: http://www.audiosignal.co.uk/Resources/Surround_sound_psychoacoustics_A4.pdf, 1974

[Lil10] Jonathan Moore Liles, Non-Mixer, an alternative JACK mixing/routing application: http://non-mixer.tuxfamily.org/, 2010

[Net08] Jörn Nettingsmeier, Ambisonics at Home: http://stackingdwarves.net/public_stuff/linux_audio/tmt08/Ambisonic_Listening_Rig_with_Free_Software.pdf, 2008

[Sbr09] Denis Sbragion, DRC, the digital room correction package: http://drc-firs.sourceforge.net, 2010

[Sle2010] Stephane Letz et al., JACK2 on Windows: http://trac.jackaudio.org/browser/jack2/trunk/jackmp/windows/Setup/src/README, 2010

Acknowledgements

The author would like to thank Martin Rumori and Judith Nordbrock at Kunsthochschule für Medien, Köln, and Prof. Thomas Neuhaus and Martin Preu at ICEM, Folkwang Hochschule Essen, for their kind support and expertise in conducting the listening tests.

Real-Time Kernel For Audio and Visual Applications

John Kacur
Red Hat
19243, Wittenburg
Germany
jkacur@gmail.com

Abstract

Abstract: Many of the Linux Distributions that are dedicated to audio and video make use of the Linux real-time kernel. This paper explores some of the advantages and disadvantages of using real-time. It explains how the real-time kernel achieves low-latency and shows how user-space can take advantage of real-time capabilities. This talk is presented by one of the real-time kernel programmers, and gives an overview of the real-time kernel for audio and video.

Keywords

real-time, linux kernel

1 Introduction

Linux distributions dedicated to audio and video were some of the first distributions to ship the Linux real-time kernel to a general audience. Software such as Jack and Ardour among many other programs are designed to take advantage of real-time capabilities. The standard Linux Kernel has the facillities to do priority based scheduling, but for optimum low latency, the PREEMPT_RT kernel is required. Indeed many features of the PREEMPT_RT kernel have already been integrated into the standard kernel as an option, most notibaly soft and hard threaded irqs. Debugging features such as ftrace and lockdep also were originally designed for the real-time kernel. This paper will explore how the real-time Linux Kernel achieves such low latency, how to configure a system to take advantage of the capabilities it offers, and an introdution to the programming interface from user-space.

2 What is real-time?

When we talk about real-time systems, the most important feature is predictability or determinism. Often people who are new to real-time think it is about raw speed, but this isn't so. Real-time actually sacrifices some throughput performance in-order to achieve predictability and low latency where it is deemed important. A standard kernel will achieve better throughput on average, but a process may run very quickly some of the time but be delayed the rest of the time.. In contrast a real-time system will be less “bursty”, but have predictable performance. The measurement of the degree that time deviates from the average is called jitter, and it is this jitter that is greatly reduced with real-time.

The PREEMPT-RT kernel achieves predictability by ensuring that no operation takes more than 100 microseconds. In some cases the latency is as low as 50 microseconds which is close to the capabilities of the hardware. The low latency is important for high priority processes to accomplish what is required of them on time. By this, we mean the process must do what is required neither too slowly, nor too fast. If you are playing a note in a song, it is just as wrong for it to play too late as it is for it to play too early.

Although some throughput is sacrificed for determinism and low-latency, kernel programmers are working to lower the throughput gap as well.

2.1 Hard real-time vs soft real-time

Surprisingly no single definition exists of hard-real time. Hard real-time can be thought of as a system that never misses it's deadlines, and a soft-real-time system is one that sometimes can be allowed to miss it's deadlines.

The reality is more like a spectrum between these two poles. Since no hardware (or software for that matter) is perfect, it is doubtful that the ideal of hard real-time actually exists. So, the interesting question is, what should the system do

if it misses a deadline? For example, if a deadline is missed, should an event be cancelled or delivered late? If we are talking about video, it is often okay to drop some frames without it being noticeable, so that would be a case where we could cancel meeting a deadline. In cases where we are controlling a machine, it might actually be useless to deliver an event that controls, say a motor after it is too late.

Sometimes a softer version of hard real-time is acceptable. An audio application may have a desired latency of 5 milliseconds, but can occasionally be allowed to miss this deadline, but only if it isn't more than 10 milliseconds. [1}

2.1.1 What is PREEMPT_RT?

PREEMPT_RT is a real-time solution for Linux in which almost everything in the kernel is preemptable. The standard Linux kernel has an option called Preemtible Kernel (Low-Latency Desktop) PREEMPT_DESKTOP, in which all kernel code that is not in a critical section, that means protected by locks, is involuntarily preemtible. The PREEMPT_RT option extends this to make even most critical sections involuntarily preemtible. It does this by replacing kernel spinlocks with rt-mutexes. In addition rt-mutexes are supported by priority inheritance – more on that latter

PREEMPT_RT includes threaded soft and hard irq interrupts – this is now an option in the standard kernel. By making interrupts threaded, they are also now preemtible, they can be scheduled just like any other process, and can be given a priority, which can undergo priority inheritance if necessary.

Another critical component of a real-time system developed for PREEMPT_RT is high resolution timers.

Finally, many tracing and verification features such as ftrace and lockdep were originally developed for PREEMPT_RT, but are now part of the standard kernel.

2.1.2 How does PREEMPT_RT work?

First some background: Voluntary preemption is when kernel code can choose at preselected points in the code to be preempted by higher priority processes. Involuntary preemption is when the scheduler can force a process to stop running in order to allow a higher priority process to run.

The Linux Kernel is designed to run on SMP (Symmetric Multiprocessing) systems. This means that code must safely run on multiprocessors. In order for this to work properly, kernel programmers must identify critical sections. Critical sections are code with data that could be corrupted if it were suddenly interrupted, and later rerun on the same or a different processor. In order to protect this data, various types of locks are used. The most prevelant kind of lock is a spinlock. It works by simply waiting (spinning) while another process holds the lock to a critical section. When hopefully a short time later the lock is released, the waiting process can then grab the lock and continue to work. PREEMPT_DESKTOP can preempt most code, but not code that is running a critical section – that is locked code.

PREEMPT_RT works by converting most spinlocks into sleeping spinlocks, which in turn can be preempted. If you need a lock that can't sleep on -rt, then you identify it by making it a raw_spinlock. Obviously the goal is to have the bare minimum of these raw_spinlock as necessary, since they reduce the areas of code that are preemtible.

2.1.3 What is Priority Inversion and Priority Inheritance?

Priority Inversion occurs when a lower priority task runs at the expense of a higher priority task. Here is one of the simplest ways in which this can occur:

Imagine you have a high priority task A with a resource needed by an even high priority task B. Task B must block until task A frees it's resource. Then imagine a task C with a priority between A and B. Because the priority of C is higher than that of A, it preempts A and steals the processor. Thus we have process C running at the expense of the higher priority task B The solution that PREEMPT_RT provides to deal with this siutation is called priority inheritance. Priority Inheritance works by having higher priority tasks temporarily lending their prirority to lower priority tasks that hold resources that they require. So in the situation described here, task B would lend it's priority to task A, and A's prirority would rise to the same level as A's. That way task C would not be allowed to preempt A. Task A would run until it freed the resource required by B. B would then be scheduled to run, and A would return to it's original priority.

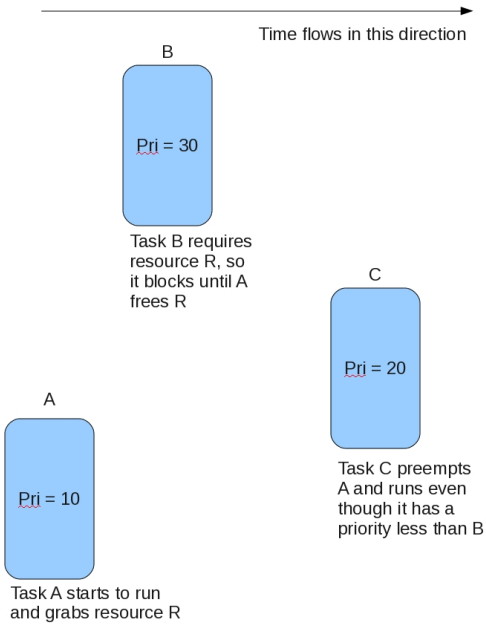


Fig.1. Without priority inversion, C preempts A and runs at the expensive of higher priority B

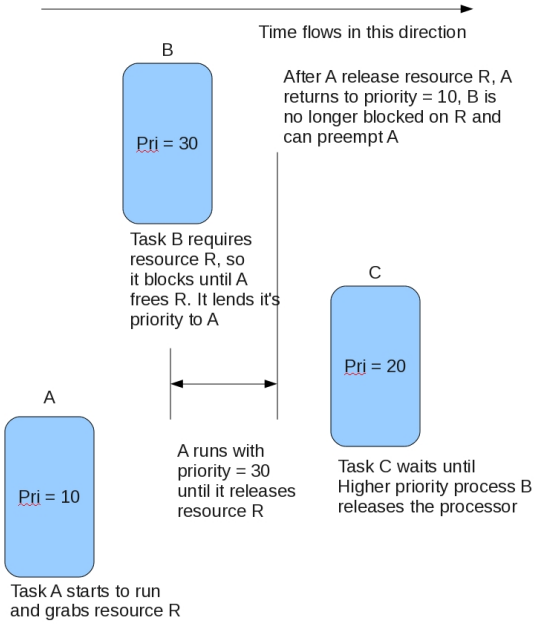


Fig.2. With priority inversion, B lends it's priority to A until A frees resource R, allowing B to run

3 Configuring real-time

3.1 Fetching, building and configuring the kernel

Some of the audio distributions are a little slower than the mainline distributions to update. So, if you need the latest hardware support, or simply want to try out the latest and greatest kernel, you may have to fetch, compile and install it by hand. Here's how to do so.

You can fetch the latest rt-patch (and archived ones as well) from:
http://www.kernel.org/pub/linux/kernel/projects/rt/

The latest one when I wrote this document (March 2010) was patch-2.6.33.1-rt10.bz2

From the name we can tell that it patches linux-2.6.33 with the stable patch linux-2.6.33.1 so we will need to fetch those as well.

```
wget
http://www.kernel.org/pub/linux/ke
rnel/v2.6/linux-2.6.33.tar.bz2
wget
http://www.kernel.org/pub/linux/ke
rnel/v2.6/patch-2.6.33.1.bz2
wget
http://www.kernel.org/pub/linux/ke
rnel/projects/rt/patch-2.6.33.1-
rt10.bz2
```

Now untar and apply the patches

```
tar xjf linux-2.6.33.tar.bz2
cd linux-2.6.33
bunzip2 -c ../patch-2.6.33.1.bz2 |
patch -p1
bunzip2    -c    ../patch-2.6.33.1-
rt10.bz2 | patch -p1
```

The rt kernel is also available via git for those who know how. For more information, see the rt-users mailing list, or Rtwiki. (below)

```
linux-rt-users@vger.kernel.org
https://rt.wiki.kernel.org/
```

You can base your configuration on your distribution's configuration. Look in your /boot directory for the config that matches your current running kernel, or copy it from /proc/config.gz if available as the real-time kernel's .config file. This

will save you from having to answer all the questions when you run make menuconfig and friends. A few choices are necessary for real-time. Under “Processor type and features”, it is important that you select Complete Preemption (PREEMPT_RT) which will automatically select Thread Softirqs (PREEMPT_SOFTIRQS) and Thread Hardirqs (PREEMPT_HARDIRQS) for you. Also recommended are TREE_PREEMPT_RCU, Tickless System (Dynamic Ticks) (NO_HZ), and High Resolution Timer Support (HIGH_RES_TIMERS). You can also select FTRACE without any noticeable effect on latency when not enabled. One more note of caution, in the “General Setup” section, for “Choose SLAB allocator”, you must choose SLAB as SLUB is not currently supported.

3.2 Configuring your system

There are many possible ways this can be done, but the key element is a way to specify either which users or programs get real-time privileges. One possible scheme is to create a group called “realtime”, and make sure any user that requires real-time privileges is assigned to it. Other possible groups that are commonly used are “audio”, or “jackuser”. The following creates the group “realtime”, and adds username jkacur to it.

```
sudo groupadd realtime
sudo usermod -G realtime jkacur
```

3.2.1 /etc/security/limits.conf

Here we need to modify users or groups to give them the privileges they require to run with real-time priorities, without being superuser.

Typical values would be.

```
@realtime  soft  cpu   unlimited
@realtime  -     rtprio     100
@realtime  -     nice  -20
@realtime  - memlock    unlimited
```

These allow users that belong to the realtime group to run with unlimited cpu time, the maximum real-time priority, the maximum nice value, and to lock unlimited amounts of memory. You may have to reboot your system before these changes take effect.

3.2.2 /proc/sys/kernel/sched_rt_runtime_us /proc/sys/kernel/sched_rt_period_us

The default values here are:

```
sched_rt_period_us 1000000
sched_rt_runtime_us 950000
```

The first value is the amount of time in microseconds that represents 100% of the CPU bandwidth. [3] The second value is the amount of time in microseconds that real-time processes are allowed to run. This means that 1000000 - 950000 = 50000 microseconds or 0.05 seconds are reserved for non-realtime tasks, in other words for SCHED_OTHER tasks to run. This is a soft-real-time behaviour, designed to protect you against runaway real-time processes that could hijack your system. You should definitely leave this at the default setting for testing. However, when you are ready to go to production, and want something closer to hard-realtime behaviour, you can set sched_rt_runtime_us to -1. This will now allow real-time tasks to monopolize the processor 100% of the requested time. Once again, use caution here, because this means that a misbehaving user-space program can make your system unuseable.

To change the setting, do:

```
sudo -c 'echo -1 >
/proc/sys/kernel/sched_rt_runtime_
us'
```

3.2.3 IRQs

Now that IRQs are threads, they can be given a priority. An -rt distribution will set these values with a start-up script. However, for optimum performance, and because of differences between your particular hardware, you may want to tune these values.

To see the current values on your system, use ps with the -o which is the option to control the output. We will select, pid, cls for the scheduling class, see table below

CLS Scheduling Class reported from ps	
-	not reported
TS	SCHED_OTHER
FF	SCHED_FIFO

RR	SCHED_RR
?	unknown value

Table 1.

rtprio for the real-time priority, prio for the priority, nice, and finally the cmd. [2]

For example, on an untuned vanilla distribution running a real-time kernel, looking at the tasklets which are similar to the bottom halves of traditional interrupts: (that is, the part of the interrupt that is scheduled to run later than the part that is serviced right away)

```
$ ps -eLo
pid,cls,rtprio,pri,nice,cmd | grep
-i tasklet
10  FF  49  89 - [sirq-tasklet/0]
25  FF  49  89 - [sirq-tasklet/1]
```

These values can be changed using the ***chrt*** command. Chrt changes or retrieves the real-time scheduling attributes of a process or task. It is usually installed by default on most distributions. It is part of the util-linux-ng package.

Tasklets should run higher than most real-time processes so 82 would be a reasonable value to set them to on this machine. [4]
To change the above.

```
$ su -c "chrt -f -p 82 10"
$ su -c "chrt -f -p 82 25"
$ ps -eo
pid,cls,rtprio,pri,nice,cmd | grep
-i tasklet
10  FF  82 122 - [sirq-tasklet/0]
25  FF  82 122 - [sirq-tasklet/1]
```

In general hardirqs (hardware interrupts) should be set slightly higher than the soft-interrupts (which you recognize by “sirq”). With tasklets at 82, 85 would be a reasonable priority for hardirqs.

Certain threads have the highest real-time priority, of 99

```
ps -eo pid,cls,rtprio,pri,nice,cmd
| grep -i ff | grep 99
3  FF 99 139   - [migration/0]
14  FF 99 139   - [posixcputmr/0]
15  FF 99 139   - [watchdog/0]
17  FF 99 139   - [migration/1]
18  FF 99 139   - [posixcputmr/1]
29  FF 99 139   - [watchdog/1]
```

These are critical to the system, and it is not recommented that userspace real-time process compete with them. Some papers suggest, and some distributions and scripts set the audio and video processes to run between the highest priorities and that of the tasklets and hardirqs. For this to work well, the audio and video processes would have to be very well written and run with real-time priorities for very short periods. I would suggest that audio and video run at the highest priority just under the tasklets. Given the scheme presentented here, 80 would be a good value.

```
@audio  -  rtprio     80
@audio  -  nice       -20
```

This should give the audio / video processes high enough priorities to achieve very low latency without interfereing with any systerm critical real-time processes, which in turn could degrade the overall perforance of a system.

4 Tests, Benchmarks, measuring

4.1 Rt-tests

Rt-tests is a suite of tests to stress various parts of the real-time kernel. The core of it is cyclictest written by Thomas Gleixner. It is now being maintained by Clark Williams, and can be fetched via git here:
git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

The idea behind cyclictest is simply to fire off a number of high resolution timers from real-time threads and measure the difference between the time the timer is supposed to conclude and the time that it actually concludes. cyclictest can be used to measure your system's minimum, average and maximum latencies, and can thus be useful for tuning. Here is a typical run, just accepting the defaults.

```
./cyclictest
defaulting realtime priority to 2
policy: fifo: loadavg: 0.13 0.06
0.01 1/319 6879
T: 0 ( 6879) P: 2 I:1000 C:   7889
Min:     16 Act:  117 Avg:  107
Max:     298
```

The above shows two fifo threads, running at priority 2. The timer is firing at intervals of 1000 nanoseconds. (1 microsecond). So far, 7889 cycles occurred.

The results were a minimum latency of 16 microseconds, and average latency of 107 microseconds, and a maximum of 298.

4.2 hwlatdetect and SMIs

Hwlatdetect is a tool supplied with rt-tests to try to detect unexplained hardware latencies. One source of hardware latencies that we have very little influence over is SMIs – System Management Interrupts. These interrupts cannot be handled by software and can last tens of microseconds. It can be very dangerous for the stability of your hardware to attempt to turn them off too, because they often are in charge of such functions as making sure that your cpu doesn't overheat. In some cases with extreme caution, if your BIOS allows you to, you can turn some of them off. In some cases the SMI routines have been poorly written and the best we can do is bring to the attention of hardware makers that an SMI on a particular piece of hardware is taking an excessive amount of time.

Hwlatdetect works in cooperation with the hwlat_detect.ko kernel module. This kernel module comes as a standard feature in recent rt kernels. It tries to detect SMIs by monopolizing a cpu for a long time, with interrupts disabled. Since the only thing that could interrupt such a cpu is an SMI, any gaps detected in the time that hwlatdetect calls stop_machine are thus likely due to SMIs.

4.3 Rteval

Rteval is a program for evaluating the latency and performance of your system. The idea is simply to stress your system with some standard linux benchmarks such as dbench and hackbench, to see if they have an effect on the real-time capabilities of your system as measured by cyclictest.

You can fetch it via git here:

```
git://git.kernel.org/pub/scm/linux
/kernel/git/clrkwllms/rt-tests.git
```

5 Userspace Programming, a whirlwind tour and a caution.

What does a real-time program look like in userspace? Surprisingly simple. Of course it is a real art to identify which processes need to run with real-time priorities, and in most cases it is optimal to run with a higher priority for the minimum amount of time necessary.

Here is a whirlwind tour of some of the calls.

To raise the priority of a process, you can use standard POSIX calls such as:

```
sched_setscheduler(pid_t pid, int
policy, const struct sched_param
*param)
```

This call sets both the policy (SCHED_FIFO in the example below) and the priority. If the process id is set to 0, the parameters are applied to the calling process.

For example. (error checking code omitted)

```
struct sched_param param;
struct sched_param *pparm =
&param;
pparam->sched_priority = 50;
sched_setscheduler(0, SCHED_FIFO,
pparam);
```

You can retrieve the current scheduling policy with

```
sched_getscheduler(pid_t pid)
```

To retrieve or set the priority, you can use:

```
sched_getparam(pid_t pid, struct
sched_param *param)
sched_setparam(pid_t pid, const
struct sched_param *param)
```

To determine the minimum and maximum priorities allowed for a particular scheduling policy on your system, you can use:

```
int sched_get_priority_max(int
policy);
 int sched_get_priority_min(int
policy);
```

To prevent unexpected page faults of real-time programs, it is common to lock all current and future memory.

```
mlockall(MCL_CURRENT | MCL_FUTURE)
```

However, one caution, priority inheritance is only supported by rt_mutexes in the kernel. In user space this translates to pthread_mutexes. Ordinary unix semaphores in user-space are not supported by priority inheritance. The reason for this is, that it is not possible (or easy in any case) to identify the owner (pid) that blocked around a semaphore.

If you don't want to use pthread programming but want support for priority inheritance, you can create a lock using shared memory and a pthread_mutex that the usual POSIX style process calls can access. For an example of how this is done, see pip_stress in the rt-test suite.

6 Conclusion

Linux is well suited as a low-latency real-time operating system for audio and video. With a little bit of back-ground knowledge, an end user can tune his / her system for maximum performance and real-time determinism. This paper should get you well on your way to experimenting and tuning your real-time Linux system for optimal audio and video performance.

7 Acknowledgements

Ingo Molnar, Thomas Gleixner, Paul McKenney, Steven Rostedt, Peter Zilstra and countless others for creating PREEMPT_RT.

References

[1} Paul McKenney, 2007.

SMP and Embedded Real Time, January 2007 Linux Journal, issue #153 – There are more examples here illustrating the differences between hard and soft real-time.

[2] http://subversion.ffado.org/wiki/IrqPriorities

does an excellent job and explaining how to set irq priorities for real-time audio.

[3] See Documentation/scheduler/sched-rt-group.txt in linux kernel source 2.6.33.1-rt11 or later

[4] The suggested values are based on Red Hat's MRG distribution.

Other Sources of information

“Real Time” vs “Real Fast”: How to Choose?

Paul McKenney, Eleventh Real-Time Linux Workshoop, Dresden, 2009

Programming for the Real World, POSIX.4

Bill O.Gallmeister, O'Reilly & Associates, INC. Copyright 1995

Web Resources

https://rt.wiki.kernel.org

Approaches to Real-time, Jon Corbet
http://lwn.net/Articles/106010/

Realtime preemption, part 2, Jon Corbet

http://lwn.net/Articles/107269/

A real-time preemption overview, Paul McKenney

http://lwn.net/Articles/146861/

Re-Generating Stockhausen's "Studie II" in Csound

A Study About Algorithmic Composition

Joachim HEINTZ
Incontri - Institute for new music, HMT Hannover
Emmichplatz 1
30175 Hannover, Germany
jh@joachimheintz.de

Abstract

Stockhausen's "Studie II" (1954) is one of the classical works of electronic music. Written at a time when Computers played no role in the production of sound, it exhibits a way of composing which is quite similar to programming. This can be shown by re-programming the complete piece just with the input of five numbers. Beside this reduction - which is made in Csound - the compositorial decisions come to the fore, showing a flexibility and variability of algorithms[1] which can be inspiring and challenging still today.

Keywords

Music Composition, Audio Programming, Csound, Algorithmic Composition, History of Electronic Music

1 Introduction

Stockhausen's *Studie II* which was composed and realized in 1954, is well known as the perhaps earliest and strictest application of the serial technique of composition[2] to a piece of electronic music. The interest of Stockhausen and other serial composers on working with synthesized sounds suggests itself, because only in this field can also the timbre be composed in a serial way. But this paper wants to point out yet another relationship between this piece and electronic music: the close correlation between serial composing and programming. As the serial technique can be understood as a bundel of definable rules, it can be programmed.

By doing this, two different goals are persued. One is more sportive: proving the thesis of programmability by re-programming *Studie II* in practice, based on a series of 5 numbers as input. The second one is more important as a study about composing in general and about algorithmic composing in particular: How is the "machine" built? Is there something like "put n formulas in it and let it play for m seconds", or are there changes? Did the composer react to the result he sees (or hears) in the process of composition, or is there an automatism? And how can the irregularities we see in the score be judged: as errors, or intentional decisions, or both?

2 Building the Musical Material

When a composer wants to write a piece of purely electronic music, a word from the bible may come to his mind: "And the earth was without form, and void" (Genesis 1, 2). There is nothing which is self-confident; no ambitus, no scales, no chords, no times. If one doesn't want to use scales and durations "because they are there", one has to build ones own structure from the "void". So the first task in re-generating *Studie II* is to repeat what Stockhausen did in preparing the musical material:

a) to build a collection of series which are used for many different purposes, and

b) to build a pool of distinct values for frequencies, durations and intensities.

[1] I use the terms "algorithm" or "algorithmic composition" in a wide, general way. Algorithm in my use is an instruction which can be executed in a definite way to come from a state A (the input) to a state B (the output). I call "algorithmic composition" a way of composing music in which the whole generation of pitches, durations, sounds etc. can be described by a set of such definite executable instructions. In terms of computers, it means: it can be programmed. So "algorithmic composition" is in my understanding in no way restricted to a simple method like: "apply this formula(s) to these notes and you get the piece." In contrary, the algorithms can be different and complex, and they are derived from musical decisions. If the term algorithm is used in a strictly formula-like mathemathical way of changing state A to state B, it should not be applied to Stockhausen's Studie II (thanks to Gottfried Michael Koenig for pointing me on the issue of the use of "algorithmic" etc. in this piece).

[2] "Serialism" means "integral serialism" here: organizing all relevant parameters (not just the pitch) by series.

2.1 Building the Series (Number Squares)

The main input for *Studie II* is a serie of 5 numbers: 3 5 1 4 2. By a number of procedures, this series is expanded to a set of 2 x 5 "Number Squares", each consisting of 5 x 5 numbers. The first one, as an example, is the following:

```
3 5 1 4 2
5 2 3 1 4
1 3 4 2 5
4 1 2 5 3
2 4 5 3 1
```

For the re-generation of *Studie II*, these collections of 25 numbers (each building one series) are stored in an array for further use. In Csound, an array can be built as a "function table". So practically the program does at this point the following:

a) It defines functions (in Csound: "User Defined Opcodes") which perform the various modifications: from the starting 5 numbers to the first square (series), and then by a different method from the first square to the next four, and then by an again different method to the next five ones.

b) It fills 10 function tables with the results.

As a simple example, this is one of the functions for transforming:

```
  opcode SS2_TP1Val, i, iiii
;calculates a new value from ival and the
transposition interval interv, within the
scope of iminval and imaxval
ival, interv, iminval, imaxval xin
ivalneu   =         ival+interv
ires      =         (ivalneu>imaxval?\
  ivalneu-imaxval : (ivalneu<iminval?\
  ivalneu+imaxval : ivalneu))
          xout      ires
  endop
```

If this function is applied in a loop which looks for the distance ("transposition") between one number and the next, it returns the new value. So by

```
ival       tab_i           indx, ift
indxnext   =          (indx+1 < iftlen ?\
             indx+1 : (indx+1) - iftlen)
ivalnext   tab_i      indxnext, ift
interv     =          ivalnext - ival
ires       SS2_TP1Val itransval,\
              interv, iminval, imaxval
           tabw_i  itransval,indx,ifttrans
itransval  =          ires
           loop_lt   indx, 1, iftlen, loop
```

the function table ift = [3 5 1 4 2] is transformed to the new function table ifttrans = [5 2 3 1 4], if the starting value is 5. If this procedure is repeated with the starting values 3 5 1 4 2 (by which the starting sequence of 5 numbers are simply read vertically), the first number square (see above) is produced.[3]

2.2 Building the Pools for Frequencies, Durations and Intensities

For the generation of possible values for frequencies, Stockhausen used a method which is quite similar to the well-known approach of an equal tempered scale. The partition of the octave in 12 equal steps can be described as 12th root of 2 as the relation for one frequency to the next possible. Stockhausen used essentially the same method, but as he was possessed with the number 5 in this piece, and also wanted to build new harmonies instead of the usual cromatic scale, he decided to take the 25th root of 5 (or in other words, the 5*5th root of 5 ...). This division of the double octave plus pure major third in 25 equal steps gives a multiplier of 1.066495... - which is quite close to the cromatic semitone of 1.059463.... So his gain is to have something rather common which nevertheless leads to new sounds.

It's easy to build this scale by recursively multiplying the starting value of 100 Hz with $5^{1/25}$. It is stored again in a function table.

The durations are built by the same method, starting with 2.5 cm (0.5 * 5 ...) as the minimal length of the tape,[4] and recursively multiplying by $5^{1/25}$ for getting 61 different durations.

For practical reasons, the intensities are built from 0 to -30 dB in one dB steps. For bringing this table to the same length as the frequency and duration table, the dB values (exept 0 dB) are mirrored, so that also the intensities are collected in a table of 61 values.

3 Building the Events in the Five Parts

The collection of 10 series - each of them containing sequences of the numbers from 1 to 5 - and the pool of frequencies, durations and intensities is now used to build the events. The whole piece is clearly divided in five parts; the length of each part is given by a certain run of series and different meanings or interpretations of them.

---

[3] The methods for building the next four number squares from the first one are different and more complicated. I can't describe them here. The number squares 6-10 are built by reversing the first five ones.

[4] The tape speed was 76.2 cm/sec, so 2.5 cm equates to approximately 1/30 second.

3.1 First Part

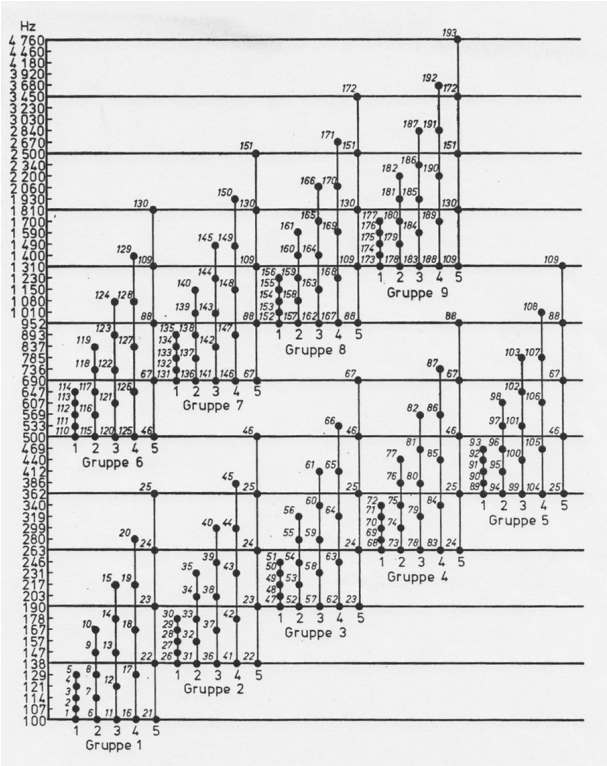
The whole piece can be considered as a collection of in total 5 * 75 + 5 = 380 single events. Each event needs six determinations:

a) the base frequency
b) the mixture built on this frequency
   (there are *5* mixtures ...)
c) the duration
d) the start time
e) the maximum intensity
f) the envelope

The method which Stockhausen used for obtaining the data for the events can't be seen as simple and straightforward. It is rather complex, giving the impression of a kind of organic structure, organic movement, as if there are worms which move under certain circumstances. I will describe the first part more in detail, so that then the other parts can be discussed in relation and deviation to it.

3.1.1 Frequencies and Mixtures

For obtaining a base frequency for an event, the frequency table (cf. 2.1) is structured by 9 groups [5]:



Now four series are used to get a frequency and a mixture. The first series (A) determines one of the 9 groups in abstract. Let the first value of this serie be 3. Then a "transposition" value (B) is applied to this. This "transposition" works like a transposition in music: a transposition by 1 (prime) lets the note as it is, a transposition by 2 transposes it 1 step up, and so on. Let this transposition value be also 3. The result of the abstract group value, transposed by the transposition value (A t B) is then 5. This is the number of the group. Then two other series are used to determine the column of the group (C) and the position in this column (D). Let the values of the column and the position be 5 respective 3, then we have as the base frequency of the first mixture 690 Hz. The kind of mixture equals the column value, so we get mixture number 5 and by this the frequencies of the other four sine tones above 690 Hz as 952, 1310, 1810 and 2500 Hz.

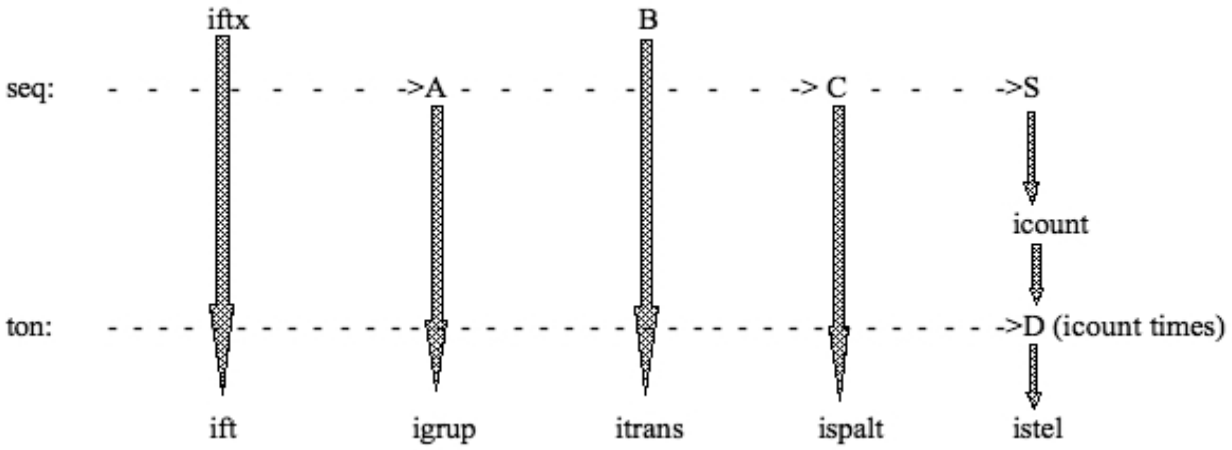
It is important that the four series are not running at the same speed but asynchronously. There is one "superior" series (S) which determines how many repetitions of A and C are performed. For instance, if the first five values of A are 3 5 1 4 2, and the values of S are 2 4 5 3 1, the result is A with these repetitions:

A: 3 3 | 5 5 5 5 | 1 1 1 1 1 | 4 4 4 | 2
S: 2 4 5 3 1 (repetitions)

This is the way A and C are running. The transposition value B does not change during the whole part.[6] And the series D (giving the positions) is running entirely without repetitions. So there are three different movements: A and C are moving in a "regular speed", ruled by S; B is not moving at all; D is moving fast. In writing a function for this, it results in two nested loops:

[5] Heinz Silberhorn, Die Reihentechnik in Stockhausens Studie II, Rohrdorfer Musikverlag 1980, S. 17

[6] Obviously it was the plan to have one transposition value for each part: 3 5 1 4 2. But Stockhausen changed this plan later (see 4.3).



In Csound Code:

```
indxcount     =                 0
indxtonabs    =                 0
seq:
icount        tab_i             indxcount, iftcount
igrup         tab_i             indxcount, iftgrup
ispalt        tab_i             indxcount, iftspalt
indxton       =                 0
ton:
istel         tab_i             indxtonabs, iftstel
ival          SS2_ValsAusGTSS   iftbas, igrup, itrans, ispalt, istel
              tabw_i            ival, indxtonabs, iftout
indxtonabs    =                 indxtonabs + 1
              loop_lt           indxton, 1, icount, ton
              loop_lt           indxcount, 1, ftlen(iftcount), seq
```

3.1.2 Durations and Starts of Events

The method for obtaining the durations is essentially the same as for the frequencies. So, in the code the same function (called SS2_MkParamTab_Meth1) could be used. It returns a function table which contains the durations of the 75 events of the first part.

As for the starting times of the events, two different decisions have been made. The first one is a general one for all the event starts in *Studie II*: A second series of durations is generated. But these durations are not read as durations, but as starting points: they are imagined to succeed each other immediately, and the resulting times are captured. For instance, if these ghost-duration serie had the values 20.9, 45.3, 104.6, 71.2, 91.9, 91.9 (in cm), the resulting starts are 0, 20.9, 66.2, 170.8, 242.0, 333.9, 425.8.

The second decision is different for each of the five parts of *Studie II*. How are these values used?

In the first part, Stockhausen decided to give the events a somehow melodic structure. If one group consists of 3 events, just the time for the first one is taken from the ghost serie. Then event 2 starts at the end of event 1, and event 3 at the end of event 2:

In the code, instead of calculating all the 75 starts, just the 25 starts of the groups (sequences) are calculated. These values are then taken as the start values of a loop, in which the durations are shifted by the previous ones.

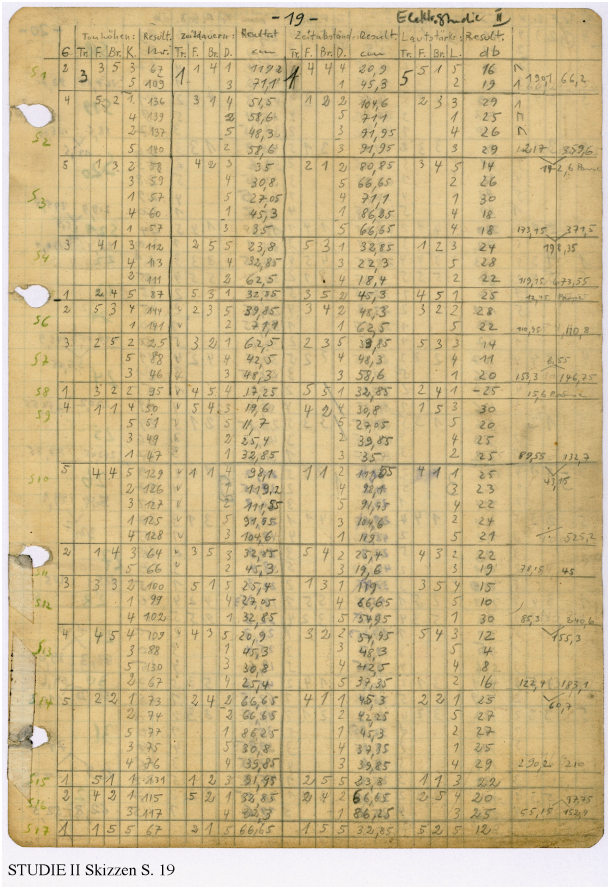
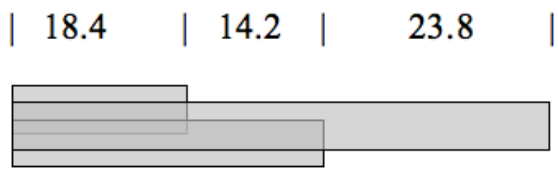


Figure 1: Stockhausen's sketches for Studie II, part 1. © Archiv der Stockhausen-Stiftung für Musik, Kürten (www.stockhausen.org)

3.2 Second Part

The main difference of the second part to the first one is the way the single events are treated. They no longer succeed each other but are played simultaneously, building "chords". These chords have either common starts or common ends. The durations are working like additions; if the durations of a group of 3 events are 18.4, 14.2 and 23.8 cm, and the 3 events start at the same time, the first event has 18.4 cm, the second has 18.4 *plus* 14.2 cm, and the third has 18.4 + 14.2 + 23.8 cm:



The realization as Csound Code works again with two nested loops (see above 3.1.1). Here the inner loop has to add the durations:

```
note:
;common start: first duration normal,
                    all others are added up
if ienv == 1 || ienv == 2 then
 ;durations from normal table
idurnorm      tab_i  indxnotabs, iftdur
 ;real duration = sum of preceding
  durations of this sequence plus idurnorm
idur          =      idurnorm + iduraccum
 ;this is the next value for the
                        assembled durations
iduraccum     =      idur
 ;write durations to iftout
              tabw_i idur,indxnotabs,iftout
;common end: last duration normal, all
            others are added up reversely
else
 ;counting down
indxback = indxabsseq+icount - indxnote-1
idurnorm      tab_i  indxback, iftdur
idur          =      idurnorm + iduraccum
iduraccum     =      idur
              tabw_i idur, indxback, iftout
endif
indxnotabs    =      indxnotabs + 1
       loop_lt   indxnote, 1, icount, note
```

3.3 Third, Fourth and Fifth Part

In order to not go into too much detail but to discuss more general questions in the next chapter, the focus here is taken on the main differences between the parts and its methods.

In the **third** part all events are isolated; like *staccato* tones on a piano. For widening the ambitus to potentially all frequencies, the method of using the series (cf. 3.1.1) changes in the way that the previously constant B now also changes like the A and C series.

(see figure 2, next page)

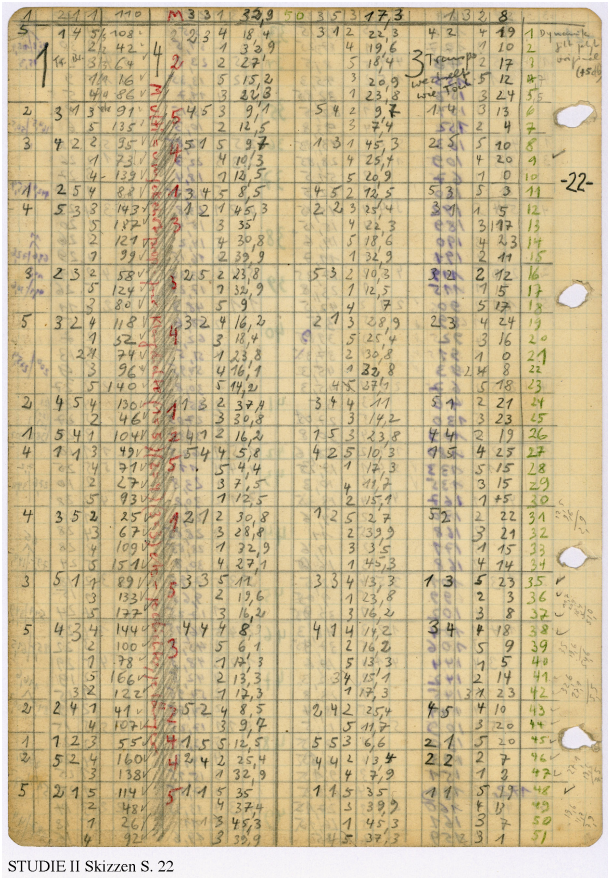


Figure 2: Stockhausen's sketches for Studie II, part 3. Note that the "Tr" (Transposition) is now changing. © Archiv der Stockhausen-Stiftung für Musik, Kürten (www.stockhausen.org)

In the **fourth** part this method is kept. Besides, the way of building chords from part 2 is repeated.

Part **five** is a mixture of all the different methods. It consists of "melodic" sequences like in part 1, "chord" structures like in part 2 and 4, and "staccato" passages like in part 3. So the inner loop of the start times has to differentiate between the cases:

```
ton:
istartdiff3 tab_i indxtonabs, iftstarts5a
;starts if direct from iftstarts5a/typ=3
istartabs   =     istartabs + istartdiff3
;typ1: concatenating the notes of one
                                      sequence
if ityp == 1 && indxton < icount-1 then
idur1       tab_i  indxtonabs, iftdurs
idurshift   =     idurshift + idur1
istart1     =     istartseq + idurshift
;value for the next note after the
                    duration of this note
      tabw_i istart1, indxtonabs+1, iftout
;typ2: chords
elseif ityp == 2 then
; for common starts
if (ienv == 1 || ienv == 2 || ienv == 5)
       && (indxton < icount-1) then
istart2     =     istartseq
;else starting point as difference to the
maximum duration of the sequence
else
idur2     tab_i   indxtonabs, iftdurs
istart2   =    istartseq + (imaxdur-idur2)
endif
;value for this note from this calculation
         tabw_i istart2,indxtonabs,iftout
;value for the next note from normal table
     tabw_i istartabs, indxtonabs+1,iftout
;typ=3: isolated events, or last note
                    from a typ=1 sequence
else
     tabw_i istartabs, indxtonabs+1,iftout
endif
indxtonabs   =      indxtonabs + 1
             loop_lt indxton,1,icount, ton
;indxenv++ if value was used
indxenv      =      (ityp == 2 ?
                    indxenv+1 : indxenv)
```

4 Results, Problems, Conclusions

Studying *Studie II* is not meant as an act of museal interest. The principal questions here are:

- Can this composition which was made entirely by hand - as a composition and as a realization - be regenerated by program code? If yes - why?
- Does everything run smooth or are there serious problems in the re-generation?
- What can be learned by Stockhausen's way of generation for today's algorithmic compositions?

4.1 Can *Studie II* be regenerated as program code?

Yes, it can. It can be proven that in principle all the events of the piece can be re-generated, by just starting with the numbers 3 5 1 4 2 and executing some (describable = programmable) methods. It has been done in Csound, and it should be possible to do it in any other audio programming language with a similar power as Csound. The reason *why* this is possible leads to the method of composing which Stockhausen applied in this piece (and probably tests for the first time so consistently), the so-called serial way of composing. Serialsm can be decribed as a method which situates the decisions of a composer in the space of structures and methods. *Studie II* structures the material (frequencies, durations, timbres, intensities), starting "from nothing", and defines methods of generating events, using series which are also generated in a regular way. So there is a very interesting interrelation between a compositional technique which came from a purely musical development (Schoenberg - Webern - Messiaen)

and the emergence of the computer and mainly the first high-level programming languages (Fortran, Lisp) exactly at the same time in the mid-50s.

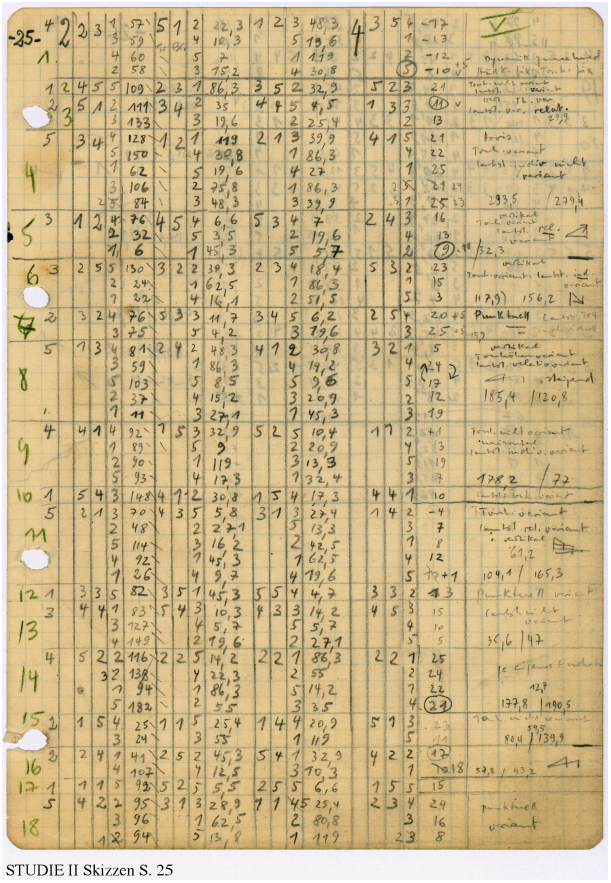
4.2 What are the difficulties in the regeneration of *Studie II*?

The problems in the regeneration of *Studie II* can be divided into three different categories.

First, there are obviously a number of errors in Stockhausens calculations of some frequencies or durations. These deviations of the correct calculation are recorded truly in the work of Silberhorn[7]. They are easy to understand in a work with 380 events, each event requiring at least four calculations, all done by hand and paper.

But, **second**, there are certain deviations from the "actual" result, which are more complicated. It seems that sometimes Stockhausen decided intentionally for a different single event. In some cases it seems he avoided a nearly-repetition, in other cases corrected densities[8]. And in part five one gets the impression that he often decided "on the fly", on the basis of the predefined structures, what he preferred for this situation. From the point of programming, these latter kind of deviations are crucial. If they are more than exceptions, the re-generation makes no sense, because it has just to translate the exceptions, and this is no more a re-generation but a transcription of the score. In my opinion, in the first four parts of *Studie II*, there are just deviations by error or small exceptions, but the fifth part comes close to the point of changing to a new quality. A closer look at the sketches proves it:

(see figure 3, next column)



STUDIE II Skizzen S. 25

Figure 3: Stockhausen's sketches for Studie II, part 5. Note the spontaneous side notes about the musical content. © Archiv der Stockhausen-Stiftung für Musik, Kürten (www.stockhausen.org)

Third, there are some incertainties with respect to the generation of the envelopes. It can be shown by accurate analysis that part two and four use series of five shapes for the envelopes.[9] This should be possible to show also for part one. Part three uses just one envelope, and part five is also in this respect a combination of all the previous shapes. So I believe the envelopes can be generated by series, too, and could show the series for parts two and four, but the origin of the series are obscure.[10]

[7] pp 128-167

[8] cf Silberhorn pp 118-125

[9] In my analysis and transcription, the series for part two and four are the following:

1 2 4 3 5	2 5 4 3 1
1 4 3 5 2	2 1 5 4 3
1 4 5 2 3	2 3 1 4 5
4 3 1 5 2	2 5 1 3 4
3 1 4 5 2	2 3 5 1 4

[10] Unfortunately, nothing could be found in the sketches of Stockhausen which clarifies the origin of the series which are mentioned in the previous footnote.

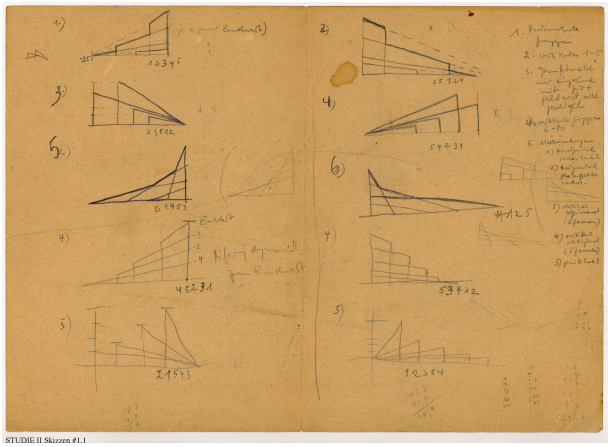


Figure 4: Stockhausen's early sketch for Studie II, containing a list of possible envelopes. © Archiv der Stockhausen-Stiftung für Musik, Kürten (www.stockhausen.org)

4.3 What does the regeneration of *Studie II* show regarding questions of algorithmic composition?

With a certain right, *Studie II* can be considered an early model of algorithmic composition, the application of serial compositional methods in the area of electronic sound (and today: computer music). Re-generating this composition means: coming to the root of musical decisions; being where the composer was while working. From this, some important conclusions can be drawn by this work.

1. *Change the Methods Depending on the Musical Structures you Want to Get*
   It is impressive and inspiring to see how Stockhausen arranges and adapts the flow of the series depending on the musical structures and differences he wants to get. This way of composing can be compared with an architect of landscapes, regulating the streams of water, planting this here and that there. It is totally different from a way of "composing" which can unfortunately not be considered as an exception in the field of algorithmic composition: put one algorithm in the machine and let the machine play for a long long time ...
2. *Listen to the Results and React as a Musician Instead of Sticking with Principles*
   It is extremely interesting to see how Stockhausen changed his own plans, obviously because he saw some musical problems if he would have kept them. One example is the generation of the frequencies. He started with a constant "transposition" (cf 3.1.1) value of 3 in the first part and 5 in the second part. We do not need too much inspiration to suspect that the values of the other three parts were planned to be 1, 4 and 2, so representing the king's crown (3 5 1 4 2) in this field. But then, at the beginning of part three, something exciting happens. Stockhausen decided to break the whole method: not to have one constant transposition in the whole part, but changing this value by running a series. Why? Because he saw that insisting on the method would have been "too consistent", this means: predictable, and wouldn't allow the staccato sequences of part three to have larger leaps. This is the paradigm of a musical reasonable decision which breaks the "purity" of a method.
3. *Listen Also to the Single Events and Change Them if Your Ear Judges So*
   As it has been discussed (see 4.2), Stockhausen changes not just methods, but also single events, or even small groups of events, according to the musical scope or a certain situation. Generalizing this, we can say that a balance must be found in algorithmic compositions between the coherence of methods, of the generation of musical structures on one hand, and the need of listening to the results - also in a small context - and consequently changing some of them by decisions on the other hand. Both extremes are to avoid: just generating in an algorithmic way, even in an intelligent and changeable way, can result in unmusical structures or "untempered" relations at a scope; too much deviations gets lost of the structure and the large context.

5 Conclusion

Re-Generating Stockhausen's *Studie II* forces one to take to the inner movement and compositorial method of this piece. It reveals meaningful correlations between a compositorial technique and a formalized way of thinking which becomes reality in programming, and it shows important qualities of a non-dogmatic but musical way of working in this domain. In this early piece, Stockhausen has shown two virtues (which he was not famous for later in life): to be modest ("I'm doing a study") and appropriate, and to be short, knowing that stopping before "all has been sayed" can often be more than showing all the material one has generated. This work is a worthy model exactly in this way, and a re-generation can convey a modern formulation for this exemplarity.

6 Acknowledgements

Thanks to Anna Buschart for reading the manuscript, and to the students of the analysis seminar of electronic music at the HMT Hannover in winter 2009/2010 for their patience and interest.

Special thanks to Kathinka Pasveer and the Stockhausen Stiftung for the generousity of not just letting me study the sketches but also giving the permission for printing some of the sketches here.

References

The Re-Generation of Stockhausen's *Studie II* is part of the regular QuteCsound distribution since version 0.4.5. See
http://sourceforge.net/projects/qutecsound
(in the Examples -> Music menu).

It is based of the analysis of Heinz Silberhorn:
Heinz Silberhorn, Die Reihentechnik in Stockhausens Studie II, Rohrdorfer Musikverlag 1980

Using open source music software to teach live electronics in pre-college music education

Hans Roels
University College Ghent - Faculty of Music
Hoogpoort 64
B-9000 Ghent, Belgium
hans.roels@hogent.be

Abstract

A basic course of live electronics is needed in pre-college music education to teach children how to perform on a digital musical instrument. This paper describes the basic components of such a live electronics course, examines whether open source music software is suited to realize these components and finally presents Abunch, a library in Pure Data created by the author, as a solution for the potential educational disadvantages of open source music software.

Keywords

live electronics, music education, Pure Data, digital musical instruments, open source music software

1 Introduction

Since more than a decade home computers and laptops have become powerful enough to process sound in real time. This in fact transformed computers into musical instruments. As this change was taking place, computers became more widely available in households and schools. Anno 2010 the computer has probably become the most widespread musical instrument in a large part of the developed world.

This unique situation urgently prompts us to rethink and redesign our music education which is still largely built upon our traditional knowledge of acoustical instruments and music and to question why pre-college[1] music education in Europe lacks a course of live electronics[2] in which children, teenagers and amateur musicians can learn to perform on a digital musical instrument.

This paper describes the basic components of live electronics courses on a pre-college level, examines whether open source music software -and in specific Pure Data- is suited to realize these components and finally presents Abunch, a library in Pure Data created by the author, as a solution for the potential educational disadvantages of open source music software.

2 Live electronics

2.1 Digital Musical Instruments

It is important to know which fundamental differences exist between live electronic and acoustical music before the content and methodology of a live electronics course can be discussed.

First, in a digital musical instrument (DMI) the sound production unit and the user interface can be seperated and recombined[1]. An acoustical link between both parts as in an acoustical instrument is unexisting and unnecessary. The unique modular nature of a DMI should be central to live electronics education as it is this particular feature that distinguishes it from other traditional instruments.

Secondly, we should ask ourselves what 'performing well' means in a live electronic context as we wish to learn our children and students to play this instrument well. Virtuosity in the digital musical era has an off-stage component which is almost as important as the on-stage one and which is not only different but also more diverse and extensive than in acoustical virtuosity. Developing, building, modifying or adapting a digital instrument is highly important to produce a convincing and expressive performance apart from the classical training.

[1]The term 'pre-college music education' is used to denote all kinds of music education for children, teenagers and grown-ups who haven't taken music courses on a college, university or professional level

[2]Live electronics is used in a broad sense to describe a performance with at least a human performer and an electronic device producing or processing sound

The third difference can be found in the information stream between composers and performers. In electronic music the number of parameters that can be manipulated (before or during a performance) turn a traditional score into a restricted medium to communicate a message between a composer and a performer. Because not all parameters can be notated (in detail) a performer often needs to improvise along more general guidelines. If a performer wants to learn from a composer or from other performers how to perform he has to attend a performance, talk directly to a collegue or composer or consult other media (recordings, texts, source codes, patches, websites, mailing lists, videos,...) than a score. Information has become multimodal in live electronics.

Live electronic music is clearly different from acoustical music, the categories of human activities within music making reflect these changes. The boundaries between composer, improviser, performer and instrument-builder have been blurred and 'performing' has in fact become quite an inadequate term. In general whenever this term is used in live electronics, a larger amount of composition, improvisation and instrument-building is implied than in acoustical music because of the reasons mentioned above. Therefor creativity and autonomy- have become more outspoken in the live electronic music scene.

2.2 Content

Taking into account the specific character of the musicianship and the instrument in live electronics, a basic content of a beginners course for live electronic music is presented. In short the following knowledge domains should be part of this content:

1. Digital Signal Processing techniques
2. Basic audio hardware
3. Mapping techniques
4. History of electronic music
5. Auditory training
6. Sound organisation in real time
7. Performance training

Each of these seven domains is very extensive and could be the subject of a new course. In a beginners course of live electronics these subjects need to be treated only very basically and a selection within each domain has to be made. The seperate implementation in music education of six of these domains is not new and has been researched and applied in classrooms before[2][3]. Introducing mapping techniques on a pre-college level is new and necessary because they are essential to use the full and unique potential of a DMI (see 2.1). The essential parts of these mapping consist of:

1. basic math
2. basic boolean operators
3. comparison operators
4. assignment operator
5. relay switch
6. a module or system to order all this logic and math in time

Because mapping is in fact programming, the components could be a lot more extensive but this selection provides a basic set with a lot of possibilities to connect user interface and DSP in many different ways.

2.3 Methodology

In a live electronics course the sound experience and performance should form the basis of the teaching methodology. Performance requires fast skills and automatisms of the human body. These are in fact feedback loops between the sensoritory information and body gestures. Our brain receives perceptions from our ears, eyes, hands, etc., processes these in a conscious and inconscious way and creates intentions that are embodied in body gestures which adapt and change the sound [4]. This -almost simultaneous- cycle of perception, cognition, intention and movement is action-based and all the separate units are related to the central act of performance. In an action-based methodology the theoretical knowledge can be integrated in listening and performance experiences and foster further progress in sound imagination, experimentation and performance. Music theory and technical knowledge thus are a tool to develop performance -and in general musical- skills.

The forementioned mapping techniques might seem to be boring or very theoretical in a music course, but, if they are integrated in performance and instrument design tasks, they can be fun. One can *hear* whether the result of a mapping logic is right or wrong and consequently recognize a problem and try to solve it... (a method that math or programming teachers would certainly find very attractive).

The multimodal information, the lack of detailed scores and the modular nature of a DMI in live electronics (see 2.1) require a high level of creative input from the children and a need to rely on direct aural information (and not on a score). The teaching methodology in live electronics

should therefore be mainly auditory based and encourage personal autonomy and creativity. For example, as there are no fixed and absolute rules about the right coupling of user interface and DSP, it is very important that children have sufficient time and freedom to experiment with those techniques in order to learn more about the different factors (available equipment, physical skills, artistic demands,...) on which this coupling choice depends. These experiments also help to hear to what extent the user interface and mapping define the audio result, even when the same DSP techniques are used.

To summarize, a live electronics course should center on the immediate perception and performance of sound, use background theory and auditory training to improve and develop this musical activity, promote the tools to take advantage of the modular nature of a DMI and foster the creativity and autonomy of each pupil. Any software to learn live electronics should fit in this general framework.

3 Open Source Music Software in live electronics education

Is open source music software suited to teach and learn live electronics? In the next section Pure Data (Pd) is used as an example to answer this question.

3.1 A continuous learning environment

Anyone who wants to learn to play an instrument should regularly have access to his instrument. This is an almost self-evident truth in instrument training. The simple but very powerful argument in favor of open source music software for live electronics education is its accessibility, low cost and ability to run on several operating systems. These features enable schools and pupils to install and use this software at school and at home. In this way pupils can have regular access to their digital musical instrument and can start developing all the subtile automatisms and gestures that are required to perform music on an instrument. As in acoustical instrument training, the main part can be done at home while the class room only serves to guide this continuous process. In this way open source music software enforces the importance of performance in live electronics.

3.2 The combination of user software and programming language

At the moment there is a wide choice of open source software for live electronic music (Pure Data, ChucK, CSound, SuperCollider, etc.). All these programs are in fact combinations of user software and audio programming languages and some pedagogical problems -especially on a pre-college level- may arise because of this combination.[3]

First, a beginner can easily get lost in the massive and confusing range of possibilities and lose his motivation to learn more about electronic music.

Second, if a newbie wants to find information (articles, websites, mailing lists, books,...) about open source software for live electronics, it is often quite technical and requires a lot of inside knowledge. For a beginner some texts or mailing lists look like cryptograms. Although a lot of patches and example files for beginners do exist in a program like Pure Data, the level is often too high for pre-college teenagers, especially for the ones that have no background in electronic music or software programming.

Third, as I started teaching live electronics with Pd to teenagers in a music school, I noticed that there is another disadvantage to this combination: for a musician who wants to perform or compose it takes too long before he can be creative with the instrument design. He has to know too many basic units and syntax rules before he can produce sound and start combining them.

Of course the combination of user software and programming language has a great pedagogical advantage, especially in a graphical environment like Pd where there is no compiler and no split between the source code and the compiled program. This kind of transparancy helps a tutor to deal with more theoretical knowledge within a performance context and corresponds very well to the action-based methodology of live electronics courses. At the same time this transparancy also helps to abstract or transcend the software that is used in the classroom and to learn more about electronic music and its performance in general. By seeing the basic source code and learning more about fundamental techniques, it becomes easier to

[3]It is no coincidence that most software packages for live electronic music are more or less programming languages. This reflects the fundamental changes in Digital Musical Instruments and in the performance of live electronic music described in 2.1

recognize similar procedures in other software for live electronic music.

Finally as programming languages these forementioned open source music software packages have all the basic tools for learning and applying the mapping techniques. Understanding and using the modular nature of a DMI becomes feasible in a live electronics course using this kind of software.

4 Abunch

4.1 Content

Abunch[4] is a collection of 60 high-level objects (so-called abstractions) in Pd and an additional set of information files. The aim of the main part of the abstractions is to perform electronic music while the remaining abstractions and the info files demonstrate, analyse or explain techniques and musical applications in live electronic music.

The abstractions provide a set of ready made objects to

- record and play sound files (from hard disk and memory)
- manipulate and process sound (effects)
- generate sounds (synthesizers)
- prepare control data (sequencers with different graphical interface)
- synchronize control data (clocks)
- analyse sound and control data (oscilloscope, spectrum analyzer,...)
- record control data to a score
- receive data from common interfaces
- algorithmically generate control data

These Abunch objects use techniques like FM synthesis, granular synthesis and random walk algorithms. The majority of the abstractions were made by the author while approximately one fourth is based on files by other authors.[5] All Abunch objects share a common architecture and can easily be connected with each other (and with native Pd objects) to create all kinds of custom made live electronics. Thus in the first place this library is an active toolbox to experience and learn electronic music.

To start off with performing, Abunch also contains a lot of information and documented patches. Every Abunch object has a help file that explains its workings and that is accessed using the normal help procedure in Pd -right-clicking an object-. Moreover there are more than 40 example files that not only demonstrate the general application rules of Abunch objects but also the internal structure of general audio techniques (FM synthesizer, loopstation device,...) and general musical 'recipes'. The latter uses guidelines and tricks -for the musical application of specific techniques- that have been developed in the last 60 years by composers and performers in electronic music of different styles.

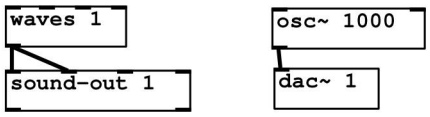
Finally a 'Quick Start' tutorial was made about Abunch and a mapping tutorial. This last tutorial gives some simple examples on how to connect user interface and sound production using the basic operators explained in 2.2.

4.2 Simplified procedures

In the first place the ease of use for beginners was obtained by providing high-level objects as building blocks but the installation and working procedure was also simplified as much as possible.

The installation is straightforward and only requires 1. the Pd core version (called 'Vanilla') which implies that no externals need to be installed and 2. the Abunch folder with files.

If you want to create a new object in Pd, you have to know the name, use a '~' sign to discriminate audio from control objects and provide a set of arguments which refer to specific parameters and functions of that object. In Abunch you can start creating new objects by typing the name (not caring about the '~' in the name) and an unique number as an argument. Thus only one type of argument is used (to enable a general preset system)[6].



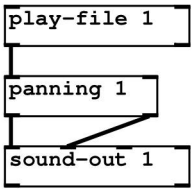
The different procedures in Abunch (left) and Pure Data (right). In Abunch the argument (the number after the object name) refers to the preset system. In Pure Data there are more options: '1000' refers to the frequency of the oscillator 'osc~' object and '1' to one audio hardware output of the digital-to-analog converter (dac~).

[4]Abunch is available for download at www.hansroels.be/abunch.htm and is released under the Creative Commons GNU General Public Licence.

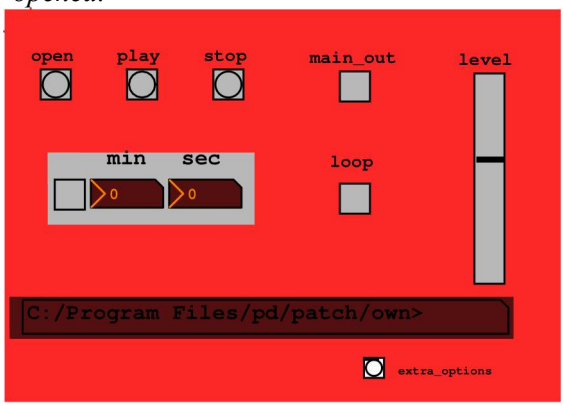
[5]Authors such as Miller Puckette, Frank Barknecht and Tristan Chambers

[6]Giving every Abunch object an unique argument enables to store several instances of the same object in the presets.

Once a new object is created, one can start connecting objects. In Pd objects can receive numbers, audio signals, lists or all kinds of messages for special functions. In Abunch the connection types were reduced to numbers (for control data), audio signals and 2 special connection types: a 'clock' signal to synchronize time related objects and a 'record' connection to combine record and play objects into live recording units.



Three Abunch objects connected to each other, the control window of every object can be closed or opened.



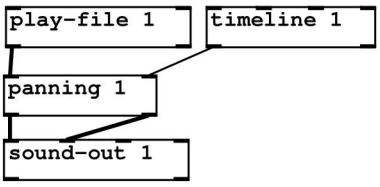
The control window of the 'play-file' object.

4.3 Goal

The main goal of Abunch is to provide a practical open source software tool that enables beginners to learn more about the musical possibilities in real time of a computer. Users can listen to basic tools and techniques and actively learn more about these techniques. In this way Abunch fosters an active teaching and learning method in which sound is the main focus and theory can be integrated in the performance. Pupils can immediately start experimenting with computer sound by combining these objects and techniques with each other to create their own desired sound devices and thus their own set of knowledge. An experimental attitude and a critical, personal opinion and methodology is encouraged by the open-ended architecture of Abunch [5][6]. The analysing objects in the Abunch library enable the students to test and evaluate the other objects and their own built patches and thus help them to take control of their own learning.

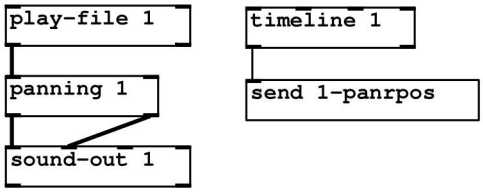
4.4 Advanced features and integration within Pure Data

Abunch was launched in 2008 and tested and used with a large number of children and students. As they got acquainted with the library and its way of working I started noticing that for some of them Abunch was becoming too easy and restricted and a mode to combine user friendliness and more advanced features had to be found. One of the solutions was to hide the more advanced procedures and use the 'wireless' sends and receives in Pd. Almost all graphical user interface (GUI) objects (faders, knobs, switches,...) within Abunch objects can be easily controlled by other objects via the inlets, the usage of the GUI elements can thus be automatized. For ease the control values are normalized to a range of 0 to 127. This also facilitates the use of MIDI hardware to control Abunch objects. The restrictions of this easy MIDI-like procedure can be superseded by using the sends and receives system. Every Abunch object can print out a list of hidden send and receive names to which values within any range can be sent. Via this 'hidden' procedure more parameters can be controlled and adjusted. In a similar way more advanced features were added without changing the easy-to-use layout.[7]



Normal procedure in Abunch: a sequencer object called 'timeline' controls the 'pan' fader within the 'panning' object by connecting it to the right input. This input of the 'panning' object is normalized to the range 0 - 127 just like the output from 'timeline'.

[7]Extra features were also added because Abunch was used by the author in his PhD-research.



Advanced procedure: the range of the values in 'timeline' can be adjusted in the 'extra_options' of this object and are sent to the 'send name' 1-panrpos of the 'pan' fader in the 'panning' object.

In a further stage pupils and students can start using native Pd objects to add more possibilities to Abunch. One part of the example files demonstrates how these native Pd objects can be combined with Abunch objects. The procedures in Abunch are made as similar as possible to Pd procedures to aid the transition from Abunch to Pd. In general Abunch uses a reduced number of procedures, thus the main challenge for the transition is to learn new methods and possibilities. Two specific changes were added to Abunch though (and these were mentioned before in 4.2): the '~' is not used in the name of audio objects and the argument of an object only refers to the general preset system.

4.5 Future plans

Abunch is of course a work in progress with room for improvement, especially because until today it is a solo project and the pace of development is mostly dictated by short-term educational demands.

A frequently heard criticism from pupils is the simple layout. Another problem is the large number of files that is needed to use abstractions and presets within a main file. There is no direct method to bundle all the files in one folder[8] and only a workaround solution was found. This problem is pedagogically relevant because pupils perform and pratice at home and often forget to copy a number of files when they return to the classroom.

Future versions of Abunch will hopefully include the following:

- more example files (about the musical application of techniques)
- more usage of the data structures in Pd to create a more attractive and diverse layout
- a neat and uniform structure within each object with information and comment in the source code
- a style guide for other developers that want to make new Abunch (or similar) objects
- an easy-to-use template for composition and improvisation algorithms

[8]There is no procedure or object in the core version of Pd ('Vanilla') to know the current file name, to copy files and to create a new folder.

5 Conclusion

The modular nature of a Digital Musical Instrument and thus mapping techniques are considered as central parts of any live electronics course in pre-college education. Open source music software for live electronic music is well adapted to teach these mapping techniques and -because of its low cost and accessibility- ensures the tutor and student regular access to their musical instrument which is a prerequisite for any course of instrument training. Therefor it is possible to use open source music software like Pd successfully to teach children to perform live electronic music.

A solution for the massive amount of possibilities and the insufficient user friendliness in an audio programming language like Pd is found in the development of a library of high level objects like Abunch. This library is a balanced mixture of performance and theory-orientated objects with simplified ready-to-use procedures and more advanced hidden features.

References

[1]Miranda, Eduardo Reck, and Marcelo M. Wanderley. *New Digital Musical Instruments: Control And Interaction Beyond the Keyboard*. Pap/Com. A-R Editions, 2006.

[2]Brown, Andrew. *Computers in Music Education: Amplifying Musicality*. Routledge, 2007.

[3]Landy, Leigh. *The ElectroAcoustic Resource Site (EARS).* Journal of Music, Technology and Education, no. 1 (November 2007): 69-81.

[4]Leman, Marc. *Embodied Music Cognition and Mediation Technology*. 1st ed. MIT Press, 2007.

[5]Holland, Simon. *Artificial Intelligence in Music Education: A Critical Review*. In Readings in Music and Artificial Intelligence, Miranda, Eduardo Reck, ed. Routledge, 2000. .

[6]Smith, Brian. *Artificial Intelligence and Music Education.* In Readings in Music and Artificial

Intelligence, Miranda, Eduardo Reck, ed. Routledge, 2000.

Field Report: A pop production in Ambisonics

LAC 2010, Utrecht

Jörn NETTINGSMEIER
Freelance audio engineer and qualified event technician
Lortzingstr. 11
Essen, Germany, 45128
nettings@stackingdwarves.net

Abstract

This paper describes an acoustic pop production in mixed-order Ambisonics, using an ambient sound field recording augmented with spot microphones panned in third order.

After a brief introduction to the hard- and software toolchain, a number of miking and blending techniques will be discussed, geared towards the capturing (or faking of) natural ambience and good imaging. I will then describe some peculiarities of Ambisonic mixing and the struggle to make the resulting mix loud enough for commercial use while retaining a natural and pleasant sound stage and as much dynamics as possible.

Keywords

Ambisonics, surround sound, pop music production, mixing techniques

Introduction

In Feb 2010, German singer/songwriter Tom Gavron scheduled a recording session featuring three different line-ups: a quartet featuring piano, violin, cello and percussion, a duo with piano and bassoon, and a jazz sextet. Overdubs were to be limited to the vocal tracks, to capture a natural group feel and allow for improvised interaction.

With so many interesting acoustic instruments, it became clear that their spatial characteristics and interaction with the room ambience had to be captured, rather than relying on panned mono sources as usual.

Although I had no idea of the recording venue acoustics, I decided to try an Ambisonic approach, using a tetrahedral main microphone backed up with a standard close-miking setup.

Since the recording was to serve both as promotional material and merchandise, it was clear that an easily accessible and distributable stereo mixdown was the primary target. In addition, we planned to create a 5.1 mix with an "on-stage" perspective as an extra feature for fans with the necessary equipment, plus a native B-format release for the limited number of Ambisonics enthusiasts out there.

Equipment

The session was recorded on a Lenovo ThinkPad X61 running a heavily customized openSUSE 11.1 with kernel 2.6.31.12-rt20 and current SVN heads of JACK (r3898), FFADO (r1794) and Ardour2 (r6635).

The audio data was written to an external USB 2.0 drive[1] and backed up to a second harddisk every night.

A Focusrite Saffire Pro 26 served as the recording interface. Its eight microphone preamps were complemented with another eight from an RME Micstasy, and eight cheap Behringer ADA 8000 channels for line signals and less important microphones. All units were connected via ADAT and externally synced to the Saffire's wordclock out.

[1] I found external disks to be more dependable than built-in notebook drives, since they have less tendency to overheat, deliver better sustained write rates and generally run more quietly. As an additional bonus, they can help circumvent shared-interrupt problems with the built-in controller and other important parts of the signal chain (as was necessary here since the SATA chip shares an IRQ with the cardbus controller).

A Core Sound TetraMic (see appendix) was used as the main microphone.

On the software side, Ardour2 [Dav10] was used both for tracking and mixing.[2]

Recording

Using a "main microphone" approach in pop music may seem strange, and it does have a few pitfalls. The soundstage is more or less determined during setup. While you can drag instruments away from their natural location with the spot mikes, you will risk incongruent directional cues if you overdo it. From early on, you have to get your client to cooperate and refrain from fancy panning suggestions in post-production. The prize will be a beautiful, natural ambience.

1 Quartet session

The first two studio days were dedicated to the violin/cello/percussion/piano line-up, and were recorded in a rather small room (about 4x7m) that was acoustically treated as a percussionist's rehearsal room, i.e. very dry but pleasant.

With no separate control room available, the recording gear (and the engineer) ended up in the same room, which greatly eased the communication with the performers. The mike setup however was rather tedious, since several test recordings of every instrument were required to be able to judge the recorded sound accurately.

The four musicians were arranged in a circle, facing each other, and the TetraMic was placed in the centre, slightly favoring the strings to ensure a good natural balance.

Spot microphones were used as follows (from rear left to rear right):

On the **pedal timpano**, a trusty old AKG D-112 about 10cms away from the head, close to the rim.

The **tom-tom** was handled by a Sennheiser e904.

A cheap Sennheiser vocal mike that we had lying around was put on the **snare** after it was found that a Røde NT5 could not cope with the sound pressure up close, even with the comparably light touch of a classical drummer.

[2]Wiring Ardour for Ambisonics is outside the scope of this paper. See [Net09] and [Net09-2] for a detailed explanation.

Illustration 1: Quartet session. Interesting mikes, top to bottom, left to right: coincident pair of AKG CK91s for drum set; Core Sound TetraMic as main microphone, offset towards the strings; AKG CK91 violin spot; BPM CR-95 cardioid cello spot

The entire set (which also included a number of splash and ride cymbals and a hi-hat) was covered with an **overhead** XY pair of AKG CK-91 cardioids at a height of about 2 metres, pointing almost straight down.

The **electric piano** (a Korg StageVintage) has very nice balanced outputs which were used to capture the direct signal. Its stereo jack outs were routed to two active RCF cabinets placed behind the pianist on the floor, tilted upwards like a monitor wedge.

This way, the electric instrument blended nicely with the room and the acoustic instruments, and the TetraMic could capture some meaningful directional information.

For the **violin**, we chose another AKG CK-91 and placed it 20-30 cm above the upper part of the fingerboard, pointing at the bridge.

The **cello** was covered with a BPM CR-95 switchable large dual-diaphragm in cardioid setting, 30cm away from an f-hole slightly below the bridge. It was shielded from the drums with some jackets draped over a music stand (not shown in photo), to reduce crosstalk from the snare drum.

All musicians were facing each other, and the nulls of all microphones were kept pointing inwards as much as possible, to improve channel separation.

For the entire duration of the session, all tracks were kept armed and recording, with short breaks

every hour or so, to save clean snapshots while the transport was stopped.[3]

Since the material to be recorded was new to the musicians, the arrangements evolved considerably during the session. It also meant that preciously few compatible parts were available for editing.

In the end, most of the recordings were entire takes (usually between 13 and 20 per day), the last few of which where usually selected for post-production with minor edits to be made. Some solo parts were recorded as separate takes to reduce stress and fatigue, but always "live", i.e. with the entire band.

A click was used to ensure a consistentent base tempo before each take, but the recordings themselves were done without.

The vocals were to be overdubbed on a separate date, allowing time for careful selection of the final material and preliminary editing.

2 Piano/bassoon session

The room was 5 by 6 metres with a height of around 3 metres, and acoustically treated.

We were lucky to be able to use a Steinway **grand piano**, which was recorded with an ORTF pair of AKG CK-91s. The lid was propped up in low position, and the mikes were located to the rear of the instrument, slightly above lid level, to avoid the boomy quality of the direct sound coming through the opening.

Another CK-91 covered the **bassoon**. During warm-up, I learned that the sound of a bassoon travels along the entire length of the instrument, depending on register: the low notes originate from the top of the bell (the upper part), whereas the high notes emerge from the boot (the lower part), with many positions in-between. To capture this interesting quality, the spot mike was augmented to an M/S pair with the BPM CR-95 in figure-of-eight setting.

Additionally, a Røde NT-5 was aimed at the bell from above, to have some additional wind noise and more pronounced overtones for flexibility in the mix.

All instruments and microphones were lined up along a common axis, the microphones pointing outward to reduce crosstalk. The main microphone was placed well outside the center of the room at a height of about 2m, so that the instruments subtended an angle of around 120 degrees. The resulting hole in the soundstage was reserved for the vocal overdub.

3 Sextet session

The jazz sextet consisted of a standard jazz drum kit, double bass, piano/keyboard, guitar, trumpet/flugelhorn and vocals. Again, the session was captured with a TetraMic in the centre. This time, the spots were used rock'n'roll fashion, i.e. extremely close, to get some channel separation.

The **drums** were covered with three Beyer clip-ons for snare and toms, an AKG D-112 for the kick, and two AKG CK-91 in XY configuration for overheads.

The **double bass** was recorded to two tracks: a DI signal from a piezo pickup and the BPM CR-95 in cardioid setting, positioned very close to an f-hole.

Regrettably, no guitar amp was available for the session, so the **guitarist** plugged his ES-175 and a Telecaster into a Lexicon MPX-100 which was driving a small active RCF P.A. cabinet miked with a good ol' SM57. Needless to say, the attack was shoddy.

The Steinway **grand** had to be kept closed so as not to upset the natural balance between the instruments in the room. Hence, the mike (an ORTF pair made of Røde NT5s) had to be placed at the only available opening, about 30cms above the tuning pegs. The coverage of the instrument in the extreme registers was not too good, but for the reduced jazzy playing style, which mostly featured the middle register, the compromise was acceptable.

The **trumpet and flügelhorn** were captured with another CK-91 with 10dB pad, aimed well above the bell. The player was instructed to lift the instrument slightly for emphasis, so that extra brilliance would be captured whenever he felt necessary.

The **vocalist** used a hand-held Beyer MCE 91 for a live take of Sinatra classic "Come fly with me".

The second piece to be recorded was an instrumental rendition of "My funny valentine", for which the vocals were to be overdubbed later.

As expected, snare and cymbals leaked into most microphones. For a few dBs of crosstalk reduction (which can make a world of difference in the mix), an empty gear case was used as a barrier between

[3] This precaution turned out to be unnecessary, since Ardour saved reliably even while transport was rolling.

piano mike and drums, and a couple of winter coats hung over a mike stand served as a shield for the bass mike.

4 Vocal overdubs

The vocal dubs were done two days after the main session, in a 4x5m living room with wooden flooring and one wall deadened with a large mattress.

We used the BPM CR-95 in cardioid setting, with a windscreen in front, positioned slightly below mouth height to ensure a relaxed singing posture and avoid the tendency to lift the chin while singing.

About half a metre behind the main mike, the TetraMic was running along for some optional room ambience.

The singer was fed a mixture of the main mike and a virtual Blumlein array from the TetraMic. It was pointing upwards, away from the direct sound as much as possible, to avoid coloration. I find that a good room signal helps reduce the listening fatigue induced by closed studio headphones. Since it feels natural even at low levels, it does not affect intonation as much as artificial reverb, which usually has to be turned way up to for a comfortable listening experience.

A stereo fold-down of the TetraMic recording of the basic tracks was used as the primary monitor signal, complemented with some dry piano for pitch reference, and additional direct signals as required by the vocalist.

The overdub takes consisted of one four-channel track for ambience and one mono track.[4] They were recorded on top of one another during the session.

To sort the material, we created four new pairs of tracks, to which the recorded takes were moved after listening: one for trashcan, one "maybe usable", one "satisfactory" and one for material deemed very good. Ardour's "Lock edit" mode proved very helpful, as it eliminates time alignment errors when dragging material between tracks. Good takes with questionable parts in them were split to exclude the problematic section.

[4]In retrospect, it would have been better to record the vocal takes to a single five-channel track each, to avoid confusion in the editing stage. Such a compound track is easily split into manageable parts using Ardour's flexible buses.

5 Post-production

5.1 Editing

Edits were done in the recording configuration, using a standard stereo master bus, and monitored through stereo speakers. Each edit was then cross-checked with headphones.

As expected, the main microphone technique and associated cross-talk made convincing edits very difficult, and the assembly process was time-consuming. After some experimenting, it was found that staggering the cuts of main mike and spots helped hiding otherwise questionable edits: minor problems with overhanging sounds and long crossfades in the ambience would become acceptable if a clean note onset had been established in a spot mike before.

Again, the “slide edit” and “lock edit” modes of Ardour were used heavily. “Slide” allows regions to be dragged around in time, and is needed to align an insert with the groove. When that has been done, “lock” fixes all regions in time and only permits the trimming of region boundaries – that way, the edit can be cleaned and made inaudible, without the danger of messing with the groove unintentionally (which happens rather easily with Ardour when screen space is limited and track heights are small).

6 Mixing

For mixdown, a new 16-channel summing bus was added for third-order Ambisonic mixdown, and the old "master" bus was deleted. Additional two-, four-, and nine-channel monitoring busses were created for monitoring in UHJ-encoded stereo, first and second order Ambisonics.

6.1 Using convolution reverb

Spot mikes need some additional reverb to sound natural at the listening spot defined by the position of the main microphone. Ideally, this reverb is an impulse response of the recording room recorded by the main mike, where the excitation speaker is placed at the location of each spot microphone. For maximum fidelity, separate IRs should be captured for each instrument group (or even every microphone position).[5]

[5]Aliki [Adr09] is a good tool for the job. The capturing of room responses is described in detail in the Aliki manual.

These IRs can then be combined into a convolution matrix for an engine such as jconvolver[6], so that it has N inputs, one for each of the IRs, and either two or four outputs, depending on whether the room response was recorded in stereo or first-order B-format.

Optionally, the early reflections and tail section of an IR can be separated. One tail can then be used globally (because the tail does not contain significant directional cues and differences from one position to another are very subtle at best), and only the short early reflection parts are treated individually. This conserves CPU and allows for an extra degree of flexibility, namely the ratio of early reflections to reverb tail.

To plug such a beast into ardour, create an N-channel bus with a corresponding N channel insert connected to the external jconvolver. Only the first few return channels will be used, and the rest can be left unconnected. Similarly, only the four active outs of the N-channel bus will be connected to the first four channels of the master bus which contain the zeroth- and first-order components.

Unfortunately, no adequate speaker for an IR measurement of the recording rooms was available, so some "foreign" IRs had to be used on the spot mikes to blend them with the slightly wetter room signal.

6.2 Source alignment

The natural sound stage as recorded by the TetraMic was used as the basis for source positioning. With stereo in mind, the sound field was rotated to provide a not-too-unconventional balance when folded down, and to leave space for the singer in the front. Where possible, strings were placed in the back, since they benefit from the slightly phasy, blurry quality which UHJ stereo encoding adds to rear sources. For the same reasons, bass instruments should be placed in the front quadrant if possible.

Spot mikes were brought up one by one and aligned with the Tetramic sound stage by ear, moving them until the sources stopped "jumping" when switched.

[6][Adr10]; jconvolver is unique among the freely available JACK convolvers in that it uses a variable partition size and can be configured to incur only one period of latency regardless of IR length.

Each spot mike was delayed to compensate for the offset to the main microphone, to avoid comb filtering effects in the combined signal. Sometimes, the actual delay used differed from the measured value by several milliseconds, if a more pleasant timbre could be obtained.

6.3 Equalisation

As shown in the photo, the recording situation was rather cramped, resulting in a pronounced bass boost in the microphone due to proximity effect. When played back, the sonic impression was quite obtrusive, although technically correct. Some bass reduction in combination with gentle reverb was employed to give the mix a more spacious feel and make the instruments back away from the listener.

Signal crosstalk was quite bad, and steep high-pass filters had to be used on almost every microphone to keep the timpani and bass drum out.

The extreme close-miking of the strings produced a slightly harsh tone that proved difficult to correct without losing the "shimmer" of the bow sound. In the end, reverb was more effective than filters.

During post-production, it was found that an EQ setting that works for the UHJ-encoded signal will also sound good in Ambisonics, but not vice-versa. The full Ambisonic rendering is a lot more spacious and transparent, and can absorb more reverb. One must resist the urge to "fatten it up" too much, as this will result in a boomy and overly wet UHJ stereo image.

Frequently, instruments which had been very pleasant when soloed sounded tinny or otherwise artifical in the mix. This is a common phenomenon when many microphones are open, and there is no way around it other than to paint the spot mike sound "with the large brush" (i.e. to over-exaggerate the desired characteristics a little), to close unused microphones wherever possible, and to keep fiddling with the delays. In retrospect, hypercardioid patterns wounld have helped to reduce crosstalk between adjacent instruments.

6.4 Building the mix

My usual approach is to start with the drum overheads and any room microphones, add spots one by one, assemble a basic track, and bring the vocals in at the end. Since the Ambi setup reacts very differently than a standard stereo system, I found that I had trouble finding room for the vocals this way and gave up after a few failed

attempts. Instead, I started with the vocals and a little bit of piano, making sure the song would work as-is. The other instruments were then tucked "under" this basic mix. Afterwards, the room microphone was brought up for some "air". Additional reverb was added to each signal individually, and finally, fader automation was used to emphasize dynamics, clean up the mix and add some final polish.

6.5 Dynamics

In the final stages of the ambisonic mixdown, it became apparent that commercial impact would require some sort of peak limiting and gentle overall compression. Regrettably, no free multichannel-capable compression tools are available at this point[7] (and Ardour2 cannot make use of a plugin's side chain port easily), so the master was left unprocessed and the individual channels were treated instead.

This is a serious drawback, since it entangles the mixing and mastering stages. Keeping them separate (and bringing in a fresh pair of ears) has its advantages: the mixing engineer does not have to deal with real-world playback systems and can create an artistic mix under optimum circumstances, and the mastering engineer can then deal with the necessary compromises to make it work on Joe Sixpack's car stereo. It takes some effort to focus on mastering processing and not constantly question and revisit earlier mixing decisions.

On the up side, the achieved mix was a lot more transparent than with sum compression, while the loudness was slightly lower than average. To compensate, some additional limiting was performed on the UHJ-encoded stereo output.

The automatic 5.1 folddown was done with Fons Adriaensen's hand-optimized second-order ITU decoder that comes with AmbDec as a default preset.

[7] JAMin [Har05] or the very promising Calf multiband compressor [Fol10] come to mind. As it stands, even something as kludgy as a compile-time switch to allow for multichannel work with hardwired side-chaining would be highly welcome.

7 Other production approaches

Recording more-or-less "live", without click, is nice but not always practical. Far more material needs to be discarded for mistakes, there are less options for repair and improvement, and the overall level of perfection that can be achieved is limited unless the musicians are of the very best.

But you can easily use traditional single-instrument overdubbing in an Ambisonic production. Naturral room ambience (if desired at all) can either be faked using B-Format impulse responses as described earlier, or you can keep the main microphone running each time a musician lays down a track. In theory, the result will be the same as if everybody had played in the room at the same time. In practice, you will also get a lot more hiss. But to blend one or two soloists into the mix, this approach is feasible.

Conclusion

Doing a pop production in Ambisonics is definitely possible. One must constantly double-check the mix in both UHJ and Ambisonic renderings, but that is a fair deal compared to the hassle of an extra surround mixing session. With the available free software tools, most recording problems can be dealt with, and Ambisonic panning opens up new creative possibilities. Even on plain stereo systems, the sound stage can be extended well outside the usual stereo triangle, without complicated manual phase trickery.

A flexible multichannel compression tool with appropriate side-chaining would help get the job done more quickly. However, with a periphonic sound stage encompassing the entire sphere, sum compression is even more questionable than for stereo (where likewise the current best practice backs away from global dynamic processing and moves towards stem-based separation mastering).

It will be interesting to take this production approach to other genres, such as electronic dance music (where a slim chance of native ambisonic playback might exist in some clubs).

Appendix: The TetraMic

An implementation of a microphone design devised by Gerzon, Craven et al. in the 1970s, the TetraMic consists of four cardioid capsules arranged in the edges of a tetrahedron. Its native signal set (called A-format) can be converted into

the B-format used in Ambisonics by a simple matrix operation:

```
W' =  LFU + RFD + LBD + RBU
X' =  LFU + RFD - LBD - RBU
Y' =  LFU - RFD + LBD - RBU
Z' =  LFU - RFD - LBD + RBU
```

(where L/R means left/right, F/B is front/back, and U/D is up/down, to uniquely identify each capsule)

The signals are primed to indicate that some EQ correction is still missing to compensate for the slight positional error of the capsules (for a perfect microphone, they should be precisely coincident). For an easy-to-understand discussion of A-to-B format conversion, see [Far06].

On Linux systems, the conversion is handled by TetraProc [Adr09-2], whose author will provide a custom configuration file in cooperation with the microphone manufacturer.

The B-format can then be used natively for Ambisonic playback (either horizonal-only or full 3D), or an arbitrary number of first-order microphone patterns can be derived from it. In practice, one would create a coincident stereo pair, or a set of five (hyper-)cardioids for Dolby Surround. The big advantage is that orientation, opening angle(s) and polar characteristics can be selected during post-production, making it a very versatile main microphone.

In terms of localisation precision, the Tetramic is one of the best microphones of its design available today, owing to its small capsules which make the array nearly coincident to begin with, and the theoretically perfect digital post-matrix filtering. Its one great disadvantage is the low signal-to-noise ratio (a consequence of the small, cheap capsules), which makes it less well suited to very soft music such as a single acoustic guitar at a distance. For the task at hand, however, it was ideal.

The Tetramic is available from Core Sound LLC, http://core-sound.com. Quieter (and more costly) variants of the design are offered by SoundField, http://soundfield.com.

References

[Adr09] Fons Adriaensen, Aliki, http://www.kokkinizita.net/linuxaudio/downloads/

[Adr09-2] Fons Adriaensen, TetraProc tetrahedral microphone processor, l.c.

[Adr10] Fons Adriaensen, jconvolver, l.c.

[Dav10] Paul Davis et al., Ardour digital audio workstation, http://ardour.org

[Far06] Angelo Farina, A-format to B-format conversion, http://pcfarina.eng.unipr.it/Public/B-format/A2B-conversion/A2B.htm

[Fol10] Krzysztof Foltman, Thor Harald Johansen et al, Calf audio plugin pack, multiband compressor contributed by Markus Schmidt, http://calf.sourceforge.net

[Har05] Steve Harris et al., JAMin, the JACK Audio Mastering interface, http://jamin.sourceforge.net

[Net09] Jörn Nettingsmeier, Ardour and Ambisonics, in: eContact! 11.3 (Journal of the Canadian Electroacoustic Community), http://cec.concordia.ca/econtact/11_3/nettingsmeier_ambisonics.html

[Net09-2] Jörn Nettingsmeier, Using Ambisonics as production format, in: Beta Ardour reference manual, http://ardour.stackingdwarves.net/ardour-en/8-ARDOUR/279-ARDOUR/280-ARDOUR.html

5 years of using SuperCollider in real-time interactive performances and installations - retrospective analysis of *Schwelle*, *Chronotopia* and *Semblance*.

Marije A.J. BAALMAN
Design and Computation Arts
Concordia University
Montréal, Québec
Canada,
marije@nescivi.nl

Abstract

Collaborative, interactive performances and installations are a challenging coding environment. *SuperCollider* is an especially flexible audio programming language suitable to use in this context; and in this paper I will reflect on 5 years of working with this language in three professional projects, involving dance and interactive environments. I will discuss the needs and context of each project, common problems encountered and the solutions as I have implemented them for each project, as well as the resulting tools that have been published online.

Keywords

SuperCollider, interactive performance, sensing, composition systems, live coding, tool/code development

1 Introduction

Between 2005 and 2010 I have been involved in three real-time interactive projects, in which *SuperCollider* (SC3) was the central core to deal with realtime sensor data, audio analysis, interaction with other programs for data exchange and show control, and sound, vibration and light output. This paper gives an overview of the techniques used within *SC3* and provides a critical analysis of the problems encountered along the way and solutions provided. The work on these three projects has resulted in a number of tools that have been made available to the general public as open source software, and that aid other artists in the realisation of their projects.

The three projects are all collaborations with artist/researcher Christopher Salter[1] and various other artists. Two of the projects are dance performances, one of them is a one-person experiential installation. Furthermore all three projects have an interactive or responsive component based on sensor inputs and dynamic mapping of these inputs to output media. They are also all situated in a collaborative context, where there are several artists collaborating using different output media, as well as different programming environments with which data is to be exchanged. As the development of these three projects has been sequential and taking place over the course of 5 years, certain approaches and methods using *SC3* have emerged, as well as a re-evaluation of methods used in earlier projects. In all of these projects, I have encountered similar problems, however, as my proficiency at *SC3* developed, I have discovered different solutions. In some cases because I found that the new project asked for a different approach, based on the specifics of the problems, or because I wasn't content with the solution used previously.

While working on artistic projects there is always a trade-off between developing "general-purpose" tools that are robust and flexible in use, and quickly putting something together, that is usable and reliable for the project at hand, but may not translate well to other projects. This paper reviews the methods I have used over the years, and identifies the components that could be, or have been, adapted to more general purpose tools for use in future projects.

For those readers unfamiliar with *SuperCollider*, I have added a section detailing some general information regarding *SC3* at the end of this paper. Class names within *SC3* will be put in bold in the following text.

2 Coding in the context of interactive performance

Coding in a professional performance context has different demands than product oriented coding, in the sense that while writing the code, the purpose of the code and its needed functionality is not yet known, but will emerge during the artistic process of discussions, experimentation and rehearsals. This is especially true,

[1] `http://www.chrissalter.com`

when the artistic project involves real-time sensing, where it is not known beforehand what the input data will be, and how it will influence the output media, which are also being shaped in the process of creation.

Within the rehearsal process for all of these projects it is important to have a flexible system which allows for on-the-fly manipulation of audio synthesis processes as well as sensor data mappings. Part of the preparation for the rehearsal process is to create systems that allow for such flexibility, so that many different kinds of interactions can be explored. This is only possible if it is clear in advance what kind of possibilities there are, i.e. what kind of data is to be expected from the sensors, the type of audio processes that will be used (its compositional structure, as well as its sonic quality), and what kind of interactions the collaborators in the project are interested in. Extensive discussions about this with the other collaborators, as well as short exploratory sessions with the performers, and a basic understanding of some of the movement material of the dancers (so that you can e.g. wear an accelerometer and produce some data yourself while writing and testing code) are essential components in this process. Having some skill at livecoding to quickly develop new interactive processes is also vital for a succesfull rehearsal process.

For the eventual showtime in the theater or at an exhibition, it is important to have a robust "show control" system[2] from which the show can be run, while at the same time being flexible to adapt to differences in setup (e.g. audio balance/mix), based on the venue in which the performance takes place. Ideally, you should be able to adapt "cues" during the show, should there be the need. Backup solutions, in case sensing infrastructure breaks down, can also be useful (even just as a reassurance).

In the case of installations, the code (and the machine it runs on) may need to be prepared to be started and stopped by gallery personnel who have no knowledge at all about coding, and in some cases even of computer environments. In the ideal case the machine running the code can boot up and start the code automatically, so the computer only needs to be turned on.

[2]In theater/performance the collective control for all events supporting the performer's action on stage (i.e. sonic, light, mechatronics, video) happening on stage is usually referred to as "show control".

3 The artistic projects

3.1 Schwelle

Schwelle is a theatrical performance that takes place between a solo dancer/actor (Michael Schumacher) and a "sensate room". The exerted force of the performer's movement and changing ambient data such as light and sound are captured by wireless sensors located on both the body of a performer as well as within the theater space. The continuously generated data from both the performer and environment is then used to influence an *adaptive audio scenography*, a dynamically evolving sound design that creates the dramatic impression of a living, breathing room for a spectator. We aimed at creating an auditory environment whose sonic behaviour is determined continuously over different time scales, depending on the current input, past input and the internal state of the system generated by performer and environment in partnership with one another.

The data from the sensors is statistically analyzed so the system reacts to changes in the environment, rather than absolute values. The statistical data is then scaled dynamically, before being fed into a dynamical system inspired from J.F. Herbart's theory on the strength of ideas [Herbart, 1969]. The dynamic scaling ensures that when there is little change in the sensor data, the system is more sensitive to it. The output of the Herbart systems is mapped to the density, as well as to the amplitude of various sounds that comprise a "room" compositional structure of more than 16 different layers. An overview of the data flow is given in figure 1.

The mapping, which is indicated between the dynamic scaling and the Herbart Groups, is a matrix which determines the degree to which each sensor influences which sound [Baalman et al., 2007]. Additionally, there is a "state" system, defining different parameter spaces within which the soundscape can move. The system moves between these states, depending on long term development of the input data. The theatrical light also has distinct behaviours based on the state of the room. The information about the current state is transferred to the computer controlling the lights via OpenSoundControl (OSC)[3] [Wright et al., 2003].

The diagram also shows that there is a second data flow path, which constitutes a more classical, instrumental approach of using the

[3]`http://www.opensoundcontrol.org`

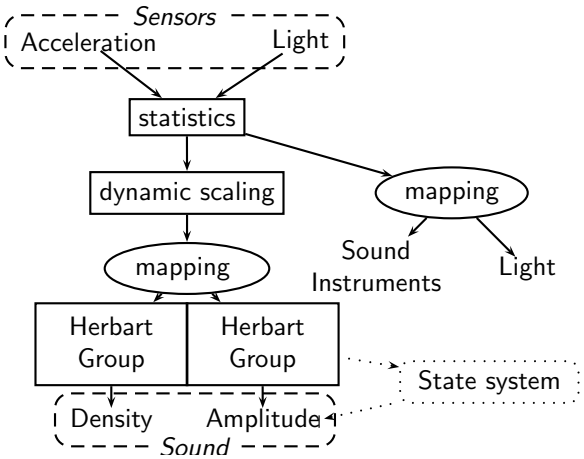


Figure 1: Data flow diagram

sensor data. The mapping between a movement created within the room by the performer or objects in the room and a resulting lighting or sonic event is direct, and recognizable as an action-reaction interaction. This interaction has been carefully tuned to certain dramatic sequences within the piece, and is easily switched on and off, depending on which scene is taking place. In some scenes I change the strength of the interaction at suitable points in the dance, thus creating a duet with the dancer.

Apart from the system mentioned above, there was also a backup system in case the wireless transmission of data from the dancer would break down, so in case of need I could mimic the data using the joysticks of a gamepad device.

The sound was spatialised across the performance space using several methods of amplitude panning between speakers. Additionally, there was an elaborate system for submixing the different audio layers, before sending them to the outputs. In this submix system there were controls, both for direct setting of volumes, and for dynamic volume control mapped to the dynamical system output data.

3.2 Chronotopia

Chronotopia is a dance performance by the Bangalore (India) based Attakkalari Centre for Movement, in collaboration with visual artist Chris Ziegler; the music score is composed and performed by Matthias Duplessy. For this performance we created a responsive light installation: a 6 by 6 matrix of cold cathode fluorescent lights (CCFL), and 3 handheld lights. We controlled these lights with wireless technology, using a combination of an Arduino board and

Figure 2: A snapshot of the view on both the stage and my computer screen visualising the light matrix.

an XBee wireless chip.

For the control of the lights, I used **Synth**s on the server sending their output to control rate busses, which I then polled at a regular interval in order to send it over a serial protocol to the XBee network from within the langauge. This approach allowed me to make use of the various envelope curves that are available within **SynthDef**s, and use the extensive **Pattern** library for sequencing of these **Synth**s. In the prototyping phase for this project, I extensively used *scgraph*[4], which allowed me to model the light matrix and see its behaviour on screen. During the setup and performances in the theater, it allowed me to monitor the behaviours of the system and compare it to the actual output on the stage (see figure 2).

Additionally, I used camera based motion tracking data (from a camera looking down at the stage), to map to the light matrix, as well as beat and pitch tracking data extracted in realtime from the soundtrack. We also exchanged data between the light control and the interactive video, both for synchronisation of cues with the soundtrack (using frametime of the playback), and for connecting the intensity of the lights to the video image (maximum output value of all the lights was used to control the brightness of the video image in specific scenes).

3.3 Semblance

JND/Semblance is an interactive installation that explores the phenomenon of cross modal perception — the ways in which one sense impression affects our perception of another sense. The installation comprises a modular, portable environment, which is outfitted with devices

[4] `http://scgraph.sourceforge.net`

that produce subtle levels of tactile, auditory and visual feedback for the visitors, including a floor of vibrotactile actuators that participants lie on, peripheral levels of light and audio sources, which generate frequencies on the thresholds of seeing, hearing and (tactile) feeling.

In this installation we use wireless sensing devices to gather data from floor pressure sensors. The loudspeaker setup consists of 12 special speakers designed to enhance home theater sound setups with tactile vibrations. These speakers are laid out in a grid of 2 by 6 underneath a platform, or bed, on which the visitor lies down. With the pressure sensors we can detect micro-movements of the body. The light appearing above the visitor is controlled with DMX (see §5.5) from Max/MSP. OSC communication is used to send the desired cues to Max/MSP.

The synthesis of vibrational output was a difficult process. While sound synthesis methods can be used, it is quite different to find vibrations that work — artistically — on a tactile level. While with sound you can sit behind the computer, and code synthesis processes and tweak them while listening to them, to lie down on a vibrational floor and code at the same time is unpractical. Furthermore, it is a medium with which neither of us had any previous experience to draw upon. It was also hard to create vibrations without an acoustical counterpart so that we had to find vibrations that were also sonically interesting.

The sensor data was analysed statistically in realtime so that changes in pressure, rather than the absolute pressure, was used to map to the synthesis processes. Since there were 24 areas of sensing, in a 6 by 4 grid, and 12 speaker outputs, some combining of sensor data was done to map local movements of the body to local vibrations. In certain parts, we used a sum of all the changes in pressure to map to an overall amplitude of the vibration. Furthermore, in one part we did amplitude tracking on the vibration output, and used that to determine the maximum level of brightness of the light.

For spatialisation we employed various methods of panning (in one or more directions), or outputting to one or more speakers at the same time, with either the signals in phase to all speakers, or with a randomized phase.

The composition is made up of three movements and lasts about 13 minutes. Within each movements there are a couple of different parts, with varying output combinations, and various mappings to the sensor data.

4 Common techniques

4.1 Collecting sensor data

The collection and processing of sensor data is an essential part of working on interactive performances. The first step is interfacing with the hardware that actually does the collection of data. In *Schwelle* we used Create USB interfaces[5], which show up as HID devices to the operating systems. In *Chronotopia* motion tracking data is used, which is received from another program (see §5.4) through OSC. In *JND/Semblance* we use wireless XBee based sensors and the data communication takes place over a serial port[6].

For *Schwelle* I made an abstraction between a class named **SchwelleSensor** and a class interfacing with the actual HID device (two classes, one for Linux, one for OSX). I then had alternate versions of the **SchwelleSensor** class which were using backends like the WiiMote and a "mix" of several other sensors. In later projects this abstraction was generalized, instead using a general purpose data framework (the SenseWorld DataNetwork) into which any kind of device can input data and further use of the data is agnostic of the way in which the data was initially gathered [Baalman et al., 2009].

4.2 Processing sensor data

In the class **SchwelleSensor**, I also stored data calculated from statistics of the data, which was performed in the class **SensorData**. These calculations all took place in *sclang*. In the later projects I moved the statistical processing to *scsynth*, taking advantage of the efficiency of the DSP algorithms implemented in the UGens. The resulting, "derived" or "cooked" data was made available on the DataNetwork — a central hub for all the control data. This approach makes it flexible to switch between using the direct data and a derived version by simply changing the data source.

4.3 Mapping sensor data

Mapping of the sensor data involves not only remapping the value ranges between the input data and output parameters, but also the merging of data streams, extracting features from

[5] `http://www.create.ucsb.edu/~dano/CUI/` and `http://overtone-labs.com`

[6] See `http://sensestage.hexagram.ca`

data streams, and creating dynamical processes which develop compelling behaviours based on realtime sensor inputs.

Schwelle was by far the most complex system of interaction as described above and shown in figure 1. The interactions between the different stages in the dataflow path is organised with the class **SchwelleSensorSystem**, which reads the input data, calculates the dynamical scaling (in **DynamicScaleSystem**) and maps it to input data for the dynamical system, implemented in the class **SchwelleHerbart**. All the processing of the data is taking place inside *sclang*, and in custom classes with a lot of interconnection between these classes.

In *Chronotopia* and *JND/Semblance* the data processing is centered around data processing units that are integrated with the SenseWorld DataNetwork framework. This approach makes the creation and alteration of dataflows much more flexible than the approach taken in *Schwelle*. However, some of the data processing algorithms used in *Schwelle* still have to be ported as units to the general framework.

For *JND/Semblance*, I started developing a framework for creating presets, combining set parameters for use with specific **Synth**s as well as mapping of parameters to specific data streams taken from the DataNetwork. These presets can then be stored to disk and recalled in future sessions, and instantiated and tweaked in realtime, both through a graphical interface and through code.

4.4 Data exchange with other programs

In each of the three projects, one of the collaborators was using the software *Max/MSP* to control theatrical lighting. In order to exchange data we had to set up OSC-communication protocols to exchange data. While in *Schwelle* this was done on an ad-hoc basis, defining a OSC address pattern each time we needed to exchange some data, for the later projects we developed a general purpose framework, namely an OSC-interface to the previously mentioned SenseWorld DataNetwork. The framework provides for robust methods to allow for quick reconnection upon restarting the code or patch. The DataNetwork provides a very quick way of sharing any data that may be needed by more than one collaborator, and it is used for sharing show control data (timing, cues), as well as sensor data and output data.

4.5 Managing synthesis processes

SC3 has two distinct methods for working with **Synth**s. One is instantiating a **Synth** directly on the server and then changing parameters of a synth either manually or automated in a task or routine. The other method is using the **Pattern** infrastructure, which provides many higher-level mechanisms for creating sequences in time.

In *Chronotopia* I mostly employed the **Pattern** infrastructure, to create spatial sequences across the light matrix. Only in a few instances I found it more convenient to instantiate **Synth**s, which would then be mapped to control busses with data from the motion tracking.

For *Schwelle* I built up an infrastructure to deal with common methods to handle **Synth**s and their parameters. The class **SchwelleInstrument** handles common methods for starting and stopping **Synth**s, including fading in and out, and providing a submix for the given instrument for individual volume control; various subclasses then implement different variants of instruments, depending on the use of samples (**Buffer**s), audio input from a microphone, mapping of controls to sensor data, as well "clouds" of **Synth**s dependent on input data. Each instrument has a default GUI to see and manipulate the status of the instrument.

For *JND/Semblance* I developed the preset system mentioned above. In addition to this preset system, I developed a central engine, the **JNDEngine** which manages all **JNDSynth**s and their connections to the DataNetwork. **JNDSynth** provides control over settings and mapping of the synthesis processes to data from the DataNetwork. **JNDEngine** also has a graphical interface, with volume controls for all running synths, and buttons to open GUIs for editing individual **JNDSynth**s.

The approach in **JNDSynth** is more general in the sense that it doesn't require many subclasses for special cases of synths, but the submixing approach of **SchwelleInstrument** is absent and it would not yet be able to handle the clouds used in *Schwelle*.

In future work, I anticipate merging the two approaches into a common class.

4.6 Spatialisation methods

All of the works involved spatialisation of some kind. In *Schwelle* there was a setup with two "rings" of speakers, four speakers around the performance area, and then four (or more)

speakers around the audience. Additionally, there were two speakers mounted at the ceiling, one pointed downwards, and one towards the ceiling, so that the audience would only get reflected sound from that speaker. Sounds were then either routed to specific speakers, or panned dynamically between the two rings of speakers. In the class **SchwelleSurround** I created various standard methods for the sound spatialisation, which could then be applied to parts of the soundscape at specific moments during the performance. Thus I was routing the output from one or more **Synth**s through another **Synth**, which did the spatialisation.

In *Chronotopia* I was working with a matrix of outputs, so I needed to pan between these outputs, without wrapping around to the other side. As *SC3* at the time of creation of *Chronotopia* only had a panner that wraps around (**PanAz**), I doubled the amount of output channels used within the panner, and then only sent the channels I actually needed to the output of the **Synth**. But, I also suggested to the *SC3*-community that there should be another panner **UGen** which can pan between multiple channels, but not wrap around. This led to the development of the **UGen PanX**. In order to direct the signals to the right outputs, I either made synths outputting to a specific channel in the matrix (using a **LightGrid** class to select the channel numbers from the one-dimensional array), or I used **SynthDef**s which embedded the **PanX UGen**.

In *JND/Semblance* I was working again with a matrix of outputs, so **PanX** was used extensively. Rather than routing **Synth** outputs to various spatialisation **Synth**s, I developed a system for dynamic creation of **SynthDef**s, where you can define a signal function, which is stored in a library (**JNDSignalLib**), and then create a **JNDSynthDef** with specific types of spatialised outputs. Furthermore all the **JNDSynthDef**s are stored in a separate **SynthDescLib**, which can be browsed with a graphical interface to test each **SynthDef**.

4.7 Show control

In all projects some kind of show control was needed. All works have distinct scenes or movements in which certain things need to happen at certain times (usually referred to as cues in live theater lingua franca).

In *Schwelle* the cues were quite closely tied to the performer's movements on stage, and in certain cases they would prompt the performer. Since there is a fair amount of improvisation in the piece, there was no absolute time at which these cues needed to be executed, although we had defined relative times between events in certain occasions (for the latter, I created a simple **ShowTimer** which showed me a window of how much time had elapsed since a certain moment). In addition, some cues needed specific code to be executed shortly beforehand to prepare processes (allocation of resources), and some others needed code to clean up afterwards (freeing resources). While I started writing a general purpose class for this, I did not end up using this, being unsure about its robustness in showtime (not having tested it extensively during rehearsals). Rather I ended up with a file with code snippets organised according to the timeline of the show, with numerous comments as to when to execute which code. This also allowed me to make quick changes "on the fly" during the performance.

On the other hand in *Chronotopia*, the timing of the piece is strictly tied to the music score and there is no improvisational aspect in the piece. Here, I ended up creating a **CueList** class, which executes functions at specific given frametimes. The cue list is stored in a code file, where functions can be changed, or alternatively you can use the class instance methods to add functions at specific times. The current time was then updated according to the playback of the sound file with the music score.

For *JND/Semblance*, I used three tasks (**Tdef**s) for the three movements of the piece and used a master task, starting these three tasks at the right time. While this allowed us to try out each movement separately, while preparing the piece, this approach was not yet completely satisfactory in its use, mainly because I had to recalculate the total durations of each movement (for proper execution of the master task) everytime we changed the timing of the piece during the preparations.

During rehearsals of the two theatrical pieces that we often had to go back and forth between specific scenes; this is actually the main challenge for coming up with a general purpose cue system, since skipping back and forth means taking care of the proper allocation and freeing of resources, depending on where we are in the show. Also certain cues may have set events in motion, which run during a number of scenes, so there has to be a check which events should be

turned on at a specific time. Finally, of course, during rehearsals the shape of the piece may change, so a quick editing of cues should be possible too.

4.8 Summary

From the above we can see that the different projects have led to the exploration of multiple ways of working on similar problems. The experiences made with capturing, manipulating and sharing sensor data has resulted in a consolidation into the SenseWorld DataNetwork.

The work done in *JND/Semblance* is moving towards a complete composition system that makes it easy to create signals with different spatialised outputs, and creating presets for different instruments and playing and managing the running synths. This system integrates with the DataNetwork. On the other hand there are still some approaches deployed in *Schwelle*, which would be useful to integrate with the JND system to create a complete system.

And finally, the different approaches for show control could still be consolidated into a generalised system that integrates with the composition system and the DataNetwork.

5 Software tools made public

My artistic work has resulted in several extensions to *SuperCollider* (available as "Quarks"[7]), additions to the standard capabilities of *SC3*, as well as standalone programs (supplied with *SC3* classes to interact with them). In this section I will briefly discuss the various tools that have been released and are publicly available.

5.1 SenseWorld and SenseWorld DataNetwork

The SenseWorld Quark is a collection of classes used for dealing with sensor data at a higher level, and some convenience methods. It contains some language side methods to calculate statistics of incoming data streams, as well as a number of PseudoUGens[8] to do the same.

The SenseWorld DataNetwork is a set of classes dealing with sharing data between collaborators using various programming environments. This framework is discussed at length in [Baalman et al., 2009].

5.2 GeneralHID

Moving back and forth between running code on Linux and OSX developed the need for a general approach for interfacing with HID devices. Working from various parallel, platform specific implementations used in *Schwelle*, and also some previous projects, I developed the abstraction **GeneralHID**, which provides a common interface to both **HIDDeviceService** (the OSX HID implementation in *SC3*), and **LID** (Linux input device). The **GeneralHID** abstraction is part of the standard distribution of *SC3* since May 2007.

5.3 WiiOSC and SC3 WII implementation

The use of the WiiMote in *Schwelle* has resulted in the publicly availabe Linux based program *wiiosc*[9], which captures the data from a WiiMote and sends it to a desired client using the OSC-protocol using *liblo*[10]. It has also resulted in a native implementation in the *SC3* language for direct access to the WiiMote, which has been part of the distribution since June 2007.

5.4 MotionTrackOSC

To use videotracking natively on Linux, I created MotionTrackOSC[11], based on the OpenCV library[12] and *liblo*, a simple adaptation of the motion tracking example of the OpenCV library, expanded with OSC control of parameters, and sending the data to a specific client. Furthermore, this program is integrated with *SC3*, through some classes implementing the OSC-communication. As the output of MotionTrackOSC is "raw", namely there is no consistent numbering of the motion tracking points, I implemented an algorithm to keep track of the positions of previously detected moving points and matching these to the new ones, so that the identifiers are consistent. Additionally, I implemented some algorithms to filter out "short-lived" tracking points, which can occur when the light conditions of the tracked area change — this typically happens a lot in a theatrical context.

[7] `http://quarks.sourceforge.net`

[8] PseudoUGens are implemented as small code blocks in *sclang* consisting of other **UGen**s, which can be used just like any other **UGen**s in a **SynthDef**.

[9] available at `http://www.nescivi.nl` since August 2007.

[10] `http://liblo.sourceforge.net`

[11] available at `http://www.nescivi.nl` since January 2009.

[12] `http://opencv.willowgarage.com/`

5.5 DMX

DMX is "the MIDI of theater lighting control", a serial protocol consisting of 8bit control values for up to 512 light channels within one DMX "universe". Most common theater light devices can be controlled via DMX, such as dimmer packs, stroboscopes and motorized lights with many individual control channels. Although there exists a *dmx4linux*-project[13], that project has a very lowlevel approach and there are hardly any programs that integrate with interactive software. As a result of the projects discussed in this paper, a DMX extension has been made for *SC3*, which can currently communicate with the EntTec DMX USB Pro[14]. This extension will eliminate the dependency on Max/MSP for DMX control and simplify some of the setups in the future.

5.6 PanX

The need to spatialise sound on a grid of outputs (speakers) rather than a circle led to the development of the **PanX UGen**, which allows for panning similar to the **PanAz UGen**, but does not wrap around. This **UGen** was implemented by Josh Parmenter and is available from the *sc3-plugins* project[15].

6 Conclusions

Interactive live performance is a challenging and exciting context for coding, and *SuperCollider* is certainly a suitable choice of language for this purpose. Creating tools for solving problems as they are encountered (or invented) may lead to ad-hoc solutions for one performance, but result in more solid tools in subsequent works as problems reoccur. While most tools were written based on an immediate need, the publication of these tools has helped and hopefully will help many other artists working in similar areas.

This retrospective analysis of my coding strategies in these three projects will hopefully give other artists and researchers some insight in the creative process of working with code in artistic projects, and the specific challenges in this context.

7 Acknowledgements

Thanks to Chris Salter and Harry Smoak for the many years of collaboration; also to Alberto de Campo for a number of pleasant and insightful coding sessions. Thanks to Josh Parmenter for implementing the **PanX UGen**.

[13] `http://llg.cubic.org/dmx4linux/`

[14] `http://www.enttec.com/`

[15] `http://sc3-plugins.sourceforge.net`

SuperCollider in brief

SuperCollider consists of two components: an audio programming language, called *sclang*, and a audio synthesis engine, called *scsynth*; these two components communicate with each other via a set of OSC messages. *sclang* is an object oriented audio programming language with dynamic type casting and garbage collection. As a reference for some *SC3* nomenclature I have been using throughout this paper:

UGen unit generator, or its representation in *sclang*.

SynthDef "blueprint" for a Synth, like an "instrument", consisting of a set of interconnected **UGens**.

Synth a running synthesis node on *scsynth*, created from a **SynthDef**; like a "voice".

Quark "packaged" set of *sclang* classes to extend the default class library of *SC3*.

SuperCollider can be found at `http://supercollider.sourceforge.net`.

References

Marije A.J. Baalman, Daniel Moody-Grigsby, and Christopher L. Salter. 2007. Schwelle: Sensor augmented, adaptive sound design for live theater performance. In *Proceedings of NIME 2007 New Interfaces for Musical Expression, New York, NY, USA*.

Marije A.J. Baalman, Harry C. Smoak, Christopher L. Salter, Joseph Malloch, and Marcelo Wanderley. 2009. Sharing data in collaborative, interactive performances: the SenseWorld DataNetwork. In *Proceedings of NIME 2009 New Interfaces for Musical Expression, Pittsburgh, PA, USA*.

Johann Friedrich Herbart, 1969. *Kleinere Abhandlungen*, chapter "De Attentionis Mensura causisque primariis" (orig. published 1822). E.J. Bonset, Amsterdam.

M. Wright, A. Freed, and A. Momeni. 2003. 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.

Applications of Blocked Signal Processing (BSP) in Pd

Frank Barknecht
Cologne,
Germany,
fbar@footils.org

Abstract

Sample processing in Pure Data generally is block-based, while control or message data are computed one by one. Block computation in Pd can be suspended or blocked to save CPU cycles. Such "blocked signals" can be used as an optimization technique for computation of control data. This paper explores possible applications for this "Blocked Signal Processing" (BSP) technique and presents a system for physical modelling and for feature extraction as examples.

Keywords

Pure Data, parallel computation, algorithmic composition, optimization, physical modelling, feature extraction

1 Introduction

Software for computer music and realtime synthesis has to deal with two competing requirements: It continuously has to compute audio samples at a rate specified by the underlying samplerate (like 44.1 kHz as "CD quality") and it has to deal with sporadic events, for instance note events coming from midi streams. In many computer music system, the events of such control streams are computed less often than the actual audio samples: In addition to the sample rate as a defining parameter of a music software a slower *control rate* was added: "The *control rate* is the speed at which significant changes in the sound synthesis process occur. For example, the control rate in the simplest program would be the rate at which the notes are played. [...] The idea of a control rate is possible because many parameters of a sound are 'slowly varying'."[1]

The separation of control and sample rate has been a part of computer music software since its early days: "Among the languages of the Music *N* family, Music IV and its derivatives (including Music 4C) are sample-oriented, whereas Music V and Cmusic are block-oriented. The Csound language is also block-oriented, since it updates synthesis parameters at a *control rate* set by users."[2]

The introduction of different rates for audio and control streams makes specific optimizations for each domain possible. For the sporadic events of control streams, that only happen rarely compared to the calculation of audio samples, redundant or unnecessary calculations can be omitted.

Audio data however can be computed in blocks of samples. Instead of computing every sample and then the next, several samples are computed in one go, which significantly reduces the overhead in systems based on "unit generators": "This is done to increase the efficiency of individual audio operations (such as Csound's unit generators and Max/MSP and Pd's tilde objects). Each unit generator or tilde object incurs overhead each time it is called, equal to perhaps twenty times the cost of computing one sample on average. If the block size is one, this means an overhead of 2,000%; if it is sixty-four (as in Pd by default), the overhead is only some 30%."[3]

Apart from avoiding the overhead of function-calls for each unit generator, blocked processing also can be implemented in a way that reduces the number of memory allocations necessary. For this, the block size has to be constant over a sufficient time.

But block computation also has a major disadvantage: It adds latency of at least one block duration. "Sample-oriented compilers are more flexible, since every aspect of the computation can change for any sample".[4]

Contrary to audio data control streams usually are not computed in blocks. As most con-

[1] [Dodge and Jerse, 1985, p. 70]

[2] [Roads, 1996, p. 801-802]

[3] [Puckette, 2007, p. 63]

[4] [Roads, 1996, p. 801]

trol events happen only sporadically there may not be enough data to fill a block that would be useful to compute. For instance midi notes may be produced only every quarter beat which at a tempo of 120 BPM would amount to only one note event every 500 milliseconds. Filling a block of 60 quarter notes would then take half a minute - no sane musician would accept a latency of that duration.

While this is an extreme and admittedly a bit silly example, the number of control events often is small enough to not let the overhead of calling the unit generators for every event affect the overall performance of the system.

But this is only valid, as long as the number of events to compute stays small. Especially in algorithmic composition, in simulations or similar cases the amount of data in a "score" can become very big. The overhead of control information now becomes a problem. Computation in blocks may yield a significant performance gain. The results of the control computations may still only be needed infrequently compared to the rate of audio signals. Some modern languages like LuaAV or ChucK are designed to deal with this problem right from the start. In ChucK, "the timing mechanism allows for the control rate to be fully throttled by the programmer - audio rates, control rates, and high-level musical timing are unified under the same timing mechanism."[5]

Pure Data was not designed with variable control rates in mind, but a peculiar feature of Pd can be exploited to do block computations on control data, confessedly in a limited way. Pd can suspend its own audio computations locally using the [switch~] object. This object can stop all sample computations inside of a Pd subpatch or canvas and restart it on demand. A common use is to activate parts of a Pd patch only when needed, for example to manage voices in a polyphonic synthesizer: inactive voices can be switched off when not in use to save CPU cycles.

A not so common use case for switched-off subpatches is introduced and explored in this paper. We use subpatches to perform block-computations at rates much lower than the audio samplerate. These computations will employ the unit generators originally intended to do audio signal computations. The computation rate is adjusted by suspending blocks locally. We will call this approach "Blocked Signal Processing" or BSP to have a catchy and short buzzword available.

[5] [Wang and Cook, 2004]

2 Blocked Signal Processing

A very simple example will now show the basic principle of BSP in Pd, how it can optimize certain actions and how it compares to traditional message computations. The task solved by the following two Pd code examples is simple: Read 4096 numbers stored in a table "ORIG", add 0.5 to it and store the result in table "RESULT".

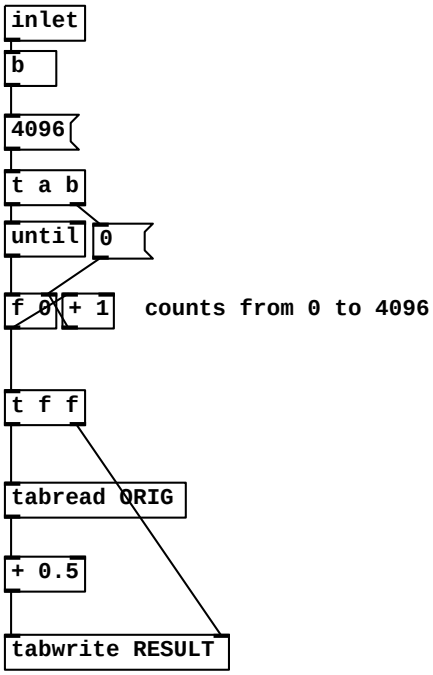


Figure 1: Transposing a table by 0.5 using message computations

Implemented with message objects, a counter is started by an incoming bang-message, its index number is used to read out the table data, a control-rate addition object adds 0.5 to it, then the value is stored.

The BSP implementation uses a pair of [tabreceive~] and [tabsend~] objects to constantly read and write the tables as a block of samples. The signal-rate addition object adds 0.5 in between. The [switch~ 4096 1 1] object resizes the blocksize to 4096 for this sub-canvas, so that the table-accessing objects can process that many samples. Additionally it switches off the DSP computation inside the subpatch at the beginning. So unless some messages to [switch~] switch on the computation, this part of the

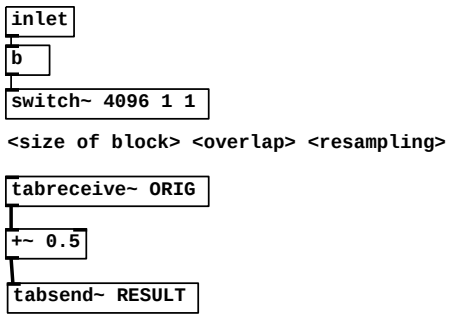


Figure 2: Transposing a table by 0.5 using BSP computations

patch doesn't consume any CPU resources. By sending a "bang"-message to [switch~], the subpatch is switched on for exactly one block of samples, then it's switched off again. During the time it is on, the actual computation happens, so the end result is the same as for the message version.

One small difference is important to note here: Probably for performance reasons Pd defers updates of the graphics in arrays or tables that are used with active [tabsend~] objects. Changes to the included data are only visible when the graph is closed and opened again.

The two implementations of the "transposer" don't show much difference in CPU usage. This is expected, as the simple calculations made here do not involve many objects so the overhead is negligible. Also both the BSP and the traditional method avoid any memory allocation by directly working on pre-allocated tables. When dealing with lists of numbers of arbitrary size, a common idiom in Pd is to build these lists by pre- or appending elements to existing lists stored in [list] objects.[6] For longer lists this can become a major cause of slowdown in Pd patches.

3 [physigs] - Physical Modelling implemented with BSP

The BSP technique should show more of its potential when applied to algorithms involving a bigger number of unit generators and more parallel tasks. We will now turn to such a use case and take a look at a physical modelling system based on spring-connected particles.

[6]See the list-help.pd file in Pd's documentation for an example. This file also shows the opposite operation of serializing a list with [list split] which is slow as well.

A particle simulation applies the Newtonian laws of mechanics to a simulation of point masses. The physically-inspired rules are used to calculate velocities and accelerations of point particles, that are defined by vectors describing their positions and impulses. Every particle also includes a force accumulator that holds any external forces applied to a particles. Forces, positions and impulses fully specify the current state of a particle system. Transitions from one state to a next are calculated in discrete steps: Usually a world clock is employed that advances one simulation step and initiates a new run of the physics calculations to find the new positions of the particles. As the same set of physical rules has to be applied to many particles, the problem is an ideal candidate for doing the calculations in blocks of particles.

With PMPD[7] and MSD[8], two implementations for this already exist as extensions to Pd (so called externals). For this paper a BSP-implementation of a particle system called "[physigs]" was written in Pd and is tested against the MSD and PMPD implementations. It's help file is shown in Fig. 7 at the end of this paper.

[physigs] is a particle simulation in two dimensions. In consists of a main Pd abstraction called [physigs] that can be called with a prefix-tag to make using the object several times in a single project possible. The state of the system is held in a number of Pd [table] objects for positions, velocities, masses, forces and a table that holds meta-data, currently only the mobility state of masses is watched: A mass can be mobile or fixed. A particle is identified by an integer number which is used as a lookup index into the state-holding tables. Particle 10 for example would hold its x-position in the 10th element of the table "pos-x", its y-position would be the 10th element in table "pos-y" and so on for tables "force-x", "force-y", "mobile" or "mass".

The size of these tables controls the size of the system: Tables of size 64 can control up to 64 particles. All 64 particles are computed regardless of particles actually being used. The table size can be selected when creating the [physigs] object.

In Figure 3 the calculations that advance the simulation one step are shown.

At the top left, the current x-positions and

[7][Henry, 2004]

[8][Montgermont, 2005]

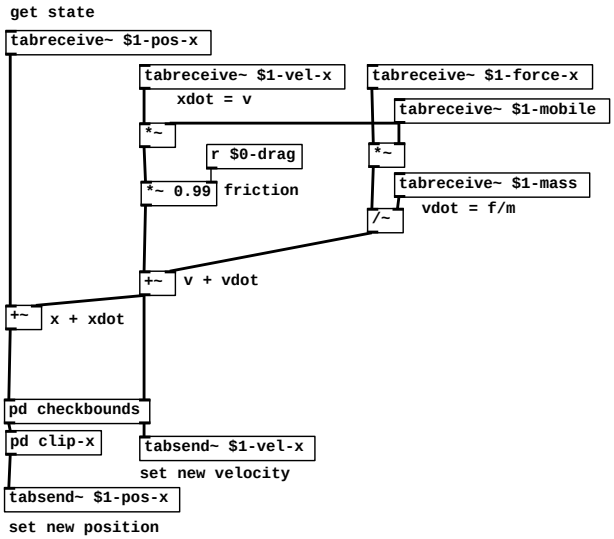


Figure 3: Advancing the world simulation one time step

x-velocities are read using [tabreceive~] objects. Respective [tabsend~] objects write the new positions back, after the physics laws have been applied. A table "force-x" holds an accumulation of all forces that have been applied to particles. The "mobile" table is 0 when a particle should be fixed, and 1 when it is mobile. Weights are stored in a "mass" table. Some friction is applied to fight instabilities and simulate air drag. The actual algorithm is rather simple: Any force on a particle is converted to an acceleration by dividing the mass, the acceleration is added to the velocity, which in turn is added to the position. The forces then get reset to zero, which is a step not shown in the figure. The subpatches "checkbounds" and "clip" restrict the possible positions to a configurable range and possibly flip the velocity to simulate bouncing of the world's ends. The same calculations are made for the y-coordinates as well.

The system is driven by an outside [metro] whose speed specifies the speed of the physical simulations. Metro periods in "haptic" ranges of 20 to 100 milliseconds are good choices. The [metro] then regularly generates bang-messages to switch on and off the DSP-signal computations in the subpatch holding the simulation step.

A similar construction in [physigs] calculates the forces of visco-elastic springs connecting two particles. These springs are defined by a number of tables again, where each spring indexes tables with its integer-ID. Any spring links two particles. The ID numbers of these two particles are held in two state tables called "link-m1" and "link-m2". Other state tables hold parameters like damping, stiffness, relaxed length and so on.

Springs generate forces that have to be accumulated into the tables holding the forces for each particle. This currently is realised in a separate step that doesn't use the BSP technique, because the "vanilla" version of Pd doesn't offer an object that can write into a table at a position specified by an audio signal. Miller Puckette intends to include such an object in future versions of Pd, the C-code for the object is rather trivial and it already exists as an external. Unfortunately falling back to traditional control-rate calculations in this part destroy many of the performance gains as we will see in the benchmark section.

3.1 Performance comparison

To compare the different implementations of a particle system, a test system has been devised that creates a configurable number of particles at random positions and with random mass and daisy-chains them with spring links. Such a performance test is part of the MSD distribution. Pd's built-in "Load meter" has been used to get rough CPU usage values. The world advances one step every 50 milliseconds (or 2 * 25 msec in [physigs] for link and mass computations each). Table 1 shows the results of several test runs.

It turned out, that [physigs] runs much slower than both MSD and PMPD when the control-rate calculations are used to distribute the spring-forces to each particle. Switching off this part of the patch in the BSP implementation will let [physigs] catch up to MSD and beat PMPD.

MSD doesn't have any overhead, as it computes the full simulation in just a single object. PMPD however has separate objects for each particle and link. In the example patch 4096 objects for the simulation participants alone are used. So here the overhead is significant as expected.

[physigs], especially when used without the control-rate workaround for a missing "write to table"-object, turns out to be a capable contender for physical modelling in Pd-vanilla or Pd-almost-vanilla environments. Note that these benchmarks are only meant to give a per-

Simulation Type	CPU Usage
MSD	5
physigs	30
physigs (w/o force distribution)	5
PMPD	16

Table 1: Simulation of 2048 particles and 2048 springs at metro-period of 50 ms using three different implementations

formance estimation and should not be taken as "hard numbers". But the author has successfully run [physigs] on the "RjDj" version of Pure Data on the iPhone with its much slower CPU compared to standard computers.

4 Feature extraction and analysis with BSP

[physigs] generates data-heavy control streams in a completely artificial manner. Similar amounts of data points have to be handled when analysing external audio coming in over the ADC (soundcard) and looking for certain features to guide a musical composition. Pitch tracking or onset detection work on single sound objects and must be prepared to react quickly. The [sigmund~] or [bonk~] objects that are part of Pd, thus run at audio rate. In this case, applying BSP would not be viable. But if a composer is interested in features on a "slower" time scale, BSP can be applied.

As an example lets consider the differences in time scale of physically moving between two rooms compared with playing two different notes on a clarinet. Changing rooms takes much more time than playing notes on instruments. To detect the clarinet's pitch changes, the software has to be constantly "alarmed" of the spectral content registered through the soundcard. However when trying to detect if a person holding a microphone changes rooms by analysing the spectral or reverberant characteristics of the two environments, spectral snapshots can be made and compared much less frequently than in the case of pitch detection.

The author has developed a set of BSP feature extraction (FE) objects mainly intended to be used in the RjDj version of Pd that runs on mobile devices. The timbreID collection of Pd externals by William Brent[9] served as an inspiration for these. Detecting "changed rooms" is a common need of composers working for RjDj. The analysis rate of the FE objects is configurable similar to the [physigs] object by adapting the speed of a [metro] object that blocks and activates them. Just as in the introductory example of a "table transposer" the objects work on shared table data and extract statistical features like centroid, mean, energy or flatness.

If the table to be analysed holds a magnitude spectrum, the extracted features describe the frequency content of the sound environment. Of course the objects may be used to acquire statistical analysis of tables holding other kinds of data as well.

As is well-known spectrum analysis with Fourier transformations is a relatively costly operation. In Pd it is already carried out in subpatches, so the analysis is a natural candidate to be "blocked" with the BSP technique. Of course the time resolution gets worse in this case, and overlapping analysis is not useful anymore, when there are pauses between adjoining runs of the transformation. But if all that is needed is a "snapshot" of an environment's spectral status, BSP-blocking is a viable way to save CPU resources.

Results of a spectral analysis written to a table can be analysed by several objects at the same time, forming feature vectors that can be post-processed for classification. The costly Fourier transformation has to be run only once for each vector.

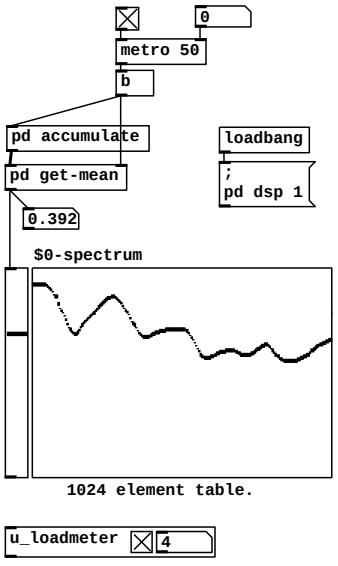


Figure 4: Calculating the arithmetic mean with BSP - main patch

[9] [Brent, 2009]. Newer versions of timbreID include both audio- and control-rate versions of its analysis objects.

As an example for an FE-BSP object we will take a closer look at the calculation of the mean of a table. The arithmetic mean needs an accumulation of all table values, which then is divided by the table size.

While it would be possible to divide every single value by the table size before adding them, here the accumulation is calculated first, then a single division is made. This gives a minor speed-up, but here it should also illustrate the way, the execution order for BSP calculations can be controlled.

With Pd's normal message computations, [trigger] objects are used to specify a certain execution order. With signal objects the execution order can only be manipulated by laying out the operations into subpatches, that are connected by signal patchcords.[10] The connected subpatches are calculated by Pd's main scheduler in the order they are connected with objects earlier in the chain calculated first (contrasting Pd's depth first scheduling for messages).

In Figure 4 both subpatches are connected like this, so first every signal object in [pd accum] is run, then every signal object in [pd getmean]. Dummy signal inlets and outlets are included in both subpatches to allow making these order-forcing connections.

The accumulation of all values in a table can be coded using only a handful of objects: Using a one-pole recursive filter as supplied by Pd in the [rpole~] object with a coefficient of 1 will accumulate all data as long as its switched on. The filter is reset to zero after each run using the [bang~] object that outputs a bang after each DSP cycle.

The output of [rpole~] is written into a summing-table using [tabsend~]. The final position in this table will hold the accumulation value.

In the second subpatch (Fig.6) it's a [bang~] object that will transport this value to the division by N before it is reported to the outlet after the block calculation has completed. Mean calculation has a latency of one block of samples or blocksize divided by samplerate. Upsampling inside of subpatches to reduce this latency can be achieved by changing the third argument of the switch-objects or by sending it respective messages.

When implementing the FE objects, the lack of a Pd object to selectively write a value into a table at a certain position specified by an audio signal was especially limiting. For example while finding local extrema (minima or maxima) in a table is easy by scanning through the table once and comparing two adjacent table values, the author hasn't yet found a way to efficiently store the locations of these extrema for later use.

Indexed table writing would also make another workaround or "hack" in the FE objects unnecessary: To calculate the geometric mean the product of all values in a table is needed, but there is no "vanilla"-way to reuse the result of the multiplication of previous samples again as it is with the one-pole filter that adds previous results (output values) back to its input: the next sample in a block. The workaround currently applied is to transform the multiplications to additions using logarithms.

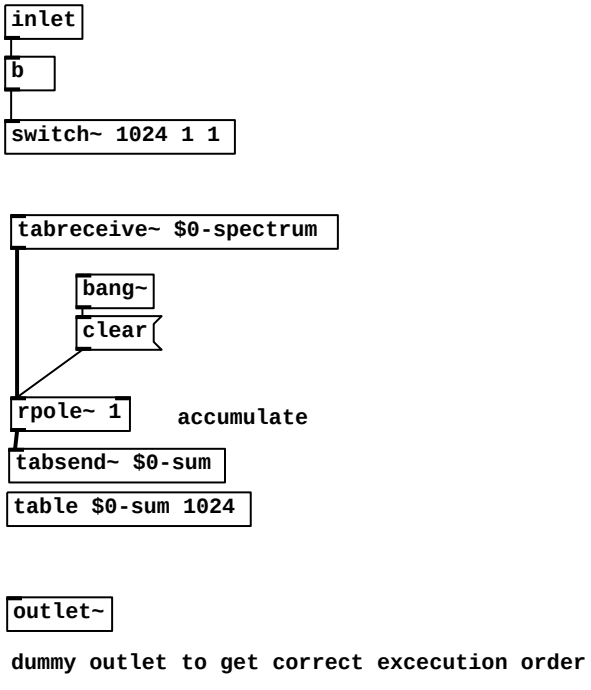


Figure 5: Subpatch "accumulate": Accumulating table values with a one-pole filter

5 Other Applications of BSP

The BSP technique can be applied to various other areas, where parallel, block-based processing is needed. Examples would be Cellular Automata, L-Systems or swarm/flock systems.

[10] [Puckette, 2007, p. 212-216]

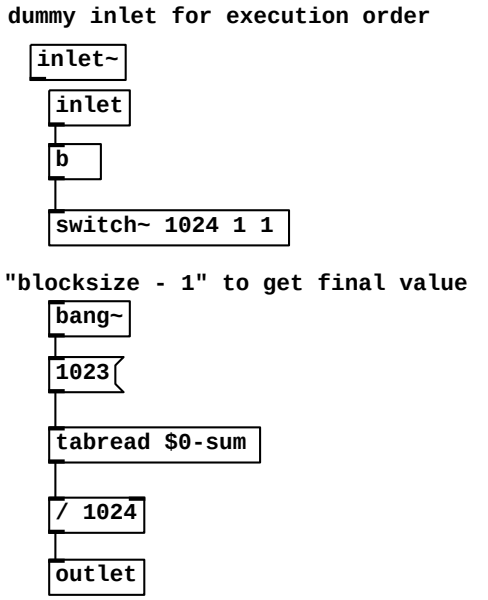


Figure 6: Subpatch "get-mean": Calculating the mean from the accumulated table and the table size.

But even for much simpler daily tasks in the life of a Pd user, BSP is worth a look. For example to copy the content of one table to another one, a blocked combination of tabplay~ and tabwrite~ can be used.

6 Limitations of BSP

As many optimization techniques, BSP has several limitations that have to be weighted against the possible performance gains. One important problem is execution order. Pd alternates audio and message computations. BSP however lives in a grey area between the audio signal and control computations. New results will be computed in the signal pass, where no other control calculations happen. It is not possible to trigger other control-events in the middle of a BSP-run.

The algorithms to be used in BSP cannot use recursion inside one block, because feedback of results computed at a later point in the DSP tree to an earlier point is not possible. The minimal feedback delay time in Pd is one block, other constructs result in "DSP loop detected"-errors in Pd. This limit is expected to complicate applying BSP in recursive algorithms like L-Systems. The inclusion of a suitable table-writing object in Pd vanilla as mentioned above could alleviate this problem a bit.

BSP deals only with numbers, so it's applicability to text processing is very limited, which affects the use of formal grammars. The symbols used in alphabets of L-Systems or similar systems based on rewrite rules have to be converted to numbers by implementing a translation map.

7 Conclusions and future work

BSP has been successfully applied to a simple physical simulation for this paper and a growing set of feature extraction objects. The [physigs] object will be refined and published under an open source license, while the feature extraction objects will become a part of the "rj" library developed for the RjDj application. While BSP is a powerful and so far under-used technique in Pd, it cannot magically transform Pd to become a true multiple-rate software. Music compilers like ChucK, SND-RT or LuaAV still deal with the competing demands of variable control rates in a cleaner and more flexible way.

8 Acknowledgements

Parts of the research for this paper was kindly supported by Reality Jockey Ltd./RjDj.

References

William Brent. 2009. Cepstral analysis tools for percussive timbre identification. In *Proceedings of the 3rd Pure Data Convention in Sao Paulo.*

Charles Dodge and Thomas A. Jerse. 1985. *Computer Music.* Schirmer Books, New York, NY, USA.

Cyrille Henry. 2004. Physical modeling for pure data (pmpd) and real time interaction with an audio synthesis. In *Sound and Music Computing '04.*

Nicolas Montgermont. 2005. Modeles physiques particulaires en environnement temps-reel: Application au controle des parametres de synthese. Master's thesis, Universite Pierre et Marie Curie.

Miller Puckette. 2007. *The Theory and Technique of Electronic Music.* World Scientific Press.

Curtis Roads. 1996. *The Computer Music Tutorial.* MIT Press, Cambridge, MA, USA.

Ge Wang and Perry R. Cook. 2004. On-the-fly programming: Using code as an expressive

musical instrument. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 138–143.

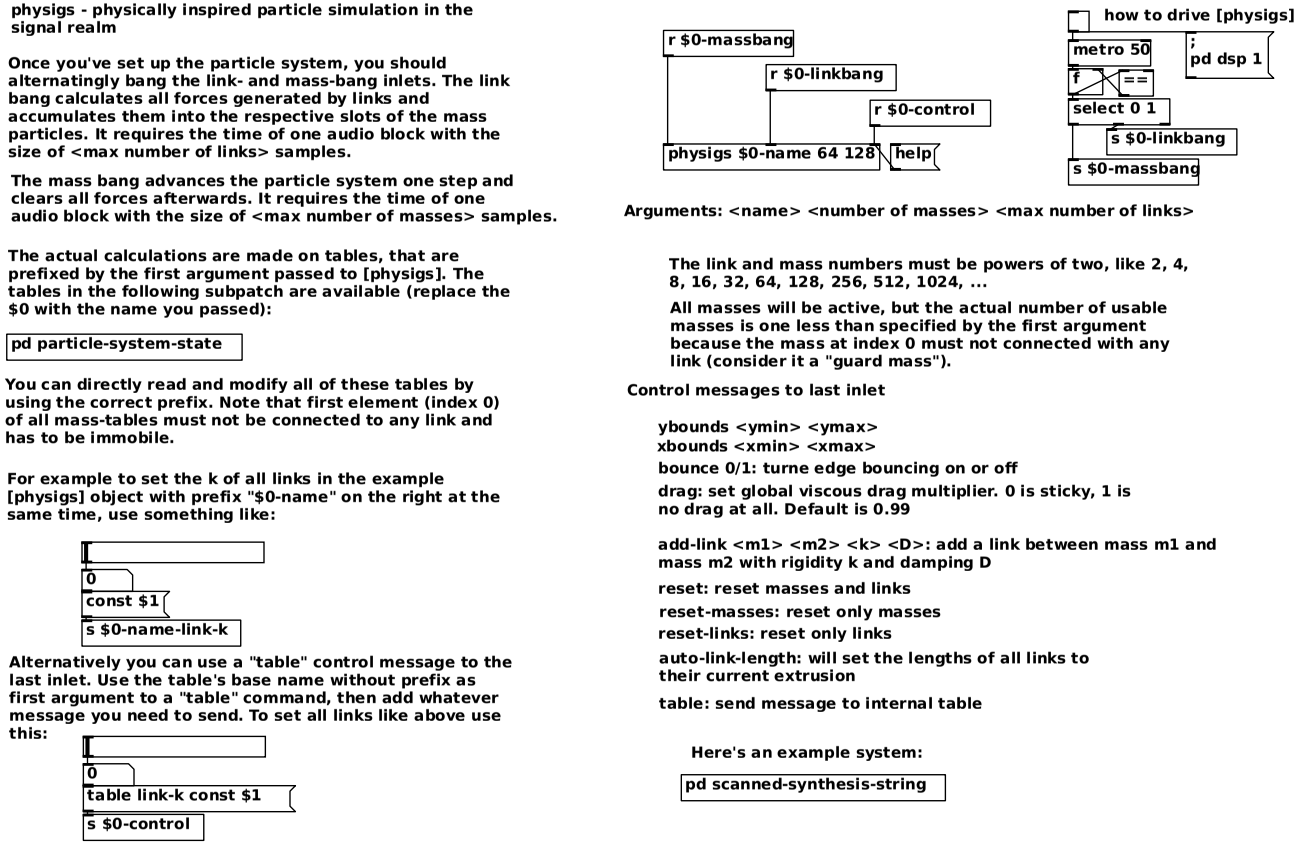


Figure 7: Help-file for the [physigs] abstraction

OrchestralLily: A Package for Professional Music Publishing with LilyPond and LaTeX

Reinhold Kainhofer, http://reinhold.kainhofer.com, reinhold@kainhofer.com
Vienna University of Technology, Austria
and
GNU LilyPond, http://www.lilypond.org
and
Edition Kainhofer, http://www.edition-kainhofer.com, Austria

Abstract

LilyPond [Nienhuys and et al., 2010] and LaTeX provide excellent free tools to produce professional music scores ready for print and sale. Here we present the OrchestralLily package for LilyPond, which simplifies the creation of professional music scores with LilyPond and LaTeX even further. All scores are generated on-the-fly without the need to manually specify the structure for each individual score or part. Additionally, a LaTeX package for the prefatory matter is available and a templates system to create all files needed for a full edition is implemented.

Keywords

LilyPond, Music scores, Publishing, LaTeX, Software package

1 Introduction

In professional music publishing applications like Finale, Sibelius and SCORE are the predominant software packages used. However, the open source applications LilyPond and LaTeX provide excellent free alternatives for producing professionally looking music scores, as well. To ease the production of such scores even further, we developed a package called OrchestralLily for LilyPond and LaTeX. Instead of having to produce each score and instrumental part manually in LilyPond, this package produces these scores dynamically from the music definitions.

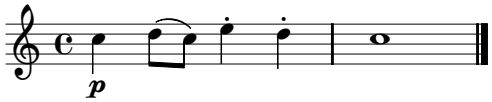
2 A Short Introduction to LilyPond

LilyPond, the music typesetting application developed under the umbrella of the GNU project, is a WYSIWYM ("What you see is what you mean") application, taking text files containing the music definitions and corresponding settings and typesetting it into a PDF file. Writing a LilyPond score is very similar to coding a software program.

In this section we will give a very short overview about the LilyPond syntax, mainly to highlight our motivation to create the OrchestralLily package, which creates the scores dynamically. Fore a more detailed overview over LilyPond we refer to the excellent Documentation of the LilyPond project: http://www.lilypond.org/Documentation/.

A very simple LilyPond score containing only one staff has the following form:

```
\version "2.13.17"
\relative c'' {
  c4\p d8[( c]) e4-. d-. |
  c1 \bar"|."
}
```



All notes are entered by their note name[1], followed optionally by the duration and additional information like beaming ([and]), dynamics (e.g. \p) and articulations (e.g. -. for staccato). When running this file through the LilyPond binary, a five-line staff with a treble clef, 4/4 time signature and C major key is implicitly created. All layouting and spacing is done by LilyPond according to best practices and standards from music engraving.

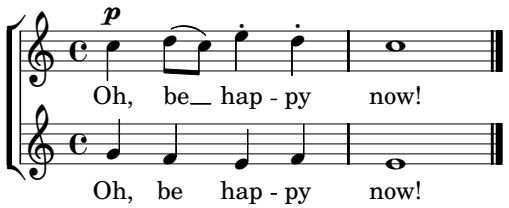
2.1 Writing Full Scores in Pure LilyPond

To produce a score containing a system with more than one staff (e.g. full orchestral scores, choral scores or vocal scores) or to produce a score with lyrics attached to the notes, LilyPond can no longer automatically create the staves, but one has to write the score structure manually into the LilyPond file:

```
\version "2.13.17"

sopmusic = \relative c'' {
  c4\p d8[( c]) e4-. d-. | c1 \bar "|." }
soplyrics = \lyricmode { Oh, be -- hap -- py
    now! }
```

[1] Using Dutch names by default: b for the note below c and the postfix -is for sharp alterations and -es for flat alterations. Other languages can easily be used by including a language file, e.g. \include "english.ily".

```
altomusic = \relative c'' {
  g4 f4 e4  f | e1 \bar "|." }
altolyrics = \lyricmode { Oh, be hap -- py
    now! }

\score {
  \new ChoirStaff <<
    \new Staff {
      \new Voice = "Soprano" {
        \dynamicUp \sopmusic
      }
    }
    \new Lyrics = "SLyrics"
        \lyricsto "Soprano" \soplyrics
    \new Staff {
      \new Voice = "Alto" {
        \dynamicUp \altomusic
      }
    }
    \new Lyrics = "ALyrics"
        \lyricsto "Alto" \altolyrics
  >>
}
```



Here, the actual music definitions require only eight lines of code, while the structure of the score requires already more lines.

As one can image, creating a full orchestral score with lots of instruments and multiple movements quickly becomes a nightmare to produce manually. Each staff and each staff group defined this way takes 3 to 7 lines of code, quickly leading to a score structure definition of hundreds of lines. For example, a large work with 23 instruments and 12 movements has 276 individual staves, not counting groups. Even worse, each movement typically has the same well-defined structure in the orchestral score. So, a lot of code is duplicated, with the only difference being the music expressions inserted into the scores.

This makes it extremely hard to maintain large orchestral scores in pure LilyPond, and even small changes to the appearance of only one instrument require lots of changes.

2.2 C++ and Scheme / Guile

Internally, LilyPond is written in C++ with Guile as embedded scripting language. Many parts of the formatting code (e.g. all graphical objects like note heads, staff lines, etc.) are defined in Guile and can be modified and overwritten easily using Scheme code embedded into the score. LilyPond even provides an extensive Scheme interface to most of the functions required to create a score. This interface is the key for our OrchestralLily package, where all scores are generated on-the-fly using Scheme.

3 OrchestralLily: An easy example

OrchestralLily uses a slightly different approach than manually writing LilyPond scores: Instead of telling LilyPond explicitly about staves and groups, the score structure is already built in, using default names, and the user only has to define some specially-named variables, containing the music, lyrics, clef, key, special settings, etc. The score is then generated on-the-fly by the following call:[2]

```
\createScore #"MovementName"
    #'("Instrument1" "Group2" "Instrument2")
```

Of course, multiple `\createScore` commands can be given in a file to produce multiple scores in the same file (in particular, this is used in large works with multiple movements, where all movements should be printed sequentially).

The specially-named variables holding the music definition mentioned above have the form

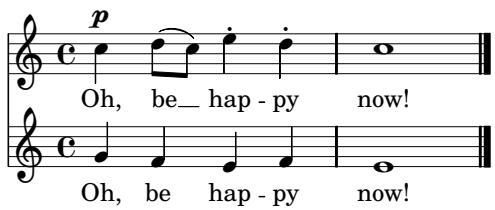
```
[MovementName]InstrumentMusic
```

where `Instrument` is replaced by the (default) abbreviation of an instrument or vocal voice and `[MovementName]` is optional for pieces with only one movement. To define a special clef, key, time signature, lyrics or special settings for a voice, one simply defines a variable containing the clef, key, time signature, etc. This special variable for each instrument is called `[MovementName]InstrumentXXX`, where `XXX` is either `Clef`, `Key`, `TimeSignature`, `Lyrics`, `ExtraSettings`, etc.

To show how this works, the two-voice example from above using OrchestralLily will now look like:

```
\version "2.13.17"
\include "orchestrallily/orchestrallily.ily"

SMusic = \relative c'' {
  c4\p d8[( c]) e4-. d-. | c1 \bar "|." }
SLyrics = \lyricmode {
  Oh, be __ hap -- py now! }
AMusic = \relative c'' {
  g4 f4 e4  f | e1 \bar "|." }
ALyrics = \lyricmode {
  Oh, be hap -- py now! }

\createScore #"" #'("S" "A")
```

[2] The hash sign `#` indicates a scheme expression, the `#'(...)` is a list in Scheme syntax.



It is clear that this automatic creation of staff groups saves a lot of effort for large-scale orchestral projects. It should be noted that OrchestralLily has a large hierarchy of orchestral instruments, including the identifier `"FullScore"` for a full orchestral score. So instead of `#'("S" "A")` above, we could have also said `#'("FullScore")` to generate a full score of all defined voices. Voices not defined will be ignored, so `#'("S" "A" "T" "B")` would have the same output, as the T and B voices are not defined and thus not included in the output.

4 Structure of a Score

To understand OrchestralLily's approach, we have to take a closer look at the organization of a full score. A music score has an intrinsic hierarchy of instruments and instrument groups, as shown in Figure 1.

This hierarchy is pre-defined in OrchestralLily and will be used, unless the LilyPond score explicitly overrides it:

- The instruments are named by their standard abbreviation (e.g. "V", "VI", etc. for violins, "Fag", "FagI" etc. for bassoon, "Ob" for oboe, "S", "A", "T", "B", "SSolo" etc. for vocal voices, etc.).
- Each group of instruments has a predefined name: "Wd" for woodwinds, "Br" for the brass instruments, "Str" for strings (except continuo, i.e. celli and basses, which are typically not included in the strings group, but placed at the bottom of a full score), "Choir", "Continuo", etc.
- Several types of scores are predefined: "LongScore", "FullScore" (like "LongScore", except that two instruments of the same type, e.g. ObI and ObII, are combined and share one staff rather than using separate staves), "VocalScore" (only the vocal voices and the piano voice), "ChoralScore" (only the vocal voices, no instruments).

The score types pre-defined in OrchestralLily adhere to the standard instrument order usually employed for full orchestral scores.

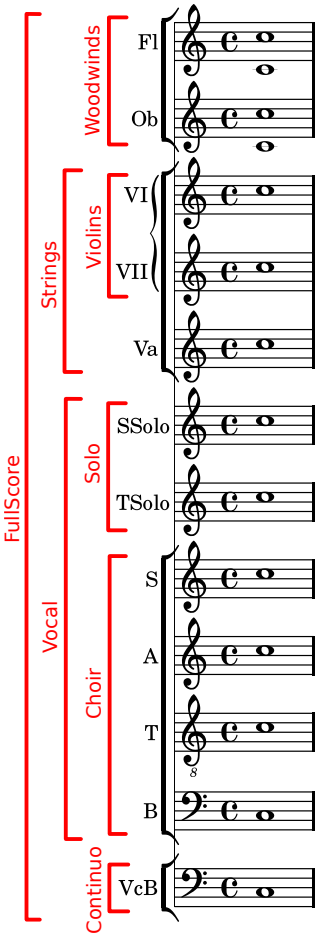


Figure 1: Hierarchy of an orchestral score

5 More complex examples

The examples so far placed the music definition and the actual score creation via `\createScore` into the same file. For larger projects, it is advisable to place the music definitions into a separate file, as we need several different score files, each of which will include this definitions file. All the examples in this section will use the following music definition file "music-definitions.ily", which defines a flute, a violin, soprano and alto, as well as a continuo part for a movement named `Cadenza`. Also, a Piano reduction is defined in this file. Notice also, that this include file already loads OrchestralLily, so we don't have to do this again in the file for each individual score.

```
\include "orchestrallily/orchestrallily.ily"
\include
    "orchestrallily/oly_settings_names.ily"

\header {
  title = "A cadenza"
}
CadenzaPieceNameTacet = "Cadenza tazet"
```

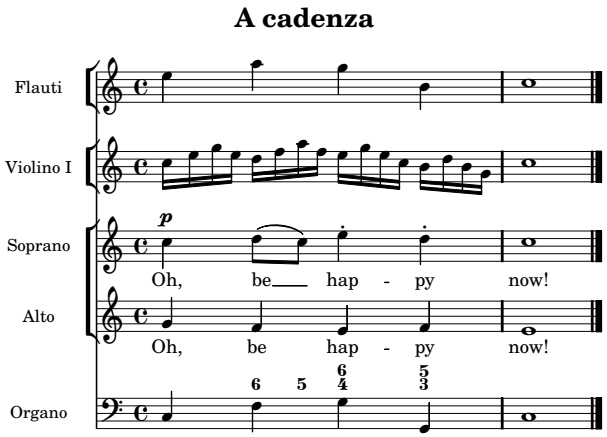
```
% Flute and Violin:
CadenzaFlIMusic = \relative c'' { e4 a g b,
    | c1 \bar "|." }
CadenzaVIMusic = \relative c'' {
 c16[ e g e] d[ f a f] e[ g e c] b[ d b g] |
 c1 \bar "|."
}

% The vocal voices:
CadenzaSMusic = \relative c'' {
  c4\p d8[( c]) e4-. d-. | c1 \bar "|." }
CadenzaSLyrics = \lyricmode {
  Oh, be __ hap -- py now! }
CadenzaAMusic = \relative c'' {
  g4 f4 e4  f | e1 \bar "|." }
CadenzaALyrics = \lyricmode {
  Oh, be hap -- py now! }

% Continuo: Organ / Celli / Bassi / Bassoon
CadenzaBCMusic = \relative c { c4 f4 g g, |
    c1 \bar "|." }
CadenzaFiguredBassMusic = \figuremode {
  s4 <6>8 <5> <6 4>4 <5 3> | s1
}

% Piano reduction:
CadenzaPIMusic = \relative c'' {
  \twoVoice {
    c16[ e g e] d[ f a f] e[ g e c] b[ d b
        g] |
  } {
    e4 a <g c>4 <b f>4
  } | % 2
  <c g e>1 \bar "|."
}
CadenzaPIIMusic = \relative c {
  <c g'>4 f <g c>4 <g d'> | % 2
  <c c,>1 \bar "|."
}
```

All variables in this file start with `Cadenza`, followed by the instrument name, which is how OrchestralLily detects that these definitions belong to a movement name Cadenza. We also defined a piece title to print before the score.

Another thing to notice here is that we also include the file "orchestrallily/oly_settings_names.ily". That file contains many instrument and score name definitions for most common instruments and causes them to be printed before each staff in the score.

5.1 The Full Score

```
\version "2.13.17"
\include
  "orchestrallily/oly_settings_fullscore.ily"
\include "music-definitions.ily"
\setCreateMIDI ##t
\setCreatePDF ##t

\createScore #"Cadenza" #'("FullScore")
```



The additional `\setCreateMIDI ##t` line causes a midi file of the score to be created in addition to the PDF file.

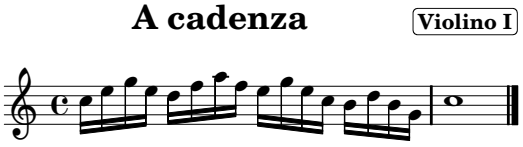
Notice that we never explicitly said that the continuo is supposed to be in bass clef. OrchestralLily already knows that the "BC" (Basso continuo) voice is in bass clef! Similarly, trombone parts will employ the correct C clef, the choir bass will also use the bass clef, etc.

If some instruments should have cue notes, we don't want to print them in the full score, so instead of `\createScore`, OrchestralLily provides the command `\createNoCuesScore`, which will additionally remove all cue notes from the printed score.

5.2 Instrumental Parts

Each instrumental part can be generated just like the full score. If one additionally defines the "instrument" header field, then the instrument name will be printed in the right upper corner, like in most printed scores.

```
\version "2.13.17"
\include "music-definitions.ily"
\include
 "orchestrallily/oly_settings_instrument.ily"
\header { instrument = \VIInstrumentName }

\createScore #"Cadenza" #'("VI")
```



If no music is defined for the given instrument for the desired movement (indicated by the first string that you pass to `createScore`), OrchestralLily will instead print a "tacet" headline. For example, if we try to create a score for the oboe, there is no oboe part defined and a "Cadenza tacet" is printed instead:

```
\version "2.13.17"
\include "music-definitions.ily"
\header { instrument = \ObIInstrumentName }

\createScore #"Cadenza" #'("ObI")
```

A cadenza

Oboe I

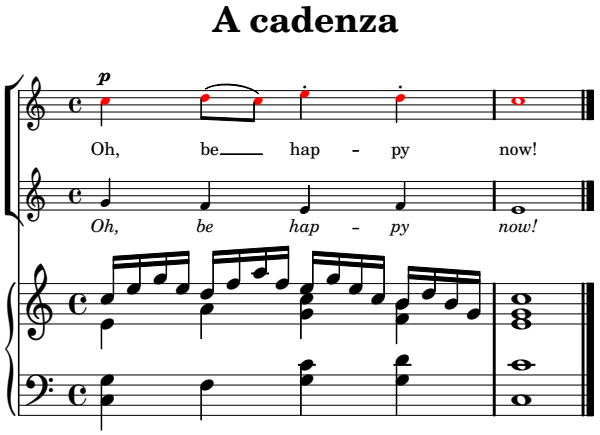
Cadenza tazet

5.3 Vocal Scores and Modifying Individual Staves

To create a vocal score (remember, we have already defined the piano reduction in the definitions!), you only have to call `\createScore` for the "VocalScore" score type. To make things more interesting, here we want the staves for vocal voices to appear smaller than the piano staff. Furthermore, the note heads of the soprano voice should be colored in red and the alto lyrics printed in italic.

These special settings for S and A can be provided by placing them into `\with` blocks and saving them into appropriately named variables called `Cadenza[SA](Staff|Voice|Lyrics)Modifications`:

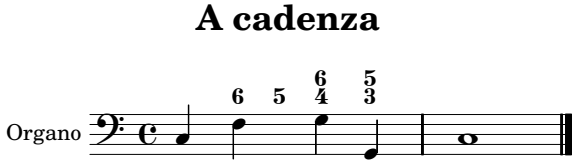
```
\version "2.13.17"
\include "music-definitions.ily"

CadenzaSStaffModifications = \with {
  fontSize = #-3
  \override StaffSymbol #'staff-space =
      #(magstep -3)
}
CadenzaAStaffModifications =
    \CadenzaSStaffModifications
CadenzaChStaffModifications =
    \CadenzaSStaffModifications

CadenzaALyricsModifications = \with {
    \override LyricText #'font-shape =
    #'italic }

CadenzaSVoiceModifications = \with {
    \override NoteHead #'color = #red }

\createScore #"Cadenza" #'("VocalScore")
```



5.4 Figured Bass

The continuo part in the music definitions above is simply the bass line of the cadenza. However, most old scores additionally provide a bass figuration to indicate the harmonies to the organist. Creating such a figured bass score is also extremely simple in OrchestralLily: All you have to do is to define its corresponding variable, named `CadenzaFiguredBassMusic` in our case, where you define the appropriate bass figure in LilyPond's `\figuremode` syntax:

```
\version "2.13.17"
\include "music-definitions.ily"

CadenzaFiguredBassMusic = \figuremode {
  s4 <6>8 <5> <6 4>4 <5 3> | s1
}
\createScore #"Cadenza" #'("Continuo")
```



5.5 Cue Notes

Suppose that we now want to add a second flute, which will set in on the third beat. In the instrumental part, we want to print cue notes from the first flute, but in the full score (or in a combined flute part) we don't want the cue notes.

In LilyPond, one can simply create cue notes by first defining the music to be quoted via `\addQuote` and then inserting the cue notes via `\cueDuring #"quotedInstrument" { r2 }`.

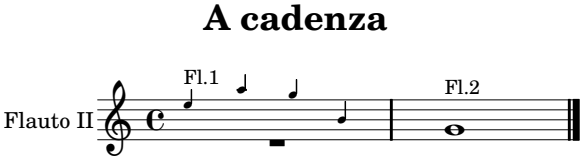
First, we add the new flute 2 part in a separate file "music-definitions-flute2.ily":

```
\addQuote #"Flute1" \CadenzaFlIMusic

CadenzaFlIIMusic = \relative c'' {
  \namedCueDuring #"Flute1" #UP "Fl.1"
      "Fl.2" { R1 } |
  g1 \bar "|."
}
```

Note that we quote the first flute directly in the music for the second flute, using the method `\namedCueDuring` (which is equivalent to LilyPond's built-in function `\cueDuring`, except that it also adds the name of the quoted instrument).

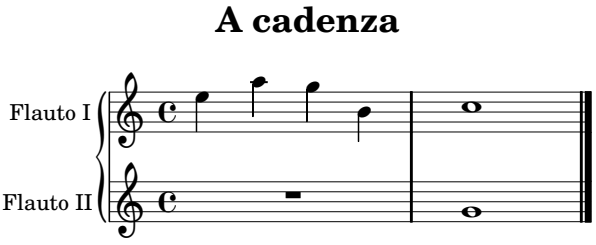
The Flute 2 part now simply is:

```
\version "2.13.17"
\include "music-definitions.ily"
\include "music-definitions-flute2.ily"

% The Flute 2 part:
\createScore #"Cadenza" #'("FlII")
```



In the full score (or a combined flutes part), however, we do not want to print the cue notes, since the notes from Flute 1 are already printed in the score. In this case, we can use `\createNoCuesScore` instead of `\createScore` to suppress the creation of any cue notes:

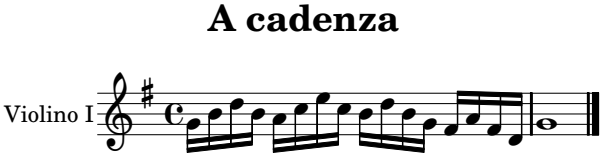
```
\version "2.13.17"
\include "music-definitions.ily"
\include "music-definitions-flute2.ily"

% remove the cues in Flute 2:
\createNoCuesScore #"Cadenza" #'("FlLong")
```



5.6 Transposition

Parts can be easily transposed (e.g. for transposing instruments between concert pitch and written pitch):

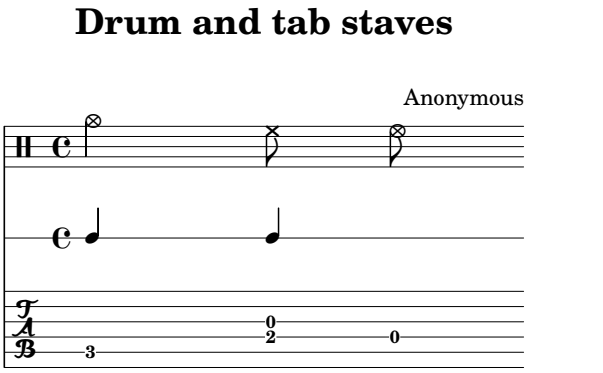
```
\version "2.13.17"
\include "music-definitions.ily"

% We need to give the key explicitly,
% so that it will also be transposed:
CadenzaVIKey = \key c \major
% Transpose to g major
CadenzaVITransposeFrom = g

\createScore #"Cadenza" #'("VI")
```



5.7 Drum Staves and other staff types

Of course, OrchestralLily is also able to print non-standard staves, like rhythmic staves or tablatures:

```
\version "2.13.17"
\include "orchestrallily/orchestrallily.ily"

\header {
  title = "Drum and tab staves"
  composer = "Anonymous"
}

drumIMusic = \drummode { crashcymbal4 hihat8
     halfopenhihat }
drumIIMusic = { c4 c4 }
tabularMusic = { c4 <e g>8 d16 r16 }

\orchestralScoreStructure #'(
  ("drumI" "DrumStaff" ())
  ("drumII" "RhythmicStaff" ())
  ("tabular" "TabStaff" ()))
\orchestralVoiceTypes #'(
  ("drumI" "DrumVoice")
  ("tabular" "TabVoice"))

\createScore #"Cadenza" #'("drumI" "drumII"
    "tabular")
```



6 Tweaking the Score

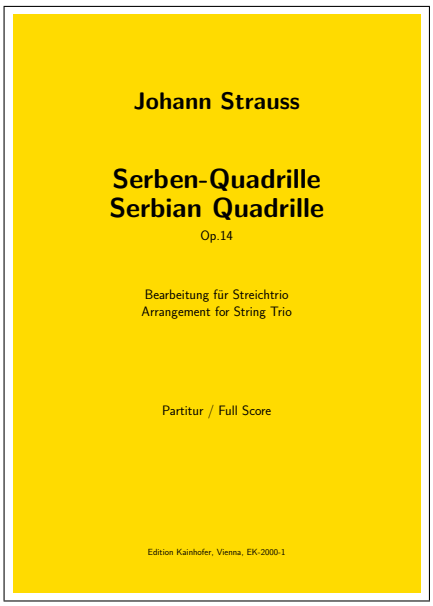
As the OrchestralLily package is implemented entirely in LilyPond syntax and Scheme code, anyone can easily adjust or extend its functionality directly in a score or in an include file, without the need to recompile or reinstall anything.

7 LaTeX for the Preface and Cover

So far, we have concentrated on creating the musical score. A professional edition, however, also features a nice title page, a preface and in many cases also a critical report. For these,

we chose LaTeX for typesetting together with a LaTeX package providing a uniform layout, many macro definitions aiding with the critical report and a beautiful title page. The music scores are directly included into the LaTeX file using the `pdfpages` package.

The templates (see next section) provided by OrchestralLily already produce a beautiful layout without the need for any special tweaks. All one has to do is fill in the missing text and the general information about the score, and the LaTeX scores will look like the following pages from a real score, typeset using OrchestralLily:

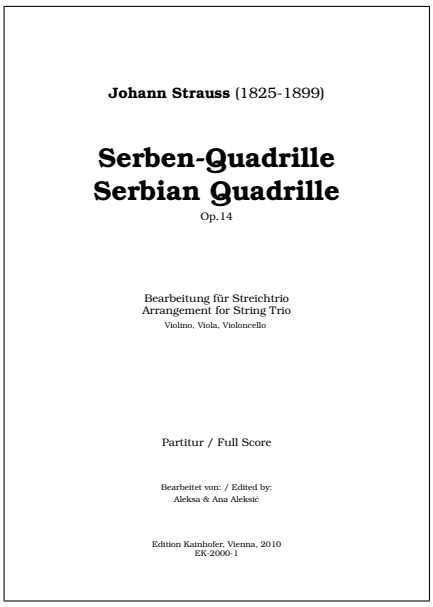

Johann Strauss

Serben-Quadrille
Serbian Quadrille
Op.14

Bearbeitung für Streichtrio
Arrangement for String Trio

Partitur / Full Score

Edition Kainhofer, Vienna, EK-2000-1


Johann Strauss (1825-1899)

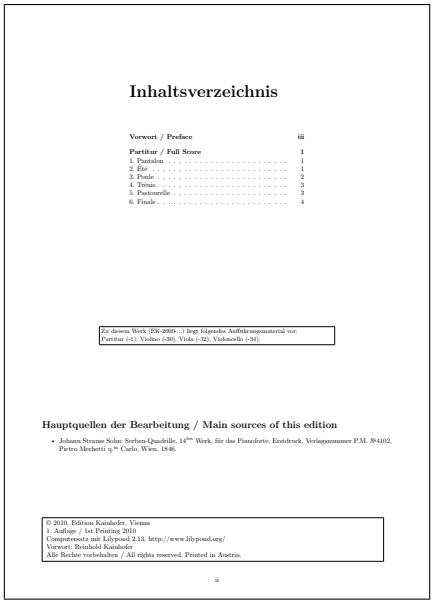
Serben-Quadrille
Serbian Quadrille
Op.14

Bearbeitung für Streichtrio
Arrangement for String Trio
Violino, Viola, Violoncello

Partitur / Full Score

Bearbeitet von: / Edited by:
Aleksa & Ana Aleksić

Edition Kainhofer, Vienna, 2010
EK-2000-1


Inhaltsverzeichnis

Zu diesem Werk (EK-2000-...) liegt folgendes Aufführungsmaterial vor:
Partitur (+1); Violino (+30), Viola (+32), Violoncello (+34).

Hauptquellen der Bearbeitung / Main sources of this edition

- Johann Strauss Sohn: Serben-Quadrille, 14ten Werk, für das Pianoforte, Erstdruck, Verlagsnummer P.M. 30-4102, Pietro Mechetti q.m Carlo, Wien, 1846.

© 2010, Edition Kainhofer, Vienna

1. Auflage / 1st Printing 2010
Computersatz mit Lilypond 2.13, http://www.lilypond.org/
Vorwort: Reinhold Kainhofer

Alle Rechte vorbehalten / All rights reserved. Printed in Austria.


Vorwort

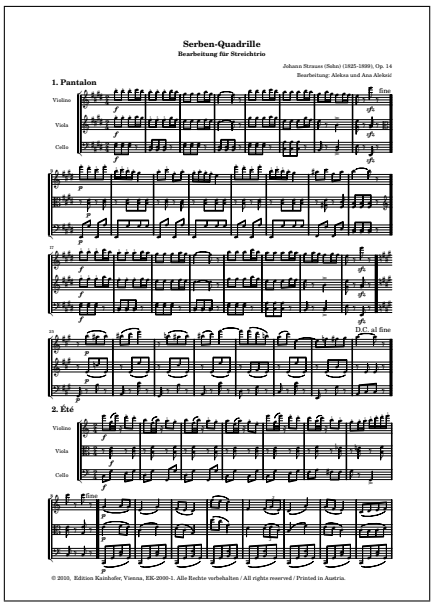
Die Serben-Quadrille ist als Opus 14 ein frühes Werk des Wiener Walzerkönigs Johann Strauss Sohn, komponiert im Auftrag von Fürst Obrenović für einen Slawenball im Jahr 1846 in Wien. In diesem Stück verarbeitet Strauss traditionelle serbische Motive in seinem klassischem Wiener Stil.
Die Rivalität von Johann Strauss Vater und Sohn ab etwa 1845 – der Vater hatte für den Sohn eine under Laufbahn als die eines Musikers vorgesehen, jedoch wurde der Sohn von seiner Mutter entsprechend unterstützt – und die Bekanntheit des Vaters in der Wiener Musikwelt hatten Johann Strauss Sohn gezwungen, sich eher den Randgruppen der gehobenen Wiener Bevölkerung zuzuwenden. Zu dieser Zeit weilte etwa der ehemalige Serbenfürst Miloš Obrenović (1780-1860) im Exil in Wien, nachdem im Jahr 1842 sein Sohn Mihailo Obrenović nach einem Aufstand zu Gunsten von Aleksander Karađorđević als serbischer Fürst abgesetzt worden und nach Wien geflohen war. Obrenović versuchte auf verschiedene Arten wieder entsprechende Rückendeckung die Herrschaft in Serbien zurückzuerlangen (was ihm 1858 schließlich auch gelang), unter anderem durch die Veranstaltung zahlreicher Slawenbälle in Wien. Für den Ball am 28. Januar 1846 im Alt-Wiener Tanzsaal „Zur goldenen Birn" (den Grazmailen im Bezirk Wien-Landstraße) beauftragte er den jungen Johann Strauss mit einer Quadrille, die nicht nur in hoher Stückzahl in Wien verteilt, sondern von Obrenović auch nach Serbien geschickt wurde. Die Aufführung beim Slawenball wurde ein großer Erfolg, ebenso wie die erste öffentliche Aufführung am 2. Februar. Die Zeitung „Gegenwart" schrieb so über den Slawenball: „Strauß's Quadrille wurde mit ungeheurem Beifall (kein Hyperbel!) aufgenommen."
Obwohl die Quadrille beim Ball anscheinend von Johann Strauss' Orchester aufgeführt wurde, erschien sie nur als Klavierausgabe im Druck bei Pietro Mechetti. Eine Orchesterfassung ist – wenn sie je existierte – nicht erhalten.

Die vorliegende Bearbeitung der Serben-Quadrille für Streichtrio entstand ebenfalls anlässlich eines Slawenballs in Wien: Für den Wiener Serbenball im Jahr 2006 sollte das Aleksić Streichtrio zur musikalischen Umrahmung auch die Serben-Quadrille von Johann Strauß darbieten. Mangels einer entsprechenden Bearbeitung entstand – von den Mitgliedern des Streichtrios erstellt – die vorliegende Ausgabe der Quadrille für Streichtrio in der Besetzung Violine, Bratsche und Cello. Die Melodie wird dabei praktisch ausschließlich von der Violine übernommen, die Bratsche unterstützt entweder die Melodie der Violine oder die Harmonik des Cellos.

Quellen: / Sources
[Maiff] Franz Mailer: *Johann Strauß*, Kommentiertes Werkverzeichnis, Verlag Pichler, Wien, 1999.

Preface

The „Serbian Quadrille" Op.14 is an early work of the „Waltz King" Johann Strauss Jr., commissioned by prince Obrenović for a Slavic ball 1846 in Vienna. In it, Strauss uses traditional Serbian motifs in his typical Viennese style.
A quadrille – a precursor of modern square dancing – is a dance formation performed by four couples in square formation. The related music style, also called quadrille, typically consists of six figures in its Viennese style: 1. „Le Pantalon" (a pair of trousers), 2. „L'été" (the summer), 3. „La Ponle" (the hen), 4. „La Trénis" (named after dance master Trenitz), 5. „La Pastourelle" (the shepherd girl) and 6. Finale. The original Fevrach form consisted of only 5 figures, missing the Trénis.
The history of this quadrille carries some interesting aspects as well: The rivalry with his father Johann Strauss Sr., a famous dance music composer as well, who did not want his son to become a musician, and the influence of the father in Vienna had forced the son to turn his attention away from the mainstream of the Viennese high society. At that time, the former Serbian prince Miloš Obrenović (1780-1860) was living in exile in Vienna, after his son Mihailo Obrenović had been deposed in favor of Aleksander Karađorđević. Obrenović tried to regain the support of the Serbian population to return as Serbian prince (where he eventually succeeded in 1858). As one of many ways to gain popularity he organized numerous Slavic balls in Vienna.
For the ball on January 28, 1846, in the dance hall SZur goldenen Birn"(district Wien-Landstraße), he instructed the younger Strauss, with a quadrille, which was distributed not only in large quantities in Vienna, but also sent to Serbia by Obrenović. The performance at the Slavic ball was a great success, as well as the first public performance on February 2.
Although the quadrille appears to have been performed by Strauss' orchestra at the ball, it appeared in print only as a piano edition published by Pietro Mechetti. An orchestral version is not preserved.
The present arrangement of the Serb-Quadrille for string trio was also created for a Slavic ball in Vienna: At the Serbian ball in 2006, the Aleksić String Trio was asked to perform the „Serbian quadrille" by Strauss as part of the musical accompaniment. Lacking a proper arrangement, the members of the string trio decided to write this arrangement for violin, viola and cello themselves. The melody is almost entirely played by the violin, the viola supports either the melody of the violin or the harmony of the cello.


Serben-Quadrille
Bearbeitung für Streichtrio

Johann Strauss (Sohn) (1825-1899), Op. 14
Bearbeitung: Aleksa und Ana Aleksić

1. Pantalon
Violino
Viola
Cello
fine
D.C. al fine

2. Été
Violino
Viola
Cello
fine

© 2010, Edition Kainhofer, Vienna, EK-2000-1. Alle Rechte vorbehalten / All rights reserved / Printed in Austria.

8 Generating a Template

OrchestralLily also provides a template-based script to generate all files required for a full edition of a score: The score information, including instrumentation, movements, voices with lyrics, etc., are defined in one input file. After running the `generate_oly_score.py` script, one has the full set of files for the edition, including a Makefile. Only the music, lyrics, and the actual text of the preface and the critical report need to be filled in. Running `make` will always update all scores and produce ready-for-print files for all desired scores and instrumental parts.

A typical input file for the cadenza example used above is shown here:

```
{
 "output_dir": "Cadenza",
 "version": "2.13.11",
 "template": "EK_Full",

 "defaults": {
   "title": "A test for OrchestralLily",
   "composer": "Reinhold Kainhofer",
   "composerdate": "1977-",

   "year": "2009",
   "publisher": "Edition Kainhofer",
   "scorenumber": "EK-1040",
   "basename": "Cadenza",
   "parts": [
     {"id": "Cadenza", "piece": "A cadenza",
      "piecetacet": "Cadenza tazet"},
   ],

   "instruments": ["FlI", "FlII", "VI", "S",
        "A", "Continuo"],
   "vocalvoices": ["S", "A"],
   "scores": ["Full", "Vocal", "Choral"],
 },

 "scores": ["Cadenza"],
 "latex": {},
}
```

Running this file through the script generates one definitions file for the music definition, LilyPond files for each of the given scores (Full, vocal and choral scores), as well as for each individual instrumental part. Each of the scores will also have a LaTeX file that includes the title page, the preface (including the table of contents, which is exported by LilyPond!), the score and optionally a critical report.

All these files are tied together via a Makefile, so all one needs to do to create a first version is to copy in the music definition and run make.

9 Availability of OrchestralLily

The OrchestralLily package [Kainhofer, 2010b] is currently dual-licensed under the Creative Commons BY-NC 3.0 license [Creative Commons, 2010] as well as under the the GPL v3.0.

Its source code can be found in a public git repository [Kainhofer, 2010a]: `http://repo.or.cz/w/orchestrallily.git`.

More information about the OrchestralLily package can be found in the documentation at its homepage `http://kainhofer.com/orchestrallily/` (which is unfortunately not always kept up to date) or better directly from the source code.

10 Acknowledgements

A project like OrchestralLily would of course never be possible without the help of many people. The developers of LilyPond and of LaTeX – too many to name them explicitly here – made OrchestralLily possible in the first place by providing excellent open source applications for both music and text typesetting. The enormous flexibility and configurability of both applications (including the possibility to modify the internals and implement required features yourself) laid the foundation to turn a small project into a professional music publishing framework.

The cadenza example used throughout this article was originally written by me, until Ana Aleksić pointed out several harmonic shortcomings and helped me rewrite it. Similarly, Manfred Schiebel greatly improved my dilettantish attempts at producing a piano reduction.

References

Creative Commons. 2010. By-nc 3.0 at license. `http://creativecommons.org/licenses/by-nc/3.0/at/`.

Reinhold Kainhofer. 2010a. Git repository of OrchestralLily. `http://repo.or.cz/w/orchestrallily.git`.

Reinhold Kainhofer. 2010b. The OrchestralLily package for lilypond. `http://kainhofer.com/orchestrallily`. LilyPond and LaTeX package for professional music typesetting.

Han-Wen Nienhuys and Jan Nieuwenhuizen et al. 2010. GNU LilyPond. `http://www.lilypond.org/`. The music typesetter of the GNU project.

Term Rewriting Extension for the Faust Programming Language

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University
55099 Mainz, Germany
Dr.Graef@t-online.de

Abstract

This paper discusses a term rewriting extension for the functional signal processing language Faust. The extension equips Faust with a hygienic macro processing facility. Faust macros can be used to define complicated, parameterized block diagrams, and perform arbitrary symbolic manipulations of block diagrams. Thus they make it easier to create elaborate signal processor specifications involving many complicated components.

Keywords

Digital signal processing, Faust, functional programming, macro processing, term rewriting.

1 Introduction

Faust is a functional signal processing language, which is used to develop digital signal processors handling synchronous streams of sample values. It is mostly targeted at audio and music applications at this time. Faust has a formal semantics (based on the lambda calculus and a block diagram algebra) which means that it can be used as a specification language for describing signal proccessors in an implementation-independent way. These specifications are executable, however, and Faust provides sophisticated optimizations and compilation to C++ to turn the specifications into efficient code which can compete with carefully hand-crafted routines. Faust works with an abundance of different platforms and plugin environments such as Jack, LADSPA, VST, Pd, Max and several programming languages, just a recompile is enough to create code for the various architectures. Last but not least, the Faust compiler also provides automatic documentation facilities which produce block diagrams in SVG format and LaTeX documents.

Faust has been discussed at the Linux audio conference and elsewhere on various occasions [3; 1] and is quickly gaining traction in the signal processing community. A description of the formal underpinnings can be found in [2]. Faust is free software distributed under the GPL V2+, see `http://faust.grame.fr`.

This paper reports on the Faust term rewriting extension which equips the language with a kind of hygienic macro facility for specifying complex block diagrams in a more convenient fashion. Macros are defined by rewriting rules and are expanded away at compile time by rewriting terms in the Faust block diagram algebra. The resulting Faust program is then compiled as usual. This facility has been developed by the author in collaboration with Yann Orlarey, the principal author of Faust. It has already been available in recent Faust versions for quite some time but has never been documented anywhere; this paper attempts to fill this gap.

2 Basic Faust Example

Faust works on *sampled signals* which are thought of as functions mapping (discrete) time to sample values. Signals can be constant or time-varying, or they can be *control signals* embodied by special elements which are typically implemented as GUI, MIDI and/or OSC controls, depending on the target architecture. The following listing shows a simple Faust program which implements a sine tone generator:

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

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

process = osc(freq)*vol : panner(pan);
```

The example features three control elements for specifying volume, panning and frequency. The sine signal created by the `osc` function gets multiplied by the volume and then passed through a panner which turns it into a stereo output signal. The `process` function is the "main" function of a Faust program; it denotes the signal processing function realized by the program. In this case, the `process` function has

no arguments and thus the implemented signal processor has no inputs. In general, any Faust function (including `process`) can have an arbitrary number of input and output signals.

3 The Term Rewriting Extension

Faust signal processors are essentially *terms* in the Faust *block diagram algebra* (BDA) which is described elsewhere [2]. Briefly, the BDA consists of algebraic operations (written as infix and postfix operators) which specify various combinations of signal processing functions, in particular:

- `f'` and `f@n` *delay* `f` by one and a given number `n` of samples, respectively.
- `f:g` and `f,g` specify the *serial* and *parallel* composition of two signal processing functions.
- `f<:g` and `f:>g` *split* and *merge* (mix) the outputs of `f` and route them to corresponding inputs of `g`.
- `f~g` combines `f` and `g` to a *loop* with implicit 1-sample delay.

In addition, the usual arithmetic, logical and comparison operators (`+`, `*`, etc.) as well as mathematical functions (`exp`, `sin`, etc.) work on signals in a pointwise fashion. Thus, e.g., `f+g` is the signal obtained by adding each sample of the signals `f` and `g`.

Term rewriting provides us with a means to manipulate these BDA terms in an algebraic fashion at compile time. For instance, the following two rewriting rules define a macro named `fact` which implements the factorial:

```
fact(0) = 1;
fact(n) = n*fact(n-1);
process = fact(3);
```

The last line in this program gives the usual `process` function of a Faust program. The resulting signal processor outputs the constant signal $3! = 6$.

Note that Faust doesn't have a special keyword for denoting macro definitions, instead these are flagged by employing *patterns* on the left-hand side of such a rule, which can be constants (such as the constant `0` in this example), variables (such as `n`) and arbitrary expressions formed with these and the BDA operations. That is, a pattern is an arbitarily complex BDA expression involving variables and constants. At least one of the macro arguments must be a non-trivial pattern (i.e., not simply a variable), otherwise the definition will be taken to be an ordinary function definition. Also note that macro definitions may involve multiple rewriting rules, as in the example above.

Just like Faust's functions are nothing but named lambdas, there are also *anonymous macros* which take the form of a **`case`** expression listing all the argument patterns and the corresponding substitutions. For instance, the above definition of the factorial macro is actually equivalent to:

```
fact = case { (0) => 1; (n) => n*fact(n-1); };
```

4 Macro Evaluation by Rewriting

Rewriting rules are applied by matching them to BDA terms on the right-hand side of function definitions. To these ends, the rules are considered in the order in which they are written in the Faust program, binding variables in the left-hand side of rules to the corresponding values. Evaluation proceeds from left to right, innermost expressions first. Thus, in the above example the term `fact(3)` in the definition of the `process` function will be rewritten to the constant `6`, using the following reduction sequence:

```
fact(3) → 3*fact(3-1) → 3*fact(2)
→ 3*(2*fact(2-1)) → 3*(2*fact(1))
→ 3*(2*1*fact(1-1))) → 3*(2*(1*fact(0)))
→ 3*(2*(1*1))) → 6
```

Note that in this process the Faust compiler also evaluates constant signal expressions such as `3-1` → `2` and `3*(2*(1*1)))` → `6`.

Another example, which employs pattern matching on BDA operations, is the following little macro `serial` which turns parallel compositions into serial ones:

```
serial((x,y)) = serial(x) : serial(y);
serial(x)     = x;
process       = serial((sin,cos,tan));
```

The result is the same as if you had written the serial composition `sin:cos:tan`, cf. Fig. 1.

As the above examples show, macro definitions can also be recursive, making it possible to analyze and build arbitrarily complicated BDA terms. Here is another example, which employs a variation of the `fold` operation (customary in functional programming libraries) to accumulate values. In this case the values are actually signals, produced by a function (or macro) `x` which maps a running index to a signal. This

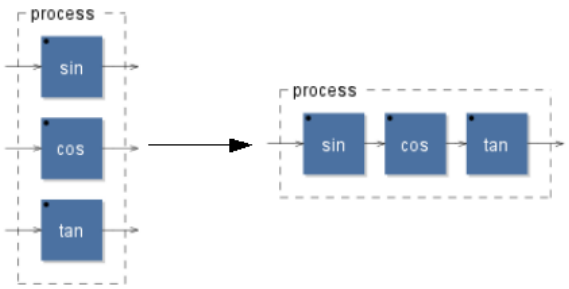


Figure 1: Parallel-serial conversion macro.

allows us to emulate Faust's `sum` function which adds up an arbitrary collection of signals:

```
fold(1,f,x) = x(0);
fold(n,f,x) = f(fold(n-1,f,x),x(n-1));
fsum(n)     = fold(n,+);

f0 = 440; a(0) = 1; a(1) = 0.5; a(2) = 0.3;
h(i)    = a(i)*osc((i+1)*f0);
v       = hslider("vol", 0.3, 0, 1, 0.01);
process = v*fsum(3,h);
```

The resulting signal processor (a simple additive synthesizer which adds up three harmonics), is shown in Fig. 2. Note that the three oscillators and their amplitudes are also defined using rewriting rules in this example.

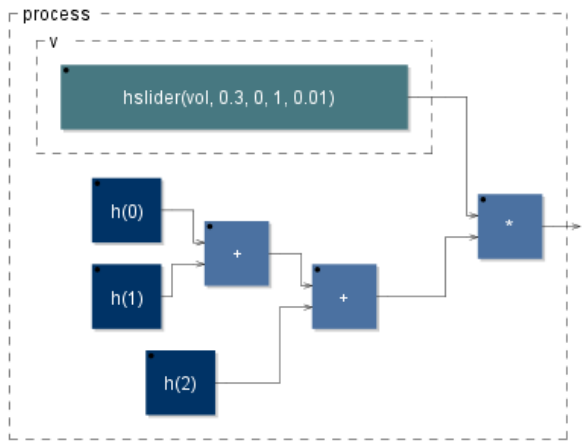


Figure 2: Sum macro example.

5 A Note on Macro Hygiene

It is worth noting the differences between Faust's rewriting-based macros and the textual macros in languages such as C. In the latter case, macros are just textual substitutions which is very flexible but also has various shortcomings, most notably *name capture*. For instance, take the following C macro definition:

```
#define F(x) { int y = x+1; return x+y; }
```

Given this definition, `F(y)` expands to `{ int y = y+1; return y+y; }` which is usually *not* what you want.

Faust macros do not have any such pitfalls since they work on the internal representation of BDA terms rather than the program text. This means that variables on the left-hand side of a macro rule are always bound *lexically*, i.e., according to the block scoping rules defined by the Faust language. Thus a Faust macro like `F = case {(x) => x+y with { y = x+1; };}` (which is roughly equivalent to the C macro above, minus the name capture) will work correctly no matter what gets passed for the macro parameter `x` (even if it is a signal named `y`). Macros with this desirable property are also called *hygienic macros* in the programming language literature.

6 Example: A Systolic Array

The following program illustrates how to use macros in order to abbreviate complicated, parameterized block diagrams. The example we consider is a kind of "systolic array", a grid of binary operations organized in a mesh (Fig. 3).

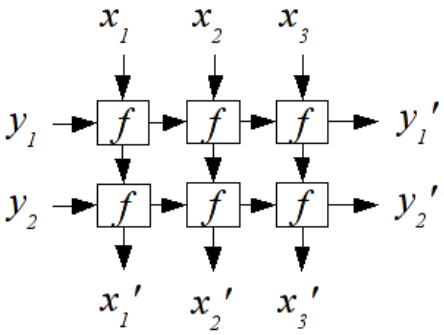


Figure 3: Systolic array.

The Faust program used to produce this layout is given below.

```
g(1,f)  = f;
g(m,f)  = (f, r(m-1)) : (_, g(m-1,f));

h(1,m,f) = g(m,f);
h(n,m,f) = (r(n+m) <: (!,r(n-1),s(m),
            (_,s(n-1),r(m) : g(m,f)))) :
           (h(n-1,m,f), _);

r(1) = _; r(n) = _,r(n-1); // route through
s(1) = !; s(n) = !,s(n-1); // skip

f = + <: _,_; // sample cell function
process = h(2,3,f);
```

Note that the macro `g` constructs a single row of the mesh for the given number m of grid cells and the given function `f` which takes two input signals and produces two output signals. The macro `h` applies `g` repeatedly in order to build an $n \times m$ mesh from its rows. Two helper macros `r` and `s` perform the necessary routing between the components. These employ two basic elements of the BDA, '`_`' which simply routes through a single input to a single output (i.e., `_(x) = x`), and '`!`' which denotes a "sink" for a single input (`!(x) = ()`). (A more detailed explanation of the construction is given below.)

The given `process` function illustrates the use of the `h` macro to construct a 2×3 "accumulator" mesh from the cell function `f = + <: _,_` which just adds its two inputs and sends the computed sum to its two outputs. A Pd patch showing this signal processor in action is shown in Fig. 4.

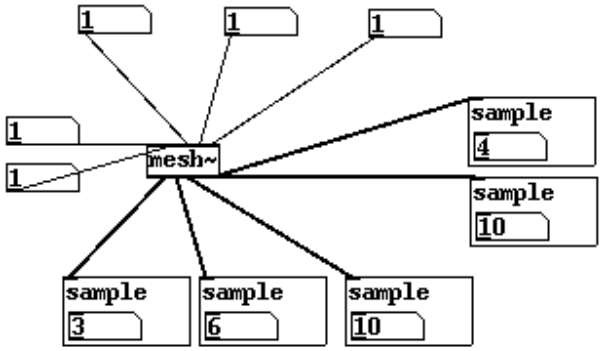


Figure 4: Systolic array example in a Pd patch.

To see how the recursive construction works, note that there are two equations for the `h` macro. The first equation deals with the case of one row. This is handled by just invoking the macro `g` which in turn applies the function `f` once (digesting the single row input y_1 and the first column input x_1) and invokes itself recursively on the first output of `f` and the remaining inputs $x_2, \ldots, x_m$ (which are routed through with the `r` macro). The base case $m = 1$ is treated in the first equation for `g` which just yields `f` itself.

The interesting case is the second equation for `h` in which we deal with more than one row, cf. Fig. 5. Here the input signal, consisting of n row inputs $y_1, \ldots, y_n$ and m column inputs $x_1, \ldots, x_m$, gets split up in two parts. The expression `!,r(n-1),s(m)` gives $y_2, \ldots, y_n$ which is simply routed through. The expression `_,s(n-1),r(m)` yields $y_1, x_1, \ldots, x_m$ which is piped into `g`, producing a single row of the mesh. The signals $y_2, \ldots, y_n$ are then recombined with the first m outputs $x'_1, \ldots, x'_m$ of `g`, and the result is passed to `h` to recursively construct the remaining mesh of $n-1$ rows, yielding the output signals $x''_1, \ldots, x''_m; y'_n, \ldots, y'_2$. Tacking on the remaining output signal y'_1 of the call to `g` gives us the final result $x''_1, \ldots, x''_m; y'_n, \ldots, y'_1$.

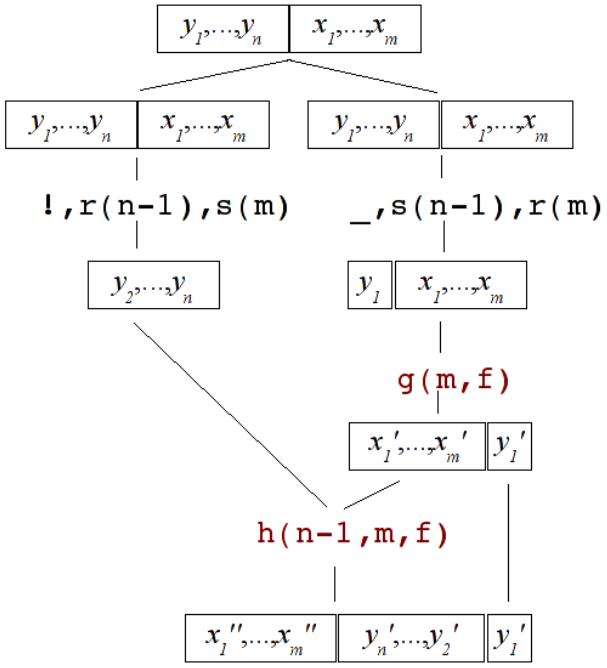


Figure 5: Recursive construction in the systolic array macro.

There are still some ways in which the `h` macro can be improved. For programming convenience and simplicity, `h` takes its inputs as $y_1, \ldots, y_n; x_1, \ldots, x_m$, with the row inputs coming first, and produces the row outputs $y'_n, \ldots, y'_1$ in reverse order. In fact, this layout of arguments and results is quite convenient in the Pd patch. However, as a programmer using this macro in your Faust programs you'd probably prefer a macro which maps an $m+n$ tuple of input signals x, y to a corresponding tuple x', y' of output signals in the "right" order. Fortunately, adding this functionality as separate pre- and postprocessing stages by making good use of the `r` and `s` macros is fairly easy. We leave this as an exercise to the interested reader.

7 Conclusion

The term rewriting extension sketched out in this paper equips Faust with a simple macro processing facility which is useful to define abbreviations for complicated, parameterized block diagrams, and to perform arbitrary symbolic ma-

nipulations on block diagrams in the preprocessing stage of the Faust compiler. To these ends, terms in the Faust block diagram algebra (BDA) are rewritten using term rewriting rules. Evaluating macro invocations using the provided rules is performed by Faust at compilation time. Faust's term rewriting macros are *structured* (they operate on term structures rather than program text) and *hygienic*, i.e., all bindings of macro variables are performed lexically, and thus Faust macros are not susceptible to "name capture" which make less sophisticated macro facilities in languages such as C bug-ridden and hard to use.

There are still some shortcomings in Faust's macro system which will hopefully be addressed in the future:

- Faust does its own normalizations of BDA terms "under the hood" and thus it can be hard to figure out exactly which patterns are needed to rewrite certain constructs of the Faust language.
- Only plain term rewriting rules are supported at this time. Adding conditional (i.e., guarded) rules would make the system more versatile.
- It would be useful to provide an interface to Faust's block diagram optimization pass so that custom optimization rules could be implemented using macros.

References

[1] A. Gräf. Interfacing Pure Data with Faust. In *5th International Linux Audio Conference*, pages 24–31, Berlin, 2007. TU Berlin.

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

[3] Y. Orlarey, A. Gräf, and S. Kersten. DSP programming with Faust, Q and SuperCollider. In *4th International Linux Audio Conference*, pages 39–47, Karlsruhe, 2006. ZKM.

Openmixer: a routing mixer for multichannel studios

Fernando Lopez-Lezcano, Jason Sadural
CCRMA, Stanford University
Stanford, CA, USA
nando@ccrma.stanford.edu, jsadural@ccrma.stanford.edu

Abstract

The Listening Room at CCRMA, Stanford University is a 3D studio with 16 speakers (4 hang from the ceiling, 8 surround the listening area at ear level and 4 more are below an acoustically transparent grid floor). We found that a standard commercial digital mixer was not the best interface for using the studio. Digital mixers are complex, have an opaque interface and they are usually geared towards mix-down to stereo instead of efficiently routing many input and output channels. We have replaced the mixer with a dedicated computer running Openmixer, an open source custom program designed to mix and route many input channels into the multichannel speaker array available in the Listening Room. This paper will describe Openmixer, its motivations, current status and future planned development.

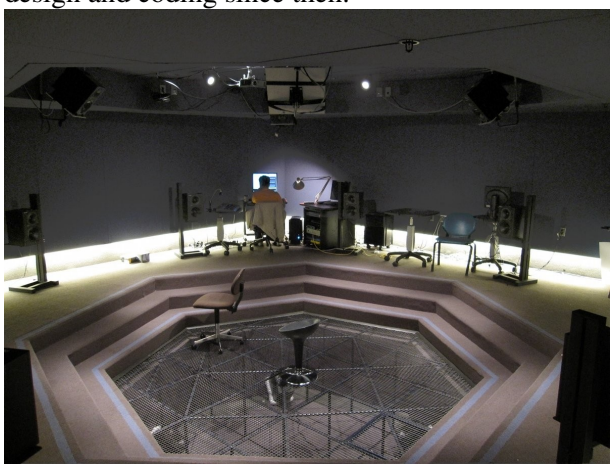
Keywords

Surround sound, studio mixer, spatialization, ambisonics.

1 Introduction

The Listening Room was created when The Knoll (the building that houses CCRMA) was completely remodelled in 2005. It is a very quiet studio with dry acoustics designed for research, composition and diffusion of sounds and music in a full 3D environment. Digital mixers proved to be less than optimal for using the room and the Openmixer project was started to write an application from scratch that would be a better match for our needs.

The need for an alternative solution to digital mixers was recognized by Fernando Lopez-Lezcano in 2006/2007 and he did some research into existing software based alternatives together with Carr Wilkerson at CCRMA. None of the options that were found quite met the goals we had in mind. It was only last year (2009) that the project really started taking off thanks to the enthusiasm and dedication of Jason Sadural (a student at CCRMA) and the support of Jonathan Berger and Chris Chafe. Both Jason and Fernando have been working on hardware and software design and coding since then.



The Listening Room

2 Digital Mixers and the Listening Room

Existing digital mixers are very versatile but are generally not a good fit for a situation that needs many input and output channels. Most of them are designed for mix-down to stereo or at most for mix-down to a fixed 5.1 or 7.1 surround environment. If more output channels are needed expansion slots have to be used and the physical size and price of the mixer goes up very fast. It is also difficult to find mixers with the capabilities we needed and without fans.

Another usability problem of digital mixers is preset management. While it is possible to save and load the mixer state to presets (a feature absolutely necessary in our multi-user shared environment), digital mixers don't have a user database or per-user password protection, at least in mixers that are in our price range. Presets can be

protected but there is nothing to prevent someone from changing other user's presets by mistake.

Digital mixers have opaque interfaces, with options sometimes buried deeply inside a layered menu structure. It is sometimes very hard to know why a particular patch is not working.

We originally installed a Tascam DM3200 in the Listening Room, using two expansion cards to barely provide enough input/output channels (the 3200 has 16 buses and we would like to have a maximum of 24 mixer outputs). In addition to the generic digital mixer problems we found out that switching presets in the 3200 would cause clicks in the analog outputs of the expansion boards that were driving the speakers. How this can happen in a professional mixer is beyond our understanding.

3 Requirements

What would be the ideal sound diffusion system for the Listening Room?

The user interface should be very simple. Nothing should be hidden inside hierarchical menus. All controls should directly affect the diffusion of sound. It should be easy to learn to use it (intuitive). It should be possible to load and save presets using the same authentication system that is used to login in all computers at CCRMA. It should also be possible to control all aspects of the mixer remotely over the network using OSC (Open Sound Control).

The system should support a routing and level control matrix with many multichannel input streams (up to 24 channels wide) coming from different types of sources (analog, digital, network) and many analog outputs (up to a maximum of 24 channels). The system should be able to also deal with multichannel input streams already encoded in Ambisonics, and give the user a choice of predefined decoders calibrated to the speaker array installed in the room.

The system does not need to have audio processing tools such as limiters, compressors, equalization, effects, etc. It should be as transparent as possible. Anything that complicates the user interface should be trimmed down.

It has to be a stand-alone system, always running, and users should not need to login into a computer to start a custom mixer application (for example they should be able to play multichannel content directly from the dvd or blue ray player).

4 A solution

One solution is to use a general purpose computer and off the shelf components as the heart of the mixer, and write a program that orchestrates the process of routing, mixing and decoding the audio streams. A Linux based workstation running Jack [1] and Planet CCRMA is the high performance, low latency and open platform in which we based the design.

Other projects have implemented software mixing and sound diffusion in a general purpose computer, examples include the very capable SuperCollider based software mixer written by Andre Bartetzki [2] at TU Berlin. Or the ICAST [3] sound diffusion system at the Louisiana State University. Or the BEAST system [4] at Birmingham.

Our needs are much more modest and require a system that is based on a single computer that can directly boot into a fully working mixer system.

4.1 The hardware

The Listening Room is a very quiet room with dry acoustics. It is essential it remains quiet so one of our fanless workstations [5] is used as the Openmixer computer. It is a high performance quad core computer with 8Gbytes of RAM and should be able to cope with all reasonable future needs for mixing and processing (see Figure 1 for an overview of the hardware).

4.1.1 Input / output requirements

This is what we need to have available: 16 channels of analog line level balanced input, 8 channels of microphone level input, 8 channels of line level audio input for a media player (DVD/Blu Ray), 3 ADAT lightpipes for direct connection of the audio workstation in the Listening Room (another custom made fanless computer), 3 ADAT lightpipes available for external computers, and dedicated Gigabit Ethernet network jacks with a DHCP server and netjack or jacktrip software for multichannel network audio sources (up to 24 channels per source). All external connections (analog, ADAT, network) should be available in a rack mounted patch panel.

4.1.2 Audio input / output hardware

The core of the audio interface is an RME MADI PCI express soundcard connected through MADI to a Solid State Logic XLogic Alpha-Link MADI

AX box. This combination alone provides for 64 i/o channels from the Openmixer computer including 24 high quality analog inputs and outputs and 3 ADAT lightpipes.

Additional ADAT lightpipes for external computers are provided by an RME RayDAT sound card. This card is currently not installed as we are waiting for the newer implementation of the RME driver for Linux - the existing driver does not support the word clock daughter card that we need to use to synchronize the RayDAT with the main RME MADI card using Word Clock.

The 24 analog outputs are reserved for direct speaker feeds. 16 of the 24 analog inputs are connected to a front panel analog patchbay and the other 8 are connected to the media player. Currently 2 of the 3 ADAT lightpipes go to the audio workstation and 1 ADAT lightpipe is fed from an 8 channel mic preamp with connections to the analog patchbay (this setup will change when the RayDAT card is again part of the system).

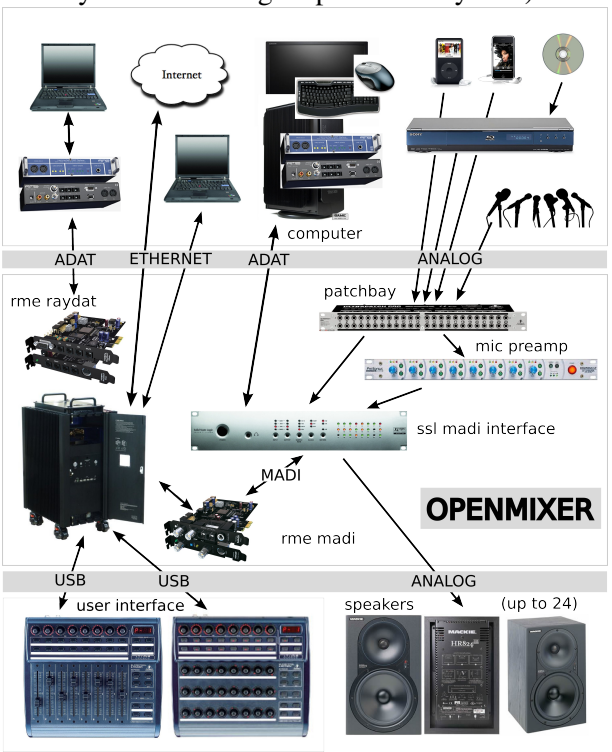


Figure 1: Hardware Overview

The computer has two ethernet interfaces, one connects to the Internet and the other provides DHCP enabled dedicated Gigabit switch ports for laptops or other wired computer systems that connect to Openmixer through the network using either Jacktrip or Netjack.

The RME MADI card is the master clock source for the system, everything else slaves to it through Word Clock or ADAT Sync.

4.1.3 User interface hardware

The only (for now) interface to Openmixer is through a couple of USB connected dedicated MIDI controllers. The current controllers are a Behringer BCF2000 for the main controller and a BCR2000 for the secondary routing controller. Those were selected based on having enough functionality for our needs and being very cheap (and thus easy to replace if they break down). Other more robust controllers might be used in the future. As Openmixer is just software it is easy to switch to a slightly different controller with simple modifications to the source code.

4.2 The software

4.2.1 Choosing a language and environment

Initial proposals included a simple implementation using Pd but we did not go that way because a system based on a text language seemed easier to be debugged and could be easily booted without needing monitors, keyboard or a mouse to interact with it. Another option that was explored was to program Openmixer in C and/or C++ and borrow code from other compatible open source projects or libraries (Openmixer is released under the GPL license). This would have given us complete control at a very low level and would probably be the most efficient implementation, but it would have been more complex to code compared with our final choice. This approach was discarded and simplicity won over efficiency. Another option briefly discussed was to actually use Ardour as the engine for the mixer (after all that is what Ardour does best!), but adapting such a complex project to our needs would have also been very time consuming. In the end we settled for the SuperCollider [6] language as it is text based and can handle most of the needs of the project by itself without the need to use any other software. It can handle audio processing and routing very easily and efficiently, it can send and receive MIDI and OSC messages, and even use the SwingOSC SuperCollider classes for a future GUI display. Of course Jack is used as the audio engine, as well as several support programs that are started and controlled from within SuperCollider (amixer,

jack_connect, jack_lsp, ambdec_cli, jconvolver, etc).

4.3 The user interface

The current interface is layered on top of the capabilities of the BCF2000/BCR2000 controllers. See Figure 2 for an overview of the main controls.

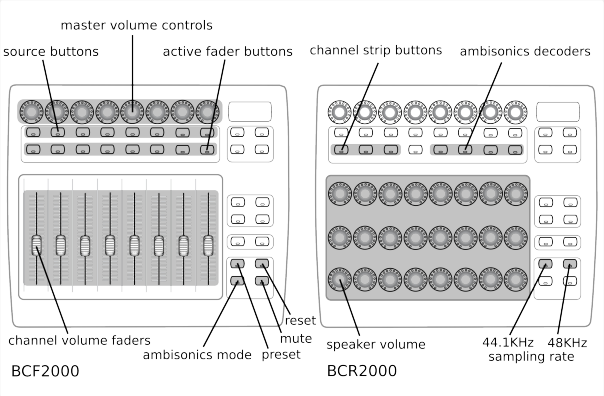


Figure 2: User interface

There are currently two modes of operation already implemented and being used:

4.3.1 Normal Mode

Normal Mode (see Figure 3) routes and controls volume levels of any channel of any input source to a single speaker or a set of speakers. Each input channel from a given input source is associated with a fader in the BCF2000, and can be routed to a single speaker or a set of speakers through the speaker volume knobs in the BCR2000. The levels of a particular input channel can be controlled through the fader {2}, the volume through each speaker can be controlled though the corresponding speaker knob, {3} and a master volume {5} can control all channels from a given source at the same time.

Diffusion of sources is simple to learn and can be done as follows:

1. Press the *Source Button* {1} for the desired input (audio workstation, analog inputs, microphone inputs, media player, network sources, etc).
2. Press one of the *Active Fader Buttons* {2} to activate it, or move the selected fader {2}. The *Active Fader Button* will become lit and the fader can be used to set the overall volume for that channel.

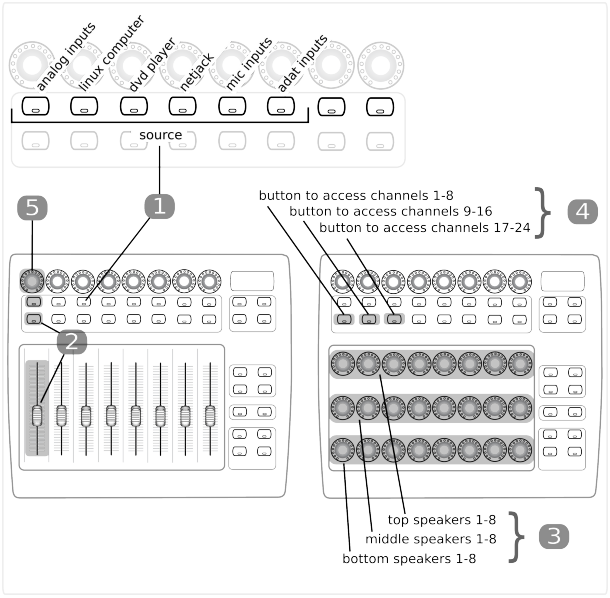


Figure 3: Normal mode

3. Turn up individual *Speaker Volume Knobs* {3} to adjust speakers levels for that channel.
4. Repeat steps 2 and 3 for each input channel. For input sources that have more than 8 channels use the *Channel Strip* buttons {4} in the BCR2000 to access additional input channel faders up to a maximum of 24 input channels for each source (a wider control surface would not need this extra step).
5. Control overall volume for that source with the *Master Volume Control* {5} for that source.

Two keys allow you to set all levels to zero (*Reset*) or load a preset for the selected source (*Preset*) in which individual input channels are pre-routed to individual output speakers.

Two more keys also allow you to select the sampling rate of the whole system (currently only *44.1 KHz* and *48 KHz* are supported).

It should be noted that input sources are not mutually exclusive, they route audio all the time to the speakers. To use more than one at the same time just select several sources separately and adjust the levels appropriately. All faders and controls will reflect the current state of a source after the corresponding source button is pressed.

4.3.2 Ambisonics Decoder Mode

In “Ambisonics Decoder Mode” (see figure 4) the assumption is made that the multi-channel

input source is an Ambisonics encoded audio stream (arrangement of input channels and scaling coefficients are fixed for each source). By pressing the *Ambisonics Mode* button while in a "normal mode" source, all channels of audio from that input source will be directly routed to an Ambisonics decoder with the same level, and from there to the speaker array.

All faders for that source are disabled and set to zero except for fader #1 which is the master volume control for all channels of the encoded input source (if any of the faders besides fader #1 are adjusted, it will automatically be adjusted back to zero after a few seconds).

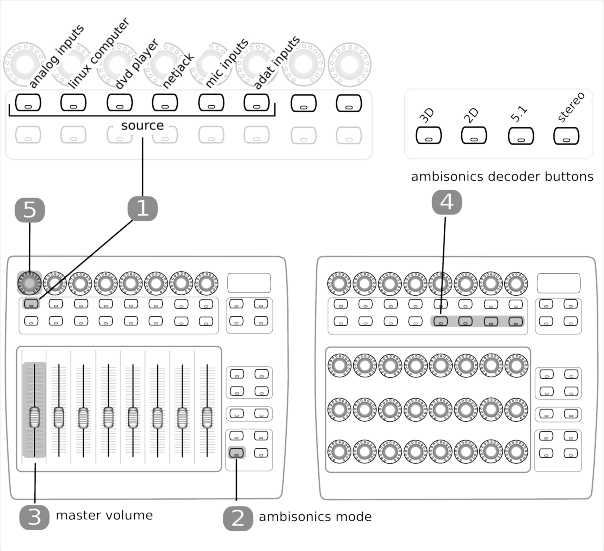


Figure 4: Ambisonics mode

Routing audio to the ambisonics decoders can be done as follows:

1. Select an input source {1} as you would in normal mode.
2. Press the *Ambisonics* decoder button {2} to route that input source directly to the Ambisonics decoders.
3. Adjust fader #1 {3} as the master volume control for the multichannel input source. All other fader or volume knobs are deactivated.
4. Choose the decoder {4} you would like to use.
5. Master mode volume controls {5} are set to unity by default and have exactly the same functionality as in Normal Mode.

4.3.3 Ambisonics Decoders

Openmixer uses *ambdec_cli* [7] and *jconvolver* [8] as external Jack programs that perform the Ambisonics decoding. Several decoders tuned to the speaker array are currently available:

1. 3D 2nd order horizontal, 1st order vertical
2. 2D third order horizontal
3. 2D 5.1 optimized decoder
4. Stereo (UJH decoding done through jconvolver)

This enables the composer or researcher to compare the rendering of the same Ambisonics encoded stream when it is decoded through several different decoders with varying order and capabilites, all properly tuned to the room and speaker arrangement.

When more speakers are added in the future the vertical order of the 3D decoder will be increased.

4.4 Network sources

An extremely simple and efficient way to connect external multichannel sound sources to the mixer is through a network connection. All laptop and desktop computers have a wired interface for high speed network connectivity and it would be perfect to use that for supplying audio to the mixer through the network (only one cable to connect).

There are currently two options:

4.4.1 Jacktrip

The Soundwire [9] project at CCRMA has created a software package (Jacktrip [10]) that is used for multi-machine network performance over the Internet. It supports any number of channels of bidirectional, high quality, uncompressed audio signal streaming. It would be ideal for our purposes except that it currently does not do sample rate synchronization between the computers that are part of a networked performance, so that if there is any clock drift between them there will be periodic clicks when the computers drift more than one buffer apart.

This is not of much concern in a remote network performance with high quality sound cards in both ends, as the quality of the network will usually create more significant problems, but it is crucial in a studio environment where all sources have to be in sync at the sample level (i.e.: no clicks allowed).

Still, Jacktrip is going to be supported with the idea of providing a remote (i.e.: not within the Listening Room) connection for jam sessions and small telepresence concerts.

4.4.2 Netjack

Netjack [11] is an ideal solution for a local network connection as the client computer does not use its own sound card at all (but it is possible to do that locally in the client computer with resampling being used to synchronize the two clocks), but just sends samples to an external Jack sink in the Openmixer computer. It is expected it will quickly become the preferred way to connect laptops and other external computers to the Openmixer system (as long as they can run Jack and Netjack).

The implementation of the Netjack connection in Openmixer was somewhat delayed due to the changing landscape of Netjack. The Openmixer computer is currently running Jack2 which had its own separate Netjack code base - one that relied on automatic discovery of the netjack server through multicast (which we don't currently have enabled at CCRMA). But there is now a backport of the Jack1 implementation of netjack to jack2 (*netone*) that would appear to be perfect for our needs.

The second ethernet interface of the Openmixer computer is connected to a small gigabit switch that provides local network jacks. The computer is set up to provide four DHCP ip addresses on that ethernet port and the firewall is structured to also route traffic to the Internet through the primary ethernet interface (NAT).

Four *jack_netsource* processes are spawned by SuperCollider, each one waiting for its DHCP ip address to become active with a netone jack client. When that happens the audio connection becomes active and Openmixer can control the volume and routing of all channels for that netjack source. For simplicity all potential network connected sources are currently mixed in parallel with equal gains (otherwise the user interface side of Openmixer would start to get more complicated).

5 Structure of the software

The software is written almost exclusively in SuperCollider. A boot time startup script written in perl is triggered when the computer enters into unix run level 5.

The script starts the SuperCollider sclang language interpreter, which takes care of setting up and monitoring the rest of the system. The perl script waits for the interpreter to finish (it should never exit in normal conditions) and restarts it if necessary. This can deal with unexpected problems that could cause SuperCollider's sclang to stop prematurely and it also implements a “system restart” function useful for development and debugging the system (a user can “touch” a file that triggers an orderly shutdown of the SuperCollider portion of Openmixer, and removing that file makes the perl script start it again). The perl script also logs the output of the SuperCollider program for debugging.

Once the SuperCollider sclang interpreter starts, it takes control of the rest of the Openmixer startup process. A number of external programs are used to control the hardware. The mixer of the RME card is initialized using *amixer*, and *jackd* is started with the appropriate parameters (currently jack runs with 2 periods of 128 frames – aproximately 5.3 mSecs of latency - but with the current cpu and load it should be possible to run at 64 frames if the lower latency is deemed necessary). After Jack is up and running two instances of the SuperCollider synthesis server *scsynth* are started (to spread the audio processing load between different cores of the processor), the Ambisonics decoders (*ambdec_cli* and *jconvolver*) are started and finally the *jack_netsource* processes for netjack inputs are spawned. Everything is connected together in the Jack world using calls to *jack_connect*. Finally the SuperCollider MIDI engine is initialized.

Once audio and midi are initialized the SuperCollider Synths that comprise the audio processing section of Openmixer are started and finally the SuperCollider MIDI and OSC responders that tie user controls to actions are defined and started. At this point Openmixer is operational and the user can start interacting with it.

Changing sampling rates is done by shutting down all audio processing and Jack, and restarting everything at the new sampling rate.

Periodic processes check for the presence of the USB MIDI controllers and initializes and reconnects them if necessary (that makes it possible to disconnect them and reconnect them to Openmixer without affecting the system).

Openmixer is currently around 2000 lines of SuperCollider code.

Much needs to be done, some parts of the code need work and reorganizing, and code needs to be added to make the system more reliable. All external programs will be monitored periodically by SuperCollider processes and restarted and reconnected if necessary. Depending on which program malfunctions the audio processing may be interrupted momentarily, but at least the system will recover automatically from malfunctions. The worst offender would be Jack, if it dies the whole system would need to be restarted (as it also happens when, for example, the sampling rate of the system is changed). If SuperCollider itself dies the perl script that starts up the system will restart it. The system is expected to be very stable as the computer in which Openmixer is running is dedicated to that purpose alone (ie: users are not logged in and running random software in it).

6 Future Development

6.1 Encoding to Ambisonics and VBAP

This is a different mode of operation in which individual channels of a given source can be positioned in space through Openmixer instead of being spatialized in the source through a separate program. In this case the second controller (BCR2000) is used to set elevation and azimuth angles in 3D space for each input channel instead of controlling the volume of individual speakers.

This has already been implemented for the case of Ambisonics but we have not yet made it available to end users.

An alternative implementation would do the same thing but using VBAP for spatialization instead of Ambisonics.

All parameters of the spatialization can also be controlled though OSC from a computer connected to the network.

6.2 Start-up sequence

The start-up sequence of the SuperCollider code needs to be changed so that the MIDI subsystem is initialized at the very beginning. That will enable us to use the control surfaces to give feedback to the user while Openmixer starts. Currently there is no way to know how far in the start-up sequence the software is, and when it is available for use (except for the movement of the faders at the very end if their position is different from the default).

7 Future

Saving and recalling presets is a very important feature and is difficult to implement properly. As a stopgap measure Jason has written a very simple OSC responder that enables a user to send OSC messages that save and restore named presets from a common directory. For now the communication is unidirectional so it is not possible to list existing presets, and they can't be protected or locked, but it enables users to save and restore a complex set-up which would otherwise need to be redone from scratch each time a user starts using Openmixer.

Without a display and a keyboard it is difficult to design an interface that is easy to use (because ideally preset management should be tied to the LDAP user database for authentication). One possible option to explore would be to use a web server interface in the Openmixer computer so that preset management can be accessed through a web browser in any computer connected to the network in the Listening Room. A user would need to login with the CCRMA user name and password to be able to access the web site. After that presets would be stored and recalled from a dot subdirectory of the home directory of the user. This would provide adequate functionality without adding a keyboard, mouse and display to the system and without relying of special software in the user's computer.

A GUI with feedback for level metering, state of the mixer, etc., would also be very useful (and could potentially make it easier to code a preset management interface).

Once the code stabilizes we think it would also be very useful to add an Openmixer system to our small concert hall, the Stage.

8 Additional Applications

The fact that Openmixer is just a program enables it to be customized for projects that have very special needs. Here is an example:

A study into archaeological acoustics has led Stanford students into researching and questioning the implications of acoustical environments in a 3000-year old underground acoustic temple in Chavín de Huántar, Perú. A group of students explored the effects of reverberant acoustical features of the temple and ran a series of traditional

social scientific experiments to see what the effects of a sound environment are on an individual's ability to complete different types of tasks. Students used a customized version of Openmixer to connect piezo microphones around an apparatus in which subjects were asked to perform tasks on. The signals were convolved (using *jconvolver* as an external program run from within SuperCollider) with synthetic signals emulating certain 3d aspect of the reverberation and presented to subjects in different spatialized settings during the tasks.

9 Conclusions

So far Openmixer has improved the usability of the Listening Room and has opened the door to interesting ways of interacting with space. In particular it now enables very simple multichannel connection through a network jack. Hopefully it will keep making the Listening Room a productive environment for research and music production.

Openmixer is released under the GPL license, and is available from:

https://ccrma.stanford.edu/software/openmixer

10 Acknowledgements

Many thanks for the support of Chris Chafe and Jonathan Berger at CCRMA for this project. Many many thanks also to the Linux Audio user community for the many very high quality programs that have made OpenMixer possible.

References

[1] Stephane Letz, Jack2, http://www.grame.fr/~letz/jackdmp.html

[2] Andre Bartetzki, LAC2007, *"A Software-based Mixing Desk for Acousmatic Sound Difussion"* (http://www.kgw.tu-berlin.de/~lac2007/descriptions.shtml#bartetzki)

[3] Beck, Patrick, Willkie, Malveaux, , SIGGRAPH2009, *"The Immersive Computer-controlled Audio Sound Theater"*

[4] BEAST (Birmingham ElectroAcoustic Sound Theatre), http://www.beast.bham.ac.uk/about/

[5] Fernando Lopez Lezcano, LAC2009, *"The Quest for Noiseless Computers"*

[6] SuperCollider,
http://www.audiosynth.com/
http://supercollider.sourceforge.net/

[7] Fons Adriaensen, AmbDec, An Ambisonics Decoder, http://www.kokkinizita.net/linuxaudio/

[8] Fons Adriaensen, Jconvolver, A Convolution Engine, http://www.kokkinizita.net/linuxaudio/

[9] The Soundwire Project, https://ccrma.stanford.edu/groups/soundwire/

[10] Jacktrip, http://code.google.com/p/jacktrip/

[11] Netjack, http://netjack.sourceforge.net/

Best Practices for Open Sound Control

Andrew Schmeder and **Adrian Freed** and **David Wessel**
Center for New Music and Audio Technologies (CNMAT), UC Berkeley
1750 Arch Street
Berkeley CA, 94720
USA,
{andy,adrian,wessel}@cnmat.berkeley.edu.edu

Abstract

The structure of the Open Sound Control (OSC) content format is introduced with historical context. The needs for temporal synchronization and dynamic range of audio control data are described in terms of accuracy, precision, bit-depth, bit-rate, and sampling frequency. Specific details are given for the case of instrumental gesture control, spatial audio control and synthesis algorithm control. The consideration of various transport mechanisms used with OSC is discussed for datagram, serial and isochronous modes. A summary of design approaches for describing audio control data is shown, and the case is argued that multi-layered information-rich representations that support multiple strategies for describing semantic structure are necessary.

Keywords

audio control data, signal quality assurance, best practices, open sound control

1 Introduction

1.1 Definition

Open Sound Control (OSC) is a digital media content format for streams of real-time audio control messages. By audio control we mean any time-based information related to an audio stream other than the audio component itself. This definition also separates control data from stream meta-data that is essentially not time-dependent (e.g. [Wright et al., 2000]). Naturally such a format has application outside of audio technology, and OSC has found use in domains such as show control and robotics.

1.2 What is Open

It should be noted that OSC is not a standard as it does not provide any test for certification of conformance. The openness of Open Sound Control is that it has no license requirements, does not require patented algorithms or protected intellectual property, and makes no strong assertions about how the format is to be used in applications.

1.3 Format Structure

OSC streams are sequences of frames defined with respect to a point in time called a timetag. The frames are called bundles. Inside a bundle are some number of messages, each of which represent the state of a sub-stream at the enclosing reference timetag. The sub-streams are labelled with a human-readable character string called an address. In a message, the address is associated with a vector of primitive data types that include common 32-bit binary encodings for integers, real numbers and text (Figure 1).

Unlike audio data, which is sampled regularly on a fixed temporal grid, bundles may contain mixtures of sub-streams that are sampled at different or variable rates. Therefore, the number of messages appearing in a bundle may vary depending on what data is being sampled in that moment.

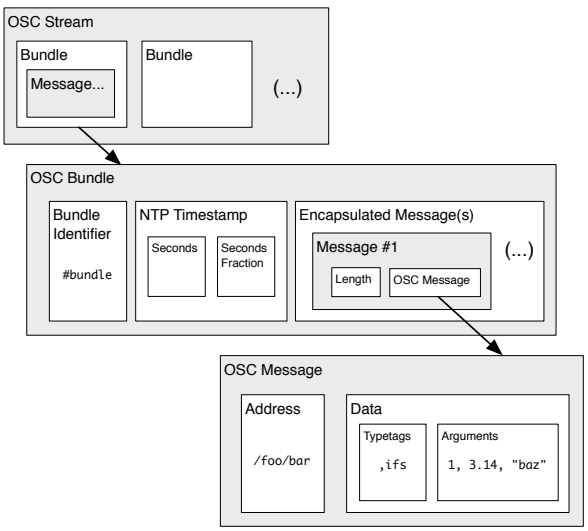


Figure 1: Structure of the OSC content format

1.4 History

OSC was invented in 1997 by Adrian Freed and Matt Wright at The Center for New Music and

Audio Technologies (CNMAT), where its first use was to control sound synthesis algorithms in the CAST system using network messaging (The CNMAT Additive Synthesizer Toolkit) [Adrian Freed, 1997]. CAST was implemented for the SGI Irix platform, which was one of the first general purpose operating systems to provide reliable realtime performance to user-space applications [Freed, 1996] [Cortesi and Thomas, 2001]. This capability was a key influence in the design of the OSC format in particular with respect to the inclusion of timestamped bundles that enable high quality time synchronization of discrete events distributed over a network. Following the success of the CAST messaging system, the protocol was refined and published online as the OSC 1.0 Specification in 2002 [Wright, 2002].

1.5 Best Practices

Considering OSC as a content format alone does not account for how it can and should be used in applications. A larger picture exists around the needs and requirements of sound control in general with respect to the signal quality and description of control data. In the past these needs have been underestimated by hardware and software designs, leading to less than ideal results. Research into the needs of audio control data are summarized in this paper along with recommendations for how to best apply OSC features so that the requirements are satisfied.

1.6 Systems Integration

In the context of the Open System Interconnection Basic Reference Model (OSI Model), OSC is classified as a Layer 6 or Presentation Layer entity.

However in the larger scope of how OSC is used, other layers are considered as part of the practice. Related topics and their associated layer are listed in Table 1.

OSI Layer	Topic in OSC Practice
7 Application	Semantics, Choreography
6 Presentation	OSC Format
5 Session	Enumeration, Discovery
4 Transport	Latency, Reliability
3 Network	Stream Routing
2 Frame	Hardware Clocks, Timing
1 Bit	Cabling, Wireless, Power

Table 1: OSC related topics in the context of associated OSI Model layers

2 Temporal Audio Control

2.1 Instrumental Gestures

Musical instrumental gestures are actuations of a musical instrument by direct human control (typically by kinetic neuro-muscular actuation). The transduction of a physical gesture into a digital representation requires measurement of the temporal trajectory of all relevant dimensions of the physical system such as spatial position, applied force, damping and friction.

An assessment of the performable dynamic range with respect to each physical dimension is beyond the scope of this document, however, in a somewhat general way it is possible to estimate the quantity of temporal information contained in an isolated sub-stream of a musical performance.

2.1.1 Temporal Information Rate

It is estimated that the smallest controllable temporal precision by human kinetic actuation is 1 millisecond (msec), based on an example of an instrumental gesture called the flam that is known to have a very fine temporal structure [Wessel and Wright, 2002]. This limit of 1msec coincides with the threshold for just noticable difference in onset time between two auditory events.

The flam technique in drumming is a method of striking the surface of a drum with two sticks so that the relative arrival time of each stick modulates the timbre (spectral quality) of the resulting sound. Because of the very close temporal proximity of the events, a human listener perceives them to be a single event where the spectral centroid of the timbre is correlated to the temporal fine structure. This inter-onset time can be reliably controlled by a trained performer between 1-10 msec.

The flam is an example of an instrumental gesture with temporal *precision* of 1 msec. However the temporal *accuracy* of instrumental gestures is at least an order of magnitude larger. The tolerable latency for performance of music requiring very tight rhythmic synchronization between two players has been measured to be 10 msec [Chafe et al., 2004]. Ignoring the complications of fixed versus variable delays, it seems reasonable to assume 10 msec as an estimate of temporal accuracy in an instrumental gesture event. And finally, it is also reasonable to suppose that trained musicians can perform event rates up to 10 events per second in a single sub-stream (polyphonic streams are not

considered here).

The temporal information present in musical events can be estimated from these numbers. The information in bits, also called the index of difficulty in Fitt's Law, is calculated from the ratio between effective target distance ρ and standard error of the effective target width σ:

$$I = \log_2 \left(1 + \frac{\rho}{\sigma}\right) \text{bits} \quad (1)$$

Suppose a musician performs the double-strike flam at a rate of 10 hz. This is equivalent to two tasks: 1) placement of events with an average separation of 100 msec and standard error of 10 msec, 2) placement of two sub-events with a separation of 10 msec and error of 1 msec. Assuming the dual-task is repeated at 10 hz then the total information rate is,

$$I' = \log_2 \left(1 + \frac{100}{10}\right) + \log_2 \left(1 + \frac{10}{1}\right) \text{bits}, \quad (2)$$

$$I' \times \frac{10}{\text{sec}} = 68 \frac{\text{bits}}{\text{sec}}. \quad (3)$$

This estimate informs us that if the gesture as described was transformed to a digital representation without loss of information, the temporal dimension alone would require 68 bits/sec to encode.

For sake of comparison the highest reported information transfer rates for target-selection with a mouse in the ISO-1941-9 test are around 3 bits/sec [MacKenzie et al., 2001].

A distinguishing feature of the musical context of gesture is that the human performer uses a combination of extensive training, anticipations of musical structure, and sensory feedback to continuously adjust and refine the gesture. Therefore, musical gestures cannot be directly compared to reaction-time studies or task-based assessments as they are used in the study of human-machine ergonomics. However it is clear from this example that musical gestures contain a far greater density of temporal information than is typical for human-computer interactions in other contexts.

2.2 Spatial Audio Control

Spatial audio effects such as early reflections and reverberation are broad-band temporal-spectral transformations, however the maximum useful rate at which a spatial audio effect can be controlled is limited to the sub-audible frequency band between 0-50 hz. This is due to the simple fact that if a spatial parameter is modulated with a high frequency, perceptual fusion takes place yielding a transformation of the source in some way other than the intended outcome. For example a virtual source with rotating dipole directivity pattern ceases to be perceived as rotating for frequencies above 10 hz [Schmeder, 2009a]. Similarly if the location of a virtual source alternates between two positions at a high rate, the observer perceives a stationary source with a wider apparent source width.

However, spatial audio control data requires very high temporal precision to avoid phase artifacts in multi-loudspeaker arrays. AES11-2003 recommends a between-channel synchronization error of +/- 5% per sample frame [Audio Engineering Society, 2003] [Bouillot and et al, 2009]. An audio signal stream at 96khz requires that the temporal synchronization error does not exceed 0.5 microseconds.

The effect of synchronization error in the control stream for a spatial audio rendering engine may have an impact on the final reproduction quality. For example in a phase-mode beamforming array ([Rafaely, 2005b]), synchronization inaccuracy is similar to a positioning error and synchronization jitter is functionally similar to transducer noise [Rafaely, 2005a].

2.3 Sound Synthesis

In a pure signal processing context audio control data can be considered as the component of the signal that is non-stationary. The representation of control information is a topic specific to the design of any given algorithm, and so it is impossible to state a universal set of requirements for audio control. It is worth noting that some audio synthesis algorithms can have significant bandwidth requirements, especially the data driven methods such as sinusoidal additive synthesis and concatenative synthesis.

3 Temporal Quality Assurance

3.1 Event Synchronization

Assuming clock synchronization is available, the timetag can be used to schedule events with fixed delays that account for the network transport delay in communication between devices. The delays must be known and bounded. This is called forward synchronization and can be efficiently implemented with the priority queue data structure [Schmeder and Freed,

2008] [Brandt and Dannenberg, 1998] (Figure 2).

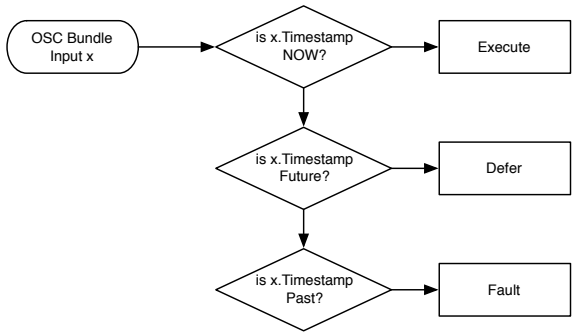


Figure 2: Forward synchronization scheduling for presentation of messages

Applications using OSC timestamps for synchronization should make clear in their documentation what type of clock synchronization is to be used, if any, as well as limits on tolerable network delay.

3.2 Effect of Jitter

Jitter is randomness in time. This may be found in the transport delay, or in the clock synchronization error. Unless it is removed, the effect of jitter on a signal is to corrupt it with noise. The noise is temporal so its magnitude depends on the rate of change of the signal, or its frequency content. Even for relatively low-rate gesture signals, jitter noise can play a significant role. In Figure 3 we see that a 2 msec jitter causes a significant reduction in the channel headroom. The fact that temporal jitter has a strong influence on signal quality is well known in the audio engineering community where a typical sampling converter operating at 96khz requires a clock with jitter measured in the picosecond range.

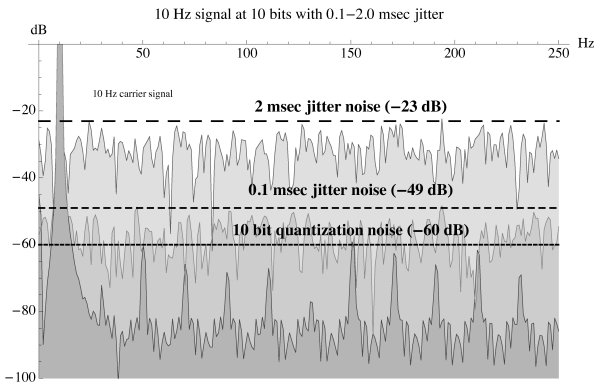


Figure 3: Effective channel headroom after jitter induced noise on a 10hz carrier signal

In Figure 4 we see what combinations of jitter and carrier frequency will degrade a gesture stream with 8-bit dynamic range.

	0.01 msec	0.1 msec	1. msec	2. msec	4. msec
0.5 Hz	100.806	80.942	60.5853	54.4588	48.2834
1. Hz	89.4672	69.2973	49.5129	**42.7719**	**37.1899**
2 Hz	83.5256	64.1865	**44.4936**	**37.811**	**32.166**
4 Hz	77.8606	58.3905	**38.2024**	**32.4498**	**25.4497**
8 Hz	72.3401	52.0053	**31.2989**	**25.7653**	**20.1786**
16 Hz	66.1133	**45.8497**	**25.8291**	**19.7408**	**14.3312**
32 Hz	60.2471	**39.6844**	**19.7202**	**13.546**	**8.26448**
64 Hz	53.9285	**33.8882**	**13.9203**	**7.90135**	**1.7457**

Figure 4: Signal headroom as a function of carrier frequency and standard deviation of delay error. **BOLD** where effective headroom is less than 8-bits dynamic range (8-bits = 48db).

3.3 Jitter Attenuation

From a simple inspection of the table shown, it is apparent that in order to transmit without loss of information an instrumental gesture data stream with frequency content up to 10 hz and 8-bit dynamic range the jitter must be less than 1/10th of a millisecond. Greater dynamic range requires proportionally less jitter where an error reduction by 50% improves dynamic range by 6 dB or 1-bit. To transmit a 10 hz signal with 16-bit dynamic range requires jitter to be less than .5 microseconds.

On contemporary consumer operating systems typical random delays of 1-10 milliseconds between hardware and software interrupts are unacceptably large [Wright et al., 2004], and this source of temporal noise ultimately inhibits the information transmission-rates for real-time control streams. If the data is isochronously sampled it is possible to use filters to smooth jitter [Adriensen and Space, 2005]. However this is not a typical expectation in audio control so something else must be done. If the clock synchronization error is smaller than the transport jitter (which is often the case) and the data stream uses timestamps, then it is possible to use forward synchronization to remove jitter from a control signal (shown in Figure 5).

This operation trades lower jitter for a longer fixed delay. Provided that total delays after rescheduling are less than 10 msec, a satisfactory music performance experience is possible.

3.4 Atomicity

Within each bundle exists a point-in-time sample of a collection of sub-stream messages. The scope over which the data is valid is defined both

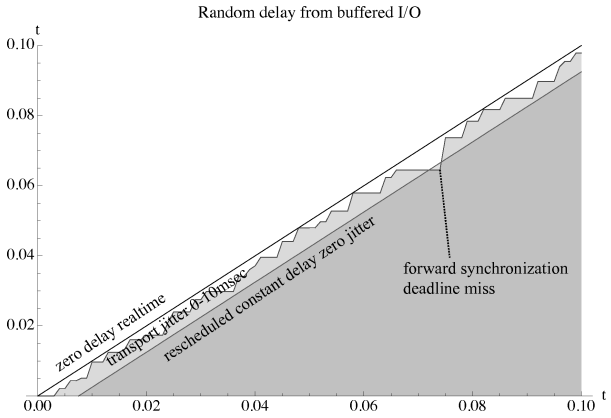


Figure 5: Typical transport jitter of 1-5 msec and its recovery by forward synchronization

with respect to the message addresses as well as some temporal window at the reference timetag.

In implementation practice for application design, message data needs to double-buffered or queued in a FIFO that is updated according to the associated timetag. This prevents unintended temporal skew between sub-streams.

4 Transport Considerations

A common transport protocol used with the OSC format is UDP/IP, but OSC can be encapsulated in any digital communication protocol. The specific features of each transport can affect the quality and availability of stream data at the application layer.

4.1 Datagram Transports

A datagram transport (UDP being a canonical example) is a non-assured transport. Each packet is either delivered in its entirety or not delivered at all. Packets may be out of order, in which case OSC bundle timestamps can be used to recover the correct order. Datagram transports provide a natural encapsulation boundary for each packet. In the case of UDP if the packet exceeds the maximum transmission unit (MTU) the packet may be fragmented over multiple pieces. The fragmentation can introduce extra delay as the UDP/IP stack must then reassemble the pieces before delivering the packet to an application.

4.2 Serial Stream Transport

Serial transports (TCP/IP being an example) provide a continuous data stream between endpoints. The OSC content format does not define a means for representing the beginning and end of a packet. In particular the OSC bundle can contain any number of encapsulated messages and there is no way for a parser to determine the total number until the end of packet is reached. Therefore the major need for sending OSC on a serial transport is that the packets must be encoded with some extra data to indicate where the packet boundaries are, called a framing protocol.

Two options have been proposed for packet framing on serial links. The idea proposed in the OSC 1.0 specification is an integer length-count prefixed on the start of each packet that indicates how many bytes to expect. This encoding requires a totally assured transport such as TCP/IP or USB-Serial. A serial transport with possible errors (such as RS232) will be broken if there is any error in the encoded length.

The SLIP method for framing packets (RFC 1055 [Romkey, 1988]) is an alternative that is robust to transmission errors and stream interruption. In general it is preferred over the former for its simple error recovery.

Assured transports must be used for any mission critical application of OSC and generally this means TCP/IP or something with a similar feature set is needed.

4.3 Isochronous Stream Transport

Isochronous protocols have guaranteed bandwidth, in-order delivery of data, but are not assured (no retries are made on failure). They may or may not provide natural packet boundaries.

Ethernet AVB and the isochronous modes of USB and Firewire are examples of this transport type. Ethernet AVB has additional features in that it also provides a network clock for synchronization and its own timestamp for synchronization of events with total latency as low as 2 msec and synchronization error of less than 0.5 microseconds. Class A streams in the AVB framework are unique among all the transports discussed here in that they are guaranteed to meet or exceed all synchronization and latency requirements needed for audio control data as described in this paper [Marner, 2009].

4.4 File Streams and Databases

For the archival recording and recall of audio control data streams, file systems and databases can be treated as serial stream transports with high block-based jitter in the retrieval phase. By considering a file to be a type of serial stream, OSC can use the same framing protocol

for serial stream encoding as a file format. Again the SLIP method is recommended, and it error recovery features enable robustness against file corruption and truncation.

When a recorded stream of OSC messages is replayed, the original timestamps are rewritten according to a simple linear transformation, and can then be reconstructed temporally using the forward synchronization scheduler. The rewriting of timestamps does not require relative time encodings. Since relative time can always be extracted from absolute time but not the converse, recording of OSC streams for archival purposes should always use absolute time values.

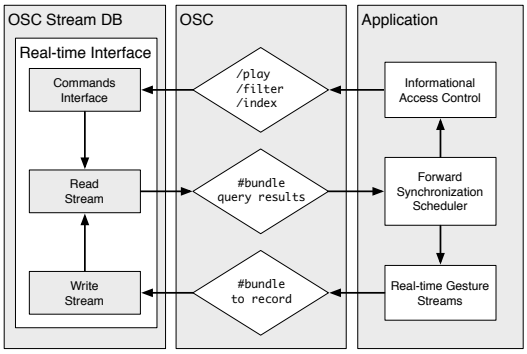


Figure 6: Multi-stream recording and playback interface to a database

A multi-stream approach for interfacing OSC streams to a database and efficient queries over the recorded data is demonstrated in the OSC-StreamDB project [Schmeder, 2009b] (Figure 6).

4.5 Bandwidth Constrained Transports

When bandwidth constrained transports are required such as wireless radios, OSC may be modified with some effort to enable lower bit-rates. Depending on the nature of the data stream, adaptive sampling dependent on the information-rate can be used to reduce the number of messages on the network. Interpolation between frames can be used to recover a smooth control signal in reconstruction. This is likely to work well for instrumental gesture data as the actual effective number of new bits of information per second is relatively low.

Another major source of bit-sparsity in OSC streams is the message address field. Because it is typically a human-readable string, and English text has a bit rate of 1-1.5 bits per character, about 80% of the bits are redundant. A dictionary-type compression scheme could be used to compress the address strings if necessary.

4.6 Network Topology and Routing

The OSC 1.0 Specification included some language regarding OSC client and server endpoints. In fact this distinction is not necessary and OSC may be used on unidirectional transports, and more complex network topologies including multicast, broadcast and peer-to-peer. The OSCgroups project provides a method for achieving multicast routing and automatic NAT traversal on the internet [Bencina, 2009].

A shortcoming of many current OSC implementations using UDP/IP is missing support for bidirectional messaging. As a best practice implementations should try to leverage as much information as the transport layer can provide, as this makes more simple the task of configuration of the endpoint addresses in applications as well as stateful inspection at lower layers.

5 Describing Control Data

The address field in an OSC message is where descriptive information is placed to identify the semantics of the message data. The set of all possible addresses within an application is called an address space.

5.1 Descriptive Styles

Existing OSC practice includes a wide variety of strategies for structuring address spaces. Here we intend to clarify the differences between the styles rather than to promote any particular method as preferred. Four common styles have emerged over decades of software engineering practice: RPC, REST, OOP and RDF. Examples of OSC messages in each style accomplishing the same task (setting the gain on a channel) are given here.

5.1.1 RPC

The RPC (Remote Procedure Call) style employes functional enumeration, and lends itself to small address spaces of fixed functions:

```
/setgain (channel number = 3) (gain value = x)
```

5.1.2 REST

The REST (Representational State Transfer) style encourages state-machine free representations by transmitting the entire state of an entity in each transaction. Web application programmers are familiar with this style wherein every time a page is loaded, the application has two phases: setup (recreating the entire application state from the transferred representation)

and teardown (throwing it all away). The state-free property is what enables web-browsers to always pages outside the context of a browsing session by recalling a bookmark.

OSC address spaces using the REST style of resource enumeration have a familiar appearance since it is the most common style used in the construction of hyperlink addresses on the web.

```
/channel/3/gain (x)
```

5.1.3 OOP

The OOP (Object Oriented Programming) style is based on an intuitive concept of objects that are self-contained entities containing both attributes and specialized functions called methods that are procedures transforming their own attributes.

```
/channel/3@gain (x)
/channel/3/setgain (x)
```

OOP enables abstraction and layering in large systems. It may also need notations beyond the basic '/' delimiter used in path-style addresses since the OOP structure requires differentiation of the object, attribute and method entity types. In the above example we have used '@' following the XPath notation to indicate an attribute. In Jamoma a ':' character is used to similar effect [Place et al., 2008].

5.1.4 RDF

The RDF (Resource Description Framework) style employs ontological tagging to describe data with arbitrarily complex grammars. This style of control is the most powerful of the alternatives shown here, however its use also requires greater verbosity since it makes no semantic assumptions about the data structure.

The interpretation of the delimiter '/' in OSC as a hierarchical containment operator as it implies in the REST and OOP paradigms is not used in this style. Instead it is interpreted as an unordered delimiter between tags, and each tag is a comma-separated triple of subject, predicate, and object entities.

```
/channel,num,3
  /op,is,set
  /lvalue,is,gain
  /rvalue,units,dB (x)
```

5.2 Leveraging Pattern Matching

In OSC there a type of query operator called address pattern matching. Similar to the use of wildcard operators in command-line file system operations, patterns enable one-to-many mappings between patterns and groups of messages. However this technique is only useful if the target messages have a structure that enables the provided pattern syntax to make useful groupings. This structure is usually present when the address space follows the REST design paradigm. Designers of address spaces for applications should consider how the resulting addresses might make use of grouped-control by patterns.

```
/channel/*/gain (common gain value x)
```

Some effort is needed to retain efficiency of query operators in very large address spaces. This is possible using database structures such as the RDTree [Schmeder, 2009b].

5.3 Problems with Stateful Encodings

A stateful encoding of a control data stream is one where the meaning of a message has some dependence on a previously transmitted message. The interpretation of the message by the receiver requires some memory of previously received messages in addition to the necessary logic to correctly fuse the information. This logic is typically a finite state machine, although in general it can be more complex.

Suppose that there is a switch, called `/button`, with two possible states, off or on, represented by the numbers 0 and 1 respectively. A designer wishing to conserve network bandwidth decides only to transmit a message when the switch changes from one state to the other and so sends the number +1 to indicate a transition from 0 to 1 and the number -1 to indicate a transition from 1 to 0.

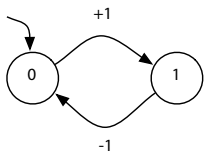


Figure 7: Finite state machine for parsing the transitions between a two states

The following is a valid sequence of messages that can be verified at by a receiver using the finite state machine shown in Figure 7.

```
/button +1
/button -1
/button +1
/button -1
```

A potential problem with this representation is that it is not robust to any errors in the

transmission of the data. Suppose that a message is lost due to the use of a non-assured transport such as UDP/IP, the sequence is then:

```
/button +1
/button +1
/button -1
```

In this example it is possible to make a more complex state machine that is capable of of recovering from the missing data, but we can immediately see that the complexity of the program has doubled (Figure 8).

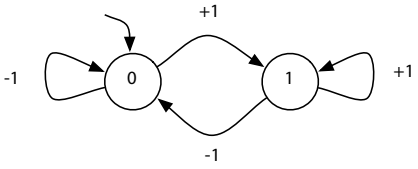


Figure 8: Finite state machine for parsing the transitions between a two states with error recovery logic

An alternative solution eliminates the need for a parsing engine entirely, by simply transferring the entire state of the switch in every message.

```
/button 0
/button 0
/button 0
/button 1
/button 1
/button 0
...
```

Furthermore, if these messages are continuously transmitted even when the state does not change, then the receiver needs to make no special effort to recover from a missed message. While this example is contrived to the point of being trivial, it does demonstrate that stateful encodings require more complex programs especially in the case of error handling. On modern network transports with typical bandwidth capacity of 1Gbits/sec, the simplicity of state-free representations is often more valuable than saving network bandwidth.

5.4 Layering Control Data

Between the source user action performed on a human input device to the high level application control stream, there are several intermediate layers of representation [Follmer et al., 2009] (Figure 9). The OSC format can be used at every layer where a digital representation of the data stream is present. Even though such streams may ultimately be not used in high level abstractions it is useful to retain the OSC address labels at each step.

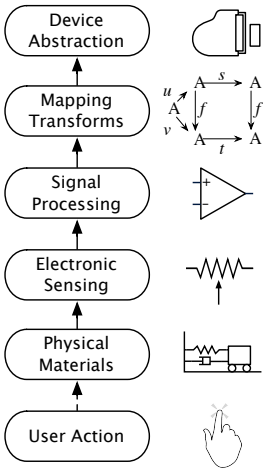


Figure 9: Intermediate layers of representation between user interface controller and an application

5.4.1 Complications of Mapping

The use of transformational mapping of gesture data is an important aspect in the design of interactive musical systems [Hunt et al., 2003]. In many cases useful mapping transformations carry out some type of information fusion that is a non-linear transformation of the data. However non-linear transformations require extra attention because they transform uniform noise into a non-uniform noise with a complicated spatial structure. It is possible to design adaptive filters that are optimal for a non-linear transformation, however this requires a more complex processing graph than what is shown in Figure 9.

Consider the non-linear transformation function,

$$f(x, y) = \frac{x - y}{x + y}. \tag{4}$$

If x and y are corrupted with any noise (which is inevitable) then the transformed variable $f(x, y)$ will greatly amplify that noise when x and y approach zero. This is evident from the fact that its derivative is unbounded as $(x, y) \to (0, 0)$.

$$\partial_{x,y} f(x, y) = -\frac{x - y}{(x + y)^2} \pm \frac{1}{x + y} \tag{5}$$

Therefore thresholding (outlier rejection) and noise filtering must take place *after* the mapping transform, even though they are ostensibly signal processing layer operations (see Figure 10). In other words, the layer model of Figure 9 is not entirely correct as the layers are not strictly ordered. To enable out-of-order process-

ing across layers, designers should retain versions of data streams before and after mapping transformations under different address labels.

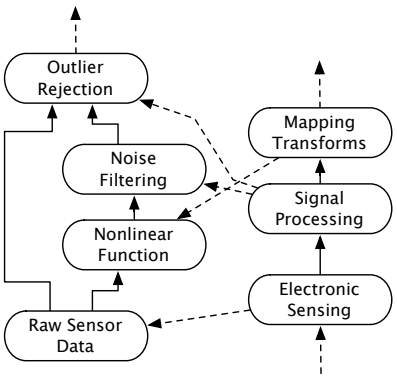


Figure 10: Cross-layer dependencies exist in the processing chain for input control data.

6 Conclusion

Here we present a brief summary of each point made in the document.

6.1 Transport and Synchronization

6.1.1 Instrumental Gesture Control

For data streams from measurement of human kinetic gestures, the temporal synchronization error should be not more than .1 milliseconds to ensure lossless transmission of periodic signals up to 10 hz with 8 bit dynamic range, with each extra bit of dynamic range requiring half the temporal error (50 usec for 9 bits, 25 usec for 10 bits, etc.). To resolve events with 1 msec relative temporal precision the sampling frequency of measurement should be 2000 hz.

6.1.2 Spatial Audio Control

The transport should be capable of updating the spatial audio parameters at rates of 100hz in order to resolve the full range of perceivable spatial effects that may extend up to 50 hz.

For spatial audio control data used in beamforming or wavefield synthesis rendering, following the AES recommended limits the synchronization error should be less than 5% of a sample frame for the highest controlled frequency. For example control over coefficients in a phase-mode beamforming array operating up to 10 khz requires a synchronization accuracy of 5 microseconds.

6.1.3 Digital Audio Synthesis Control

For audio synthesis algorithms in general the needs of control data are dependent on the nature of the algorithm and may vary widely depending on the level of detail and control bandwidth. Except for perhaps the most esoteric applications, a 0.5 microsecond temporal precision is sufficient for control of any audio synthesis algorithm.

6.1.4 Atomicity

In all cases careful use of double-buffering, lock-free queues and local memory-barrier operations should be used to ensure a best-effort is made for minimizing the synchronization skew between bundle-encapsulated sub-stream messages.

6.1.5 Latency and Jitter

The latency and jitter of secondary software interrupts typical of for human-input device streams are detrimental to the quality of control data. Bounded latency and minimal jitter should be ensured for audio control data.

6.2 Control Meta-data

6.2.1 Interface Design Patterns

Many styles of meta-data description are possible including procedural (RPC), resource-oriented (REST), object-oriented (OOP) and ontology-oriented (RDF). Application designers should feel free to choose the most appropriate style, however designers creating generic tools for OSC processing should support as many styles as possible.

6.2.2 Stateful Representation

When stateful representations of control data streams are used (with care), then assured transports should also be employed to reduce software errors that may be triggered by transmission errors.

6.2.3 Multi-Layered Representation

Retaining data stream representations at multiple levels of abstraction is useful as some operations on control data streams cannot be performed in a strictly sequential order. The general process structure for transformation of control data is a directed graph.

7 Acknowledgements

We are grateful to Meyer Sound Laboratories Inc. of Berkeley CA for financial support of this work. The anonymous reviewers provided helpful suggestions to improve this document. The users of OSC in the community including computer music application designers and researchers have played an important role in bringing to light important issues related to audio control.

References

Matthew Wright Adrian Freed. 1997. CAST: CNMAT Additive Synthesis Tools. `http://archive.cnmat.berkeley.edu/CAST/`.

Fons Adriensen and A Space. 2005. Using a DLL to Filter Time. In *Proceedings of the Linux Audio Conference*.

Audio Engineering Society. 2003. AES Recommended Practice for Digital Audio Engineering - Synchronization of Digital Audio Equipment in Studio Operations. Technical Report 11, AES.

Ross Bencina. 2009. OSCgroups. `http://www.audiomulch.com/~rossb/code/oscgroups/`.

Nicolas Bouillot and et al. 2009. *AES White Paper: Best Practices in Network Audio*, volume 57 of *JAES*. AES.

Eli Brandt and Roger Dannenberg. 1998. Time in Distributed Real-Time Systems. In *Proceedings of the ICMC*, pages 523–526, San Francisco, CA.

Chris Chafe, Michael Gurevich, Grace Leslie, and Sean Tyan. 2004. Effect of time delay on ensemble accuracy. In *In Proceedings of the International Symposium on Musical Acoustics*.

David Cortesi and Susan Thomas. 2001. *REACT™ Real-Time Programmer's Guide*. Number Document Number 007-2499-011. Silicon Graphics, Inc.

Sean Follmer, Björn Hartmann, and Pat Hanrahan. 2009. Input Devices are like Onions: A Layered Framework for Guiding Device Designers. In *Workshop of CHI*.

Adrian Freed. 1996. Audio I/O Programming on SGI Irix. `http://cnmat.berkeley.edu/node/8775`.

Andy Hunt, Marcelo M. Wanderley, and Matthew Paradis. 2003. The Importance of Parameter Mapping in Electronic Instrument Design. *Journal of New Music Research*, 32(4):429–440, December.

I. Scott MacKenzie, Tatu Kauppinen, and Miika Silfverberg. 2001. Accuracy measures for evaluating computer pointing devices. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 9–16, New York, NY, USA. ACM.

Geoffrey M Marner. 2009. Time Stamp Accuracy needed by IEEE 802.1AS. Technical report, IEEE 802.1 AVB TG.

Timothy Place, Trond Lossius, Alexander Jensenius, Nils Peters, and Pascal Baltazar. 2008. Addressing Classes by Differentiating Values and Properties in OSC. In *NIME*.

B. Rafaely. 2005a. Analysis and design of spherical microphone arrays. *Speech and Audio Processing, IEEE Transactions on*, 13(1):135–143, Jan.

B. Rafaely. 2005b. Phase-mode versus delay-and-sum spherical microphone array processing. *Signal Processing Letters, IEEE*, 12(10):713–716, Oct.

J. Romkey. 1988. RFC1055 - Nonstandard for transmission of IP datagrams over seria lines: SLIP. Technical report, IETF.

Andy Schmeder and Adrian Freed. 2008. Implementation and Applications of Open Sound Control Timestamps. In *Proceedings of the ICMC*, pages 655–658, Belfast, UK. ICMA.

Andrew Schmeder. 2009a. An Exploration of Design Parameters for Human-Interactive Systems with Compact Spherical Loudspeaker Arrays. In *Ambisonics Symposium*.

Andrew Schmeder. 2009b. Efficient Gesture Storage and Retrieval for Multiple Applications using a Relational Data Model of Open Sound Control. In *Proceedings of the ICMC*.

David Wessel and Matthew Wright. 2002. Problems and Prospects for Intimate Musical Control of Computers. *Computer Music Journal*, 26:11–22.

Matthew Wright, Amar Chaudhary, Adrian Freed, Sami Khoury, David Wessel, and Ali Momeni. 2000. An xml-based sdif stream relationships language. In *International Computer Music Conference*, pages 186–189, Berlin, Germany. International Computer Music Association.

Matthew Wright, Ryan Cassidy, and Michael Zbyszynski. 2004. Audio and Gesture Latency Measurements on Linux and OSX. In *Proceedings of the ICMC*, pages 423–429.

Matthew Wright. 2002. Open Sound Control 1.0 Specification. `http://opensoundcontrol.org/spec-1_0`.

supernova, a multiprocessor-aware synthesis server for SuperCollider

Tim BLECHMANN
Vienna, Austria
tim@klingt.org

Abstract

SuperCollider [McCartney, 1996] is a modular computer music system, based on an object-oriented real-time scripting language and a standalone synthesis server. supernova is a new implementation of the SuperCollider synthesis server, providing an extension for multi-threaded signal processing. With adding one class to the SuperCollider class library, the parallel signal processing capabilities are exposed to the user.

Keywords

SuperCollider, multi-processor, real-time

1 Introduction

In the last few years, multi- and many-core computer architectures nearly replaced single-core CPUs. Increasing the single-core performance requires a higher power consumption, which would lower the performance per watt. The computer industry therefore concentrated on increasing the number of CPU cores instead of single-core performance. These days, most mobile computers use dual-core processors, while workstations use up to quad-core processors with support for Simultaneous Multithreading.

Most computer-music applications still use a single-threaded programming model for the signal processing. Although multi-processor aware signal processing was pioneered in the late 1980s, most notably with IRCAM's ISPW [Puckette, 1991], most systems, that are commonly used these days, have limited support for parallel signal processing. Recently, some systems introduced limited multi-processor capabilities, e.g. PureData, using a static pipelining technique similar to the ISPW [Puckette, 2008], adding a delay of one block size (usually 64 samples) to transfer data between processors. Jack2 [Letz et al., 2005] can execute different clients in parallel, so one can manually distribute the signal processing load to different applications. It can also be used to control multiple SuperCollider server instances from the same language process.

This paper is divided into the following sections. Section 2 gives an introduction to the difficulties when parallelizing computer music applications, Section 3 describes the general architecture of SuperCollider and the programming model of a SuperCollider signal graph. Section 4 introduces the concept of 'parallel groups' to provide multi-processor support for SuperCollider signal graphs. Section 5 gives an overview of the supernova architecture, Section 6 describes the current limitations.

2 Parallelizing Computer Music Systems

Parallelizing computer music systems is not a trivial task, because of several constraints.

2.1 Signal Graphs

In general, signal graphs are data-flow graphs, a form of directed acyclic graphs (DAGs), where each node does some signal processing, based on its predecessors. When trying to parallelize a signal graph, two aspects have to be considered. Signal graphs may have a huge number of nodes, easily reaching several thousand nodes. Traversing a huge graph in parallel is possible, but since the nodes are usually small building blocks ('unit generators'), processing only a few samples, the synchronization overhead would outweigh the speedup of parallelizing the traversal. To cope with the scheduling overhead, Ulrich Reiter and Andreas Partzsch developed an algorithm for clustering a huge node graph for a certain number of threads [Reiter and Partzsch, 2007].

While in theory the only ordering constraint between graph nodes is the graph order (**explicit order**), many computer music systems introduce an **implicit order**, which is defined by the order, in which nodes access **shared resources**. This implicit order changes

with the algorithm, that is used for traversing the graph and is undefined for a parallel graph traversal. Implicit ordering constraints make it especially difficult to parallelize signal graphs of max-like [Puckette, 2002] computer music systems.

2.2 Realtime Constraints

Computer Music Systems are realtime systems with low-latency constraints. The most commonly used audio driver APIs use a 'Pull Model'[1], that means, the driver calls a certain callback function at a monotonic rate, when the audio device provides and needs new data. The audio callback is a realtime function. If it doesn't meet the deadline and the audio data is delivered too late, a buffer under-run occurs, resulting in an audio dropout. The deadline itself depends on the driver settings, especially on the block size, which is typically between 64 and 2048 samples (roughly between 1 and 80 milliseconds). For low-latency applications like computer-based instruments playback latency below 20 ms are desired [Magnusson, 2006], for processing percussion instruments round-trip latencies for below 10 ms are required to ensure that the sound is perceived as single event. Additional latencies may be introduced by buffering in hardware and digital/analog conversion.

In order to match the low-latency real-time constraints, a number of issues has to be dealt with. Most importantly, it is not allowed to block the realtime thread, which could happen when using blocking data structures for synchronization or allocating memory from the operating system.

3 SuperCollider

supernova is designed to be integrated in the SuperCollider system. This Section gives an overview about the parts of SuperCollider's architecture, that are relevant for parallel signal processing.

3.1 SuperCollider Architecture

SuperCollider in its current version 3 [McCartney, 2002] is designed as a very modular system, consisting of the following parts (compare Figure 3.1):

Language (sclang) SuperCollider is based on an object-oriented scripting language with a real-time garbage collector, which is inspired by Smalltalk. The language comes with a huge class library, specifically designed for computer music applications. It includes classes to control the synthesis server from the language.

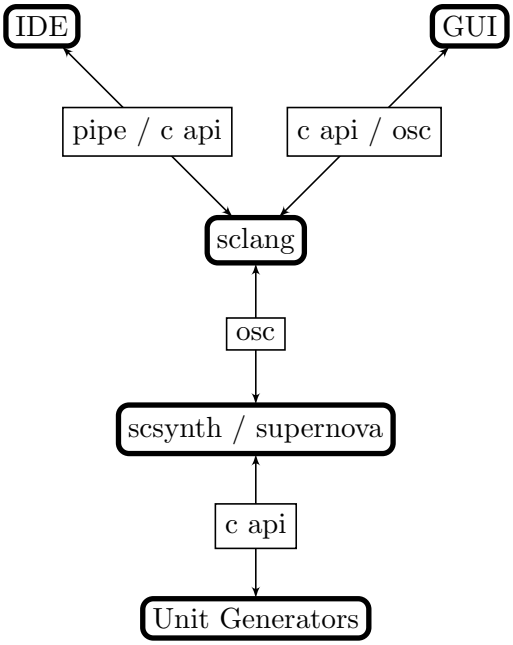


Figure 1: SuperCollider architecture

Synthesis Server (scsynth) The SuperCollider server is the signal processing engine. It is controlled from the Language using a simple OSC-based network interface.

Unit Generators Unit Generators (building blocks for the signal processing like oscillators, filters etc.) are provided as plugins. These plugins are shared libraries with a C-based API, that are loaded into the server at boot time.

IDE There are a number of integrated developer environments for SuperCollider source files. The original IDE is an OSX-only application, solely built as IDE for SuperCollider. Since it is OSX only, several alternatives exist for other operating systems, including editor modes for Emacs, Vim, gedit. Beside that, there is a plugin for Eclipse and a windows client called Psycollider.

GUI To build a graphical user interface for SuperCollider, two solutions are widely used. On OSX, the application provides Cocoa-based widgets. For other operating sys-

[1] e.g. Jack, CoreAudio, ASIO, Portaudio use a pull model

tems, one can use SwingOSC, which is based on Java's Swing widget toolkit.

This modularity makes it easy to change some parts of the system, while keeping the functionality of other parts. It is possible to use the server from other systems or languages, or control multiple server instances from one language process. Users can extend the language by adding new classes or class methods or write new unit generators (ugens).

3.2 SuperCollider Node Graph

The synthesis graph of the SuperCollider server is directly exposed to the language as a node graph. A **node** can be either a **synth**, i.e. a synthesis entity, performing the signal processing, or a **group**, representing a list of nodes. So, the node graph is internally represented as a tree, with groups at its root and inner leaves. Inside a group, nodes are executed sequentially, the node ordering is exposed to the user. Nodes can be instantiated at a certain position inside a group or moved to a different location.

4 Multi-core support with supernova

supernova is designed as replacement for the SuperCollider server scsynth. It implements the OSC interface of scsynth and can dynamically load unit generators. Unit Generators for scsynth and supernova are not completely source compatible, though the API changes are very limited, so porting SuperCollider ugens to supernova is trivial. The API difference is described in detail in Section 5.3.

4.1 Parallel Groups

To make use of supernova's parallel execution engine, the concept of **parallel groups** has been introduced. The idea is, that all nodes inside such a parallel group can be executed concurrently. Parallel groups can be nested, elements can be other nodes (synths, groups and parallel groups).

The concept of parallel groups has been implemented in the SuperCollider class library with a new `PGroup` class. Its interface is close to the interface of the `Group` class, but uses parallel groups on the server. Parallel groups can be safely emulated with traditional groups, making it easy to run code using the `PGroup` extension on the SuperCollider server.

4.2 Shared Resources

When writing SuperCollider code with the `PGroup` extension, the access to shared resources (i.e. buses and buffers) needs some attention to avoid data races. While with nodes in sequential groups the access to shared resources is ordered, this ordering constraint does not exist for parallel groups. E.g. if a sequential group G contains the synths S1 and S2, which both write to the bus B, S1 writes to B before S2, if and only if S1 is located before S2. If they are placed in a parallel group P, it is undetermined, whether S1 writes to B before or after S2. This may be an issue, if the order of execution affects the result signal. Summing buses would be immune to this issue, though.

Shared resources are synchronized at ugen level in order to assure data consistency. It is possible to read the same resource from different unit generators concurrently, but only one unit generator can write to a resource at the same time. If the order of resource access matters, the user is responsible to take care of the correct execution order.

5 supernova Architecture

The architecture of supernova is not too different from scsynth, except for the node graph interpreter. It is reimplemented from scratch in the c++ programming language, making heavy use of the stl, the boost libraries and template programming techniques. This Section covers the design aspects, that differ between the SuperCollider server and supernova.

5.1 DSP Threads

The multi-threaded dsp engine of supernova consists of the main audio callback thread and a number of worker threads, running with a realtime scheduling policy similar to the main audio thread. To synchronize these worker threads, the synchronization constraints, that are usually imposed (see Section 2.2), have to be relaxed. Instead of not using any blocking synchronization primitives at all, the dsp threads are allowed to use spin locks, that are blocking, but don't suspend the calling thread for synchronizing with other realtime threads. Since this reduces the parts of the program, that can run in parallel, this should be avoided whenever it is possible to use lock-free data structures for synchronization.

5.2 Node Scheduling

At startup supernova creates a configurable number of worker threads. When the main audio callback is triggered, it wakes the worker threads. All audio threads processes jobs from

a queue of runnable items. With each queue item, an activation count is associated, denoting the number of its predecessors. After an item has been executed, the activation counts of its successors are decremented and if it drops to zero, they are placed in the ready queue. The concept of using an activation count is similar to the implementation of Jack2 [Letz et al., 2005].

Items do not directly map to dsp graph nodes, since sequential graph nodes are combined to reduce the overhead for scheduling item. E.g. groups that only contain synths would be scheduled as one item.

5.3 Unit Generator Interface

The plugin interface of SuperCollider is based on a simple c-style API. The API doesn't provide any support for resource synchronization, which has to be added to ensure the consistency of buffers and buses. supernova is therefore shipped with a patched version of the SuperCollider source tree, providing an API extension to cope with this limitation. With this extension, reader-writer locks for buses and buffers are introduced, that need to be acquired exclusively for write access. The impact for the ugen code is limited, as the extension is implemented with C preprocessor macros, that can easily be disabled. So the adapted unit generator code for supernova can easily be compiled for scsynth.

In order to port a SuperCollider unit generator to supernova, the following rules have to be followed:

- ugens, that do not access any shared resources, are binary compatible.
- ugens, that are accessing buses have to guard write access with a pair of `ACQUIRE_BUS_AUDIO` and `RELEASE_BUS_AUDIO` statements and read access with `ACQUIRE_BUS_AUDIO_SHARED` and `RELEASE_BUS_AUDIO_SHARED`. Ugens like `In`, `Out` and their variations (e.g. `ReplaceOut`) fall in this category.
- ugens, that are accessing buffers (e.g. sampling, delay ugens) have to use pairs of `ACQUIRE_SNDBUF` or `ACQUIRE_SNDBUF_SHARED` and `RELEASE_SNDBUF` or `RELEASE_SNDBUF_SHARED`. In addition to that, the `LOCK_SNDBUF` and `LOCK_SNDBUF_SHARED` can be used to create a RAII-style lock, that automatically unlocks the buffer, when running out of scope.
- To prevent deadlocks, one should not lock more than one resource type at the same time. When locking multiple resources of the same type, only resources with adjacent indices are allowed to be locked. The locking order has to be from low indices to high indices.

6 Limitations

At the time of writing, the implementation of supernova still imposed a few limitations.

6.1 Missing Features

The SuperCollider server provides some features, that haven't been implemented in supernova, yet. The most important feature, that is still missing is non-realtime synthesis, that is implemented by scsynth. In contrast to the realtime mode, OSC commands are not read from a network socket, but from a binary file, that is usually generated from the language. Audio IO is done via soundfiles instead of physical audio devices. Other missing features are ugen commands for specific commands to specific unit generators and plugin commands to add user-defined OSC handlers. While these features exist, they are neither used in the standard distribution nor in the sc3-plugins repository[2].

6.2 Synchronization and Timing

As explained in Section 5.2 the audio callback wakes up the worker threads before working on the job queue. The worker threads are not put to sleep until all jobs for the current signal block have been run, in order to avoid some scheduling overhead. While this is reasonable, if there are enough parallel jobs for the worker threads, it leads to busy-waiting for sequential parts of the signal graph. If no parallel groups are used, the worker threads wouldn't do any useful work, or even worse, they would prevent the operating system to run threads with a lower priority on this specific CPU. This is neither very friendly to other tasks nor to the environment (power consumption). The user can make sure to parallelize the signal processing as much as possible and adapt the number of worker threads to

[2]The sc3-plugin repository (http://sc3-plugins.sourceforge.net/) is an svn repository of third-party unit generators, that is maintained independently from SuperCollider

match the demands of the application and the number of available CPU cores.

On systems with a very low worst-case scheduling latency, it would be possible to put the worker threads to sleep, instead of busy-waiting for the remaining jobs to finish. On a highly tweaked linux system with the RT Preemption patches enabled, one should be able to get worst-case scheduling latencies below 20 microseconds [Rostedt and Hart, 2007], which would be acceptable, especially if a block size of more than 128 samples can be used. On my personal reference machines, worst-case scheduling latencies of 100 µs (Intel Core2 Laptop) and 12 µs (Intel i7 workstation, power management, frequency scaling and SMT disabled) can be achieved.

7 Conclusions

supernova is a scalable solution for real-time computer music. It is tightly integrated with the SuperCollider computer music system, since it just replaces one of its components and adds the simple but powerful concept of parallel groups, exposing the parallelism explicitly to the user. While existing code doesn't make use of multiprocessor machines, it can easily be adapted. The advantage over other multiprocessor-aware computer music systems is its scalability. An application doesn't need to be optimized for a certain number of CPU cores, but can use as many cores as desired. It does not rely on pipelining techniques, so no latency is added to the signal. Resource access is not ordered implicitly, but has to be dealt with explicitly by the user.

8 Acknowledgements

I would like to thank James McCartney for developing SuperCollider and publishing it as open source software, Anton Ertl for the valuable feedback about this paper, and all my friends, who supported me during the development of supernova.

References

Stéphane Letz, Yann Orlarey, and Dominique Fober. 2005. Jack audio server for multiprocessor machines. In *Proceedings of the International Computer Music Conference*.

Thor Magnusson. 2006. Affordances and constraints in screen-based musical instruments. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 441–444, New York, NY, USA. ACM.

James McCartney. 1996. SuperCollider, a new real time synthesis language. In *Proceedings of the International Computer Music Conference*.

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

Miller Puckette. 1991. FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67.

Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

Miller Puckette. 2008. Thoughts on Parallel Computing for Music. In *Proceedings of the International Computer Music Conference*.

Ulrich Reiter and Andreas Partzsch. 2007. Multi Core / Multi Thread Processing in Object Based Real Time Audio Rendering: Approaches and Solutions for an Optimization Problem. In *Audio Engineering Society 122th Convention*.

Steven Rostedt and Darren V. Hart. 2007. Internals of the RT Patch. In *Proceedings of the Linux Symposium*, pages 161–172.

Work Stealing Scheduler for Automatic Parallelization in Faust

Stephane Letz and **Yann Orlarey** and **Dominique Fober**
GRAME
9 rue du Garet, BP 1185
69202 Lyon Cedex 01,
France,
{letz, orlarey, fober}@grame.fr

Abstract

FAUST 0.9.10 [1] introduces an alternative to OpenMP based parallel code generation using a Work Stealing Scheduler and explicit management of worker threads. This paper explains the new option and presents some benchmarks.

Keywords

FAUST, Work Stealing Scheduler, Parallelism

1 Introduction

Multi/many cores machines are becoming common. There is a challenge for the software community to develop adequate tools to take profit of the new hardware platforms [3] [8]. Various libraries like Intel Thread Building Blocks [2] or "extended C like" Cilk [5] can possibly help but still require the programmer to precisely define sequential and parallel sub part of the computation and use the appropriate library call or building blocks to implement the solution.

In the audio domain, work has been done to parallelize well known algorithms [4], but very few systems aim to help in *automatic* parallelization. The problem is usually tricky with stateful systems and imperative approaches where data has to be shared between several threads, and concurrent access have to be accurately controlled.

On the contrary, the functional approach used in high level specification languages generally helps in this area. It basically allows the programmer to define the problem in an abstract way completely unconnected of implementations details, and let the compiler and various backends do the hard job.

By exactly controlling when and how state is managed during the computation, the compiler can decide what multi-threading or parallel generation techniques can be used. More sophisticated methods to reorganize the code to fit specific architectures can then be tried and analyzed.

1.1 The Faust approach

FAUST [2] is a programming language for real-time signal processing and synthesis designed from scratch to be a compiled language. Being efficiently compiled allows FAUST to provide a viable high-level alternative to C/C++ to develop high-performance signal processing applications, libraries or audio plug-ins.

The FAUST compiler is built as a stack of code generators. The scalar code generator produces a single computation loop. The vector generator rearrange the C++ code in a way that facilitates the autovectorization job of the C++ compiler. Instead of generating a single sample computation loop, it splits the computation into several simpler loops that communicates by vectors. The result is a Direct Acyclic Graph (DAG) in which each node is a computation loop.

Starting from the DAG of computation loops (called "tasks"), FAUST was already able to generate parallel code using OpenMP directives [1]. This model has been completed with a new dynamic scheduler based on a *Work Stealing* model.

2 Scheduling strategies for parallel code

The role of scheduling is to decide how to organize the computation, in particular how to assign tasks to processors.

2.1 Static versus dynamic scheduling

The scheduling problem exists in two forms: static and dynamic. In static scheduling usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are known before

[1] this work was partially supported by the ANR project ASTREE (ANR-08-CORD-003)

[2] http://www.threadingbuildingblocks.org/

program execution. Temporal and spatial assignment of tasks to processors are done by the scheduler to optimize the execution time. In dynamic scheduling on the contrary, only a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be realized on-the-fly, and assignment of tasks to processors are made at run time.

2.2 DSP specific context

We can list several specific elements of the problem:

- the graph describes a DSP processor which is going to read n input buffers containing p frames to produce m output buffers containing the same p number of frames

- the graph is completely known in advance, but precise cost of each task and communication times (memory access, cache effects...) is not known

- the graph can be computed in a single step: each task is going to process p frames in a single step, or buffers can be cut in slices and the graph can be executed several times on sub slices to fill output buffers. (see "pipelining" section)

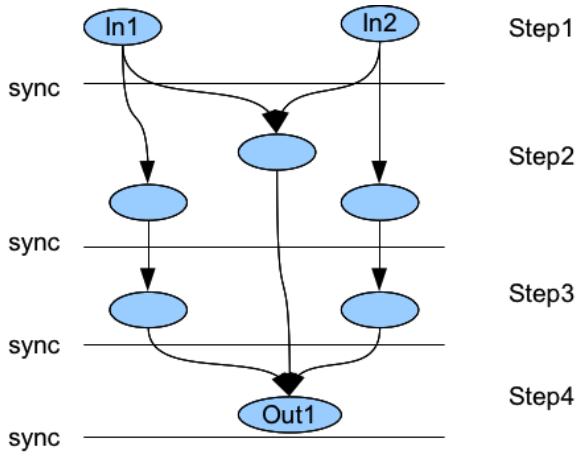


Figure 1: *Tasks graph with forward activations and explicit synchronization points*

2.3 OpenMP mode

The OpenMP based parallel code generation is activated by passing the `---openMP` (or `--omp`) option to the FAUST compiler. It implies the `--vec` option as the parallel code generation is built on top of the vector code generation.

In OpenMP mode, a topological sort of the graph is done. Starting from the inputs, tasks are organized as a sequence of groups of parallel tasks. Then appropriate OpenMP directives are added to describe a *fork/join* model. Each group of task is executed in parallel and synchronization points are placed between the groups (Figure 1). This model gives good results with the Intel icc compiler. Unfortunately until recently, OpenMP implementation in g++ was quite weak and inefficient, and even unusable in an real-time context on OSX.

Moreover the overall approach of organizing tasks as *a sequence of groups of parallel tasks* is not optimum in the sense that synchronization points are required for all threads when part of the graph could continue to run. In some sense the synchronization strategy is too strict and a data-flow model is more appropriate.

2.3.1 Activating Work Stealing Scheduler mode

The scheduler based parallel code generation is activated by passing the `--scheduler` (or `--sch`) option to the FAUST compiler. It implies the `--vec` option as the parallel code generation is built on top of the vector code generation.

With the `--scheduler` option, the FAUST compiler uses a very different approach. A data-flow model for graph execution is used, to be executed by a dynamic scheduler. Parallel C++ code embedding a Work Stealing Scheduler and using a pool of worker threads is generated.

Threads are created when the application starts and all participate in the computation of the graph of tasks. Since the graph topology is known at compilation time, the generated C++ code can precisely describe the execution flow (which tasks have to be executed when a given task is finished...). Ready tasks are activated at the beginning of the compute method and are executed by available threads in the pool. The control flow then circulate in the graph from inputs task to output tasks in the form of *activations* (ready task index called *tasknum*) until all output tasks have been executed.

2.4 Work Stealing Scheduler

In a Work Stealing Scheduler [7], idle threads take the initiative: they attempt to *steal* tasks from other threads. This is possible by having each thread owns a *Work Stealing Queue*, a special double-ended queue with a Push operation, a *private* LIFO Pop operation [3] and a *public*

[3] which does not need to be multi-thread aware

FIFO Pop operation [4]. The basic idea of work stealing is for each processor to place work when it is discovered in its local WSQ, greedily perform that work from its local WSQ, and steal work from the WSQ of other threads when the local WSQ is empty.

Starting from a ready task, each thread executes it and follows the data dependencies, possibly pushing ready output tasks into its own local WSQ. When no more tasks can be executed on a given computation path, the thread pops a task from its local WSQ using its private LIFO Pop operation. If the WSQ is empty, the thread is allowed to steal tasks from other threads WSQ using their public FIFO Pop operation.

The local LIFO Pop operation allows better cache locality and the FIFO steal Pop larger chuck of work to be done. The reason for this is that many work stealing workloads are divide-and-conquer in nature, stealing one of the oldest task implicitly also steals a (potentially) large subtree of computations that will unfold once that piece of work is stolen and run.

For a given cycle, the whole number of frames is used in one graph execution cycle. So when finished, a given task will (possibly) activate its output task only, and activation goes forward.

2.4.1 Code generation

The compiler produces a `computeThread` method called by all threads:

- tasks are numbered and compiled as a big *switch/case* block

- a *work stealing* task which aim to find out the next ready task is created

- an additional *last task* is created

2.5 Compilation of different type of nodes

For a given task in the graph, the compiled code will depend of the topology of the graph at this stage.

2.5.1 One output and direct link

If the task has one output only and this output has one input only (so basically there is a single link between the two tasks), then a *direct activation* is compiled, that is the tasknum of the next task is the tasknum of the output task, and there its no additional step required to find out the next task to run.

[4] which has to be multi-thread aware using lock-free techniques and is thus more costly

2.5.2 Several outputs

If the task has several outputs, the code has to:

- init tasknum with the `WORK_STEALING` value

- if there is a direct link between the given task and one of the output task, then this output task will be the next to execute. All other tasks with a direct link are pushed on current thread WSQ [5]

- otherwise for output tasks with more than one input, the activation counter is atomically decremented (possibly returning the tasknum of the next task to execute)

- after execution of the activation code, tasknum will either contains the actual value of the next task to run or `WORK_STEALING`, so that the next ready task if found by running the work stealing task.

2.5.3 Work Stealing task

The special *work stealing task* is executed when the current thread has no more next task to run in its computation path and its WSQ is empty. The `GetNextTask` function aims to find out a ready task by possibly stealing a task to run from any of the other threads except the current one. If no task is ready then `GetNextTask` returns `WORK_STEALING` value and the thread loops until it finally finds a task or the whole computation ends.

2.5.4 Last task

Output tasks of the DAG are connected and activate the special *last task* which in turn quits the thread.

```
void computeThread(int thread)
 {
     TaskQueue taskqueue;
     int tasknum = -1;
     int count = fFullCount;

     // Init input and output
     FAUSTFLOAT* input0
        = &input[0][fIndex];
     FAUSTFLOAT* input1
        = &input[1][fIndex];
     FAUSTFLOAT* output0
        = &output[0][fIndex];

     // Init graph
     int task_list_size = 2;
     int task_list[2] = {2,3};
     taskqueue.InitTaskList(
        task_list_size,
        task_list,
        fDynamicNumThreads,
```

[5] The chosen task here is the first in the task output list, more sophisticated choice heuristics could be tested at this stage.

```
        cur_thread,
        tasknum);

    // Graph execution code
    while (!fIsFinished) {
      switch (tasknum) {
       case WORK_STEALING: {
         tasknum
           = GetNextTask(thread);
         break;
       }
       case LAST_TASK: {
         fIsFinished = true;
         break;
       }
       case 2: {
         // DSP code
         PushHead(4);
         tasknum = 3;
         break;
       }
       case 3: {
         // DSP code
         PushHead(5);
         tasknum = 4;
         break;
       }
       case 4: {
         // DSP code
         tasknum
           = ActivateOutput(LAST_TASK)
         break;
       }
       case 5: {
         // DSP code
         tasknum
           = ActivateOutput(LAST_TASK)
         break;
       }
      }
    }
  }
```

Listing 1: example of `computeThread` method

2.6 Activation at each cycle

The following steps are done at each cycle:

- n-1 threads are resumed and start the work
- the calling thread also participates
- ready tasks (tasks that depends of the input audio buffers or tasks with no inputs) are dispatched between all threads, that is each thread pushes sub-tasks in its private WSQ and takes one of them to be directly executed
- after having done its part of the work, the main thread waits for all worker threads to finish

This is done in the following `compute` method called by the master thread:

```
void compute(int fullcount,
             FAUSTFLOAT** input,
             FAUSTFLOAT** output) {
  this->input = input;
  this->output = output;
  for (fIndex = 0;
    fIndex < fullcount;
    fIndex += 1024) {
    fFullCount
        = min(1024, fullcount-fIndex);
    TaskQueue::Init();
    // Initialize end task
    fGraph.InitTask(1,1);
    // Only initialize tasks with inputs
    fGraph.InitTask(3,1);
    // Activate worker threads
    fIsFinished = false;
    fThreadPool.
        SignalAll(fDynamicNumThreads-1);
    // Master thread participates
    computeThread(0);
    // Wait for end
    while (!fThreadPool.IsFinished()) {}
  }
}
```

Listing 2: master thread compute method

2.7 Start time

At start time, n worker threads are created and put in sleep state, thread scheduling properties and priorities are set according to calling thread parameters.

3 Pipelining

Some graphs are sequential by nature. Pipelining techniques aim to create and exploit parallelism in those kind of graph to better use multicores machines. The general idea is that for a sequence of two tasks A and B, instead of running A on the entire buffer size *then* run B, we want to split the computation in n sub-buffers and run A on the first sub-buffer, then B can be run on A first sub-buffer output while A runs on second sub-buffer and so on.

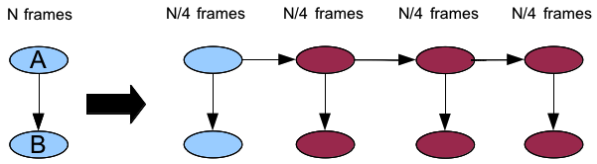


Figure 2: *Very simple graph rewritten to express pipelining, A is a recursive task, B is a non recursive one*

Different code generation techniques can be used to obtain the desired result. We choose to rewrite the initial graph of tasks, by *duplicating n times* each task to be run on a sub part of the buffer. This way the previously described code

generator can be directly used without major changes.

3.1 Code generation

The pipelining parallel code generation is activated by passing `-sch` option as well as the `--pipelining` option (or `-pipe`) with a value, the factor the initial buffer will be divided in.

Previously described code generation strategy has to be completed. The initial tasks graph is rewritten and each task is split in several sub-tasks:

- *recursive tasks* [6] are rewritten as a list of n connected sub-tasks (that is activation has to go from the first sub-task to the second one and so on). Each sub-task is then going to run on buffer-size/n number of frames. There is no real gain for the recursive task itself since it will still be computed in a sequential manner. But output sub-tasks can be activated more rapidly and possibly executed immediately is they are part of a non recursive task (Figure 2).

- *non recursive tasks* are rewritten as a list of n non connected sub-tasks, so that all sub-tasks can possibly be activated in parallel and thus run on several cores at the same time. Each sub-task is then going to run on buffer-size/n number of frames.

This strategy is used for all tasks in the DAG, the rewritten graph enters the previously described code generator and the complete code is generated.

4 Benchmarks

To compare the performances of these various compilation schemes in a realistic situation, we have modified an existing architecture file (Alsa-GTK on Linux and CoreAudio-GTK on OSX) to continuously measure the duration of the `compute` method (600 measures per run). We give here the results for three real-life applications: *Karplus32*, a 32 strings simulator based on the Karplus-Strong algorithm (figure 3), *Sonik Cube* (figure 4), the audio part software of an audio-visual installation and *Mixer* (figure 5), a multi-voices mixer with pan and gain.

Instead of time, the results of the tests are expressed in MB/s of processed samples because memory bandwidth is a strong limiting factor for today's processors (an audio application can never go faster than the memory bandwidth).

[6] those where the computation of the frame n depends of the computation of previous frames

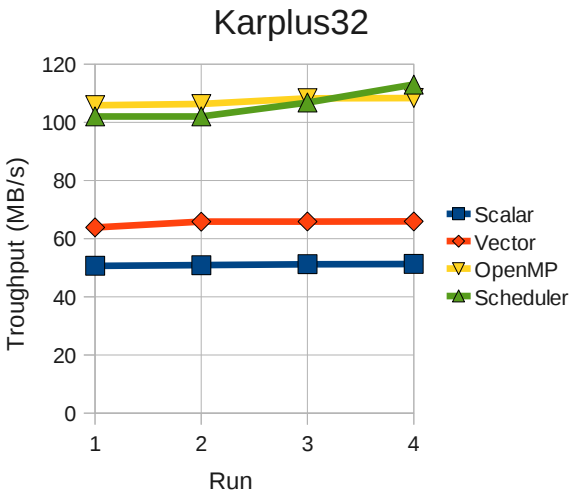


Figure 3: *Compared performances of the 4 compilation schemes on karplus32.dsp*

As we can see, in both cases the parallelization introduces a real gain of performances. The speedup for *Karplus32* was x2.1 for OpenMP and x2.08 for the WS scheduler. For *Sonik Cube* the speedup with 8 threads was of x4.17 for OpenMP and x5.29 for the WS scheduler. It is obviously not always the case. Simple applications, with limited demands in terms of computing power, tend to perform usually better in scalar mode. More demanding applications usually benefit from parallelization.

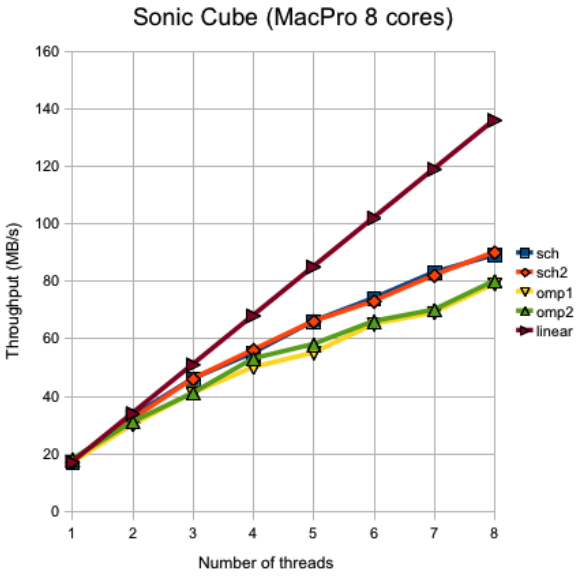


Figure 4: *Compared performances of different generation mode from 1 to 8 threads*

The efficiency of OpenMP and WS scheduler are quite comparable, with an advantage to WS scheduler with complex applications and more CPUs. Please note that not all implementations of OpenMP are equivalent. Unfortunately the GCC 4.4.1 version is still unusable for real time audio application. In this case the WS scheduler is the only choice. The efficiency is also dependent of the vector size used. Vector sizes of 512 or 1024 samples usually give the best results.

The pipelining mode does not show a clear speedup in most cases, but *Mixer* is an example when this new mode helps.

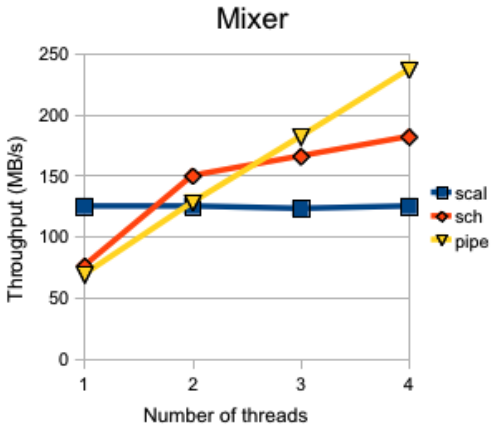


Figure 5: *Compared performances of different generation mode from 1 to 4 threads on OSX with a buffer of 4096 frames*

5 Open questions and conclusion

In general, the result can greatly depends on the DSP code going to be parallelized, the chosen compiler (icc or g++) and the number of threads to be activated when running. Simple DSP effects are still faster in scalar of vectorized mode and parallelization is of no use. But with some more complex code like *Sonik Cube* the speedup is quite impressive as more threads are added. But they are still a lot of opened questions that will need more investigation:

- improving dynamic scheduling, since right now there is no special strategy to choose the task to wake up in case a given task has several output to activate. Some ideas of static scheduling algorithms could be used in this context.

- the current compilation technique for pipelining actually duplicates each task code n times. This simple strategy will probably show its limit for more complex effects with a lot of tasks.

- there is no special strategy to deal with thread affinity, thus the performances can degrade if tasks are switched between cores.

- the effect of memory bandwidth limits and cache effects have to be better understood

- composing several pieces of parallel code without sacrificing performance can be quite tricky. Performance can severally degrade as soon as too much threads are competing for the limited number of physical cores. Using abstraction layers like Lithe [6]or OSX libdispatch [7] could be of interest.

References

[1] Y. Orlarey, D. Fober, and S. Letz. *Adding Automatic Parallelization to Faust.* Linux Audio Conference 2009.

[2] Yann Orlarey, Dominique Fober, and Stephane Letz. *Syntactical and semantical aspects of faust.* Soft Computing, 8(9):623632, 2004.

[3] J.Ffitch, R.Dobson, and R.Bradford. *The Imperative for High-Performance Audio Computing.* Linux Audio Conference 2009.

[4] Fons Adriaensen. *Design of a Convolution Engine optimised for Reverb.* Linux Audio Conference 2006.

[5] Robert D.Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: an efficient multithreaded runtime system.* In PPOPP 95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 207216, New York, NY, USA, 1995. ACM.

[6] Heidi Pan, Benjamin Hindman, Krste Asanovic. *Lithe: Enabling Efficient Composition of Parallel Libraries.* USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Berkeley, CA. March 2009.

[7] Blumofe, Robert D. and Leiserson, Charles E. *Scheduling multithreaded computations by work stealing.* Journal of ACM, volume 46, number 5, 1999, New York NY USA.

[8] David Wessel et al. *Reinventing Audio and Music Computation for Many-Core Processors.* International Computer Music Conference 2008.

[7] http://developer.apple.com/

A MusicXML Test Suite and a Discussion of Issues in MusicXML 2.0

Reinhold Kainhofer, http://reinhold.kainhofer.com, reinhold@kainhofer.com
Vienna University of Technology, Austria
and
GNU LilyPond, http://www.lilypond.org
and
Edition Kainhofer, http://www.edition-kainhofer.com, Austria

Abstract

MusicXML [Recordare LLC, 2010] has become one of the standard interchange formats for music data. While a "specification" in the form of some DTD files with comments for each element and equivalently in the form of XML Schemas is available, no representative archive of MusicXML unit test files has been available for testing purposes. Here, we present such an extensive suite of MusicXML unit tests [Kainhofer, 2009]. Although originally intended for regression-testing the musicxml2ly converter, it has turned into a general MusicXML test suite consisting of more than 120 MusicXML test files, each checking one particular aspect of the MusicXML specification.

During the creation of the test suite, several shortcomings in the MusicXML specification were detected and are discussed in the second part of this article. We also discuss the obstacles encountered when trying to convert MusicXML data files to the LilyPond [Nienhuys and et al., 2010] format.

Keywords

MusicXML, Test suite, Software development, Music notation

1 About the MusicXML test suite

MusicXML has become the de-facto exchange format for visual music data, supported by dozens of software applications. It has even been proposed as a tool for musicological analysis [Vigliante, 2007; Ganseman et al., 2008], online-music editing [Cunningham et al., 2006] or evaluating OMR systems [Szwoch, 2008] among others.

Recently, the first version of the Open Score Format specification [Yamaha Corporation, 2009] was published together with a PVG (Piano-Voice-Guitar) profile, which employs MusicXML as its data format. Thus most problems with MusicXML will automatically carry over to this new specification, too.

Despite a full syntactic definition of the MusicXML format [Recordare LLC, 2010], no official suite of representative MusicXML test files has been available for developers implementing MusicXML support in their applications. The only help were the comments and explanations given in the specification to create test cases manually. The predominant advice is to use the Dolet plugin for Finale, which is a proprietary Windows and MacOS application that is not easily available for many Open Source developers employing Linux. Also, this approach invariably will lead to MusicXML being interpreted as behaving like the Dolet plugin instead of being an application-independent specification. Furthermore, the lack of a test suite means that a lot of work is duplicated creating MusicXML test cases.

This lack of a complete MusicXML test suite for testing purposes was our main incentive for creating such a semi-official test suite [Kainhofer, 2009]. Due to the complexity of musical notation, a complete set of test cases, covering every possible combination of notation and all possible combinations of XML attributes and elements, is apparently out of reach. However, we attempted to create representative samples to catch as many common combinations as possible. Our main goal was to create small unit test cases, covering not only the most common features of MusicXML, but also some less used musical notation elements, like complex time signatures, instrument specific markup, microtones, etc.

The test suite together with sample renderings are available for download at its homepage: http://kainhofer.com/musicxml/

2 Structure of the test suite

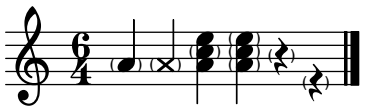
We identified twelve different feature categories, each dealing with separate aspects of the MusicXML specification (like basic musical notation, staff attributes, note-related elements, page layout, etc.). Each of these categories was further split into more specific aspects, for which we created several test cases each.

01-09 ... Basics
01 Pitches
02 Rests
03 Rhythm
10-19 ... Staff attributes
11 Time signatures
12 Clefs
13 Key signatures
20-29 ... Note-related elements
21 Chorded notes
22 Note settings, heads, etc.
23 Triplets, Tuplets
24 Grace notes
30-39 ... Notations, articul., spanners
31 Dynamics and other single symbols
32 Notations and Articulations
33 Spanners
40-44 ... Parts
41 Multiple parts (staves)
42 Multiple voices per staff
43 One part on multiple staves
45-49 ... Measure issues and repeats
45 Repeats
46 Barlines, Measures
50-54 ... Page-related issues
51 Header information
52 Page layout
55-59 ... Exact positioning of items
60-69 ... Vocal music
61 Lyrics
70-75 ... Instrument-specific notation
71 Guitar notation
72 Transposing instruments
73 Percussion
74 Figured bass
75 Other instrumental notation
80-89 ... MIDI and sound generation
90-99 ... Other aspects
90 Compressed MusicXML files
99 Compatibility with broken MusicXML

Table 1: Structure of the files, categorized by file name

The test suite currently consists of more than 120 test cases, where each file represents one particular aspect of the MusicXML format and is named accordingly. The file name starts with two digits, encoding the area of the test (see Table 1 for the exact meaning of the first two digits), followed by a letter to enumerate the test cases within each category. Finally, a short verbal description[1] of the test case is given in the file name. The file extension follows the standard of `.xml` for normal MusicXML files and `.mxl` for (ZIP-) compressed MusicXML archives as defined in `container.dtd`.

Every test case is supposed to test one particular feature or feature combination of MusicXML. If a feature has multiple possible attribute values or different uses within a score, the corresponding test file contains several subtests, separated as much as possible by using different notes or even different measures or staves for each of the values or combinations. For example, parenthesized notes or rests use the `parentheses` attribute of a `notehead` XML element. However, for an application it might make a difference if that note is a note on its own, a note with a non-standard note head or part of a chord. Similarly, parenthesized rests can have a default position in the staff or an explicit position given in the MusicXML file (using the `pitch` child element of the `note` describing the rest). The test case for parenthesizing covers all these cases:



This choice of combining closely related combinations or aspects of the same feature into one test case provides a nice balance between clearly separating test cases for different features to avoid influences of bugs in one feature on another feature, while still keeping the number of test files relatively low, which is relevant if running a test suite cannot be automated.

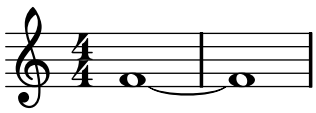
3 An example of a unit test

The unit test files are kept as simple as possible, so that they can best fulfill their purpose of checking only one particular aspect. For example, the unit test file `33b-Spanners-Tie.xml` to check the processing of simple ties – a trivial feature, which still needs to be tested in coverage and regression tests – reads:

```
<?xml version="1.0" encoding="ISO-8859-1"
    standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 0.6b
    Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <identification>
    <miscellaneous>
      <miscellaneous-field
          name="description">Two simple tied
           whole notes</miscellaneous-field>
    </miscellaneous>
  </identification>
```

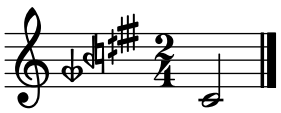
[1] A more detailed description is given in a description element inside the XML file.

```
  <part-list>
    <score-part id="P1"/>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key><fifths>0</fifths></key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <staves>1</staves>
        <clef number="1">
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>F</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <tie type="start"/>
        <voice>1</voice>
        <type>whole</type>
        <notations><tied
            type="start"/></notations>
      </note>
    </measure>
    <measure number="2">
      <note>
        <pitch>
          <step>F</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <tie type="stop"/>
        <voice>1</voice>
        <type>whole</type>
        <notations><tied
            type="stop"/></notations>
      </note>
    </measure>
  </part>
</score-partwise>
```



Other test files check for more exotic features, like for example the test case for non-traditional key signatures with microtone alterations (excerpt of `13d-KeySignatures-Microtones.xml`):

```
<attributes>
  <divisions>1</divisions>
  <key>
    <key-step>4</key-step>
    <key-alter>-1.5</key-alter>
    <key-step>6</key-step>
    <key-alter>-0.5</key-alter>
    <key-step>0</key-step>
    <key-alter>0</key-alter>
    <key-step>1</key-step>
    <key-alter>0.5</key-alter>
    <key-step>3</key-step>
    <key-alter>1.5</key-alter>
  </key>
[...]
</attributes>
```



However, in all cases the structure of the test file is kept as simple as possible to avoid cross-interactions of bugs in different areas of an application.

4 Sample renderings

Originally, the files of the test suite were generated as test cases for the implementation of musicxml2ly, a utility to convert MusicXML files to the LilyPond [Nienhuys and et al., 2010] format. Using this utility, sample renderings of all the test cases can be created automatically and are made available on the homepage of the test suite.

However, these renderings cannot be regarded as official reference renderings, either, since they represent only one particular interpretation (the one of the musicxml2ly converter), while the MusicXML specification leaves several aspects unclear, as detailed below. So several aspects of the semantic interpretation of MusicXML are left to the importing application. Furthermore, the musicxml2ly converter does not yet fully support every aspect of MusicXML. Rather, these sample renderings should be understood as an indication how one particular application understands the files.

5 Shortcomings of the MusicXML format

While generating the MusicXML test suite, we encountered several problems or inconsistencies with the MusicXML specification as given in the DTDs or XML Schemas. Those issues involve semantic ambiguities, suboptimal XML design choices and missing features.

5.1 Semantic ambiguities of MusicXML

First, while the MusicXML specification gives a precise syntactic specification of the format, the complexities of music notation inevitably lead to additional semantic restrictions, that can not properly be expressed in a DTD or an XML Schema. Additionally, MusicXML is also designed to cover the features of several commercially available music typesetting applications, where each application has implemented some

aspects differently. Unless the MusicXML specification clearly describes how certain combinations of attributes and/or elements are to be understood, there cannot be a unique interpretation of the exact musical or layout content of some MusicXML snippet.

Thus, the first type of issues we identified are semantic ambiguities, which might or rather should be clarified in the specification itself. Several of these were answered or explained by Michael Good in email threads on the MusicXML mailing list, but the official specification remains ambiguous to new developers.

5.1.1 Only a syntactical definition

The biggest problem with MusicXML is that it is a *purely syntactical definition*, while the musical content requires additional semantic restrictions. For example, spanners like tuplets, ties, crescendi, analysis brackets etc. are simply marked with a start, possibly some continuation and an end element. Several spanners can be arbitrarily overlapping, however, it is not possible to properly specify that each of these spanners must be closed (at a position where it makes sense musically). Furthermore, it does not make sense from a musical standpoint to have e.g. a crescendo and a decrescendo overlapping in the same voice. However, a part can have several different voices, each with different hairpins, so this is not a restriction for a part, only for a voice in the conventional sense. Adding such restrictions at the voice-level is simply not possible in a pure syntax specification like the DTD or XSD.

5.1.2 Voice-Basedness

Many music notation and sequencer applications are based on the concept of *voices*, which was also introduced in MusicXML through the `voice` element of `note`. Notes with the same voice element value are assumed to be in the same voice, but the voice element is not required (in the OSF PVG profile, a voice element is finally required). It is not clear, whether a missing voice element implicitly means voice one or whether notes without a voice element should be placed in a voice of their own. In any case, an importing application needs to check whether the imported assignment of notes to voices is possible in the application at all, and thus the voice element can only be used as a strong indication, but not as a definitive assignment to voices. In particular, many applications don't allow overlapping notes in the same voice. Such notes would need to be assigned to different voices upon import.

5.1.3 Attributes

Another unclear part of the MusicXML specification regards the *display of the settings* of the `attributes` element. Some applications export the time and key signature for every measure, even if it hasn't changed. The specification is quiet on whether the presence of an `attributes` element is supposed to imply the presence of a graphical indication of these settings (i.e. explicitly displaying the key or time signature) or only to assert specific values and leave the decision to implementations. In the latter case (which is apparently favored by Michael Good), the MusicXML file does not specify the layout uniquely and different applications will produce different output.

5.1.4 Chords

Chords are another unclear area in the specification: A sequence of notes with the `chord` element can only appear after a note that does not have the `chord` element set and serves as the base note for the chord. This restriction cannot be handled in a DTD, but was introduced in the PVG profile of OSF. Even worse, the grammar allows e.g. `forward` or `backward` elements before a note with `chord`, so that the note does no longer appear at the same time as the chord's base note. This additional restriction that no forwards or backward appear between the notes of a chord needs to be clarified. Also, theoretically the notes of a chord can belong to different voices in the file, while this is not supported by most applications.

5.1.5 Lyrics

Lyrics in MusicXML can have both a stanza number and a name attribute to distinguish different lyrics lines. But the specification is quiet whether the number, the name alone or the combination of number and name should be used to determine, which syllables belong together. Even worse, it seems that for the page display, the vertical position of the lyrics is the main factor to associate lyrics syllables into words. This is a violation of the otherwise rather strict separation of content and display.

5.1.6 Figured Bass

Figured bass numbers in MusicXML are always assigned to the "first regular note that follows" per specification, but it does not say if this is meant in XML order or in time-order. In particular, there might be a `forward` or `backward`

element immediately after the figured bass. In this case, the XML-order and time-wise order is clearly different. Also, the `slash` value of the `suffix` child element does not distinguish forward slashes and back ticks (usually through a "6" to indicate a diminished chord). The DTD only says "The orientation and display of the slash usually depends on the figure number." While forward and backward slashes might mean the same from a musical point of view, the display of the figure will thus vary from application to application. In most other cases, in contrast, MusicXML tries to be as precise possible as far as the display is concerned.

5.1.7 Harp Pedal Diagrams

Finally, the `harp-pedals` element for *harp pedal diagrams* lists the pedal states for the D, C, B, E, F, G, and A harp pedals, but only recommends to give the pedals in the usual order. For different orders, it is not clear whether the harp pedal should be displayed in the usual order or in the order given. Also, there is no way to indicate the usual vertical delimiter for other orders.

5.2 Sub-optimal XML design

A further type of issues with the MusicXML format concerns the general design of the XML format. As the MusicXML format is supposed to be backward compatible, these design choices cannot be undone. However, for completeness, we will still discuss how they could have been done better.

5.2.1 Strict Order of XML Child Nodes

The MusicXML DTDs specify that the children of a `note` element (and of several similar elements) have to be in a *fixed order*. In particular, the `duration`, the optional `voice` and the `type` elements have to be in this exact order, although one would intuitively place the duration (the length in time units) and the type of the note (the visual representation of the duration) together. Also, from a theoretical point of view, there is no need to force a fixed ordering, as all child elements simply specify properties of that note. The only reason for this fixed ordering is – according to Michael Good[2] – a technical one, namely that the DTD does not provide a proper way to allow arbitrary ordering of the XML child elements while at the same time adding restrictions on the number of times an element appears. With an XML Schema this would be possible to model, but the XSD for MusicXML still enforces a fixed ordering to provide backwards compatibility.

5.2.2 Names of XML Elements Containing Pitch Information

Another suboptimal design choice regards all elements that contain some kind of pitch information or alteration. Their names are chosen to include the name of the enclosing XML element, even though this can already be deduced from the enclosing context. For example, the alteration of a normal note uses the `alter` element inside a `pitch`, while the alteration of a chord root uses the `root-alter` element inside a `root` and the alteration of a chord note uses `degree-alter` inside `degree`. As all of them simply indicate an alteration of the enclosing element, it would be cleaner to use the same element name `alter` for all these uses and instead use the enclosing context, too. This would also make implementations cleaner, as they can base all pitch and alteration information on one common base class, using the same names for the children. This issue appears not only with the `alter` and `(key|tuning|root|degree)-alter` elements, but also with the `step` and `(key|tuning|root)-step` as well as with the `octave` and `(key|tuning)-octave` elements.

5.2.3 Metronome Marks and Non-Standard Key Signatures

Contrast this over-correctness in naming and ignoring the context in the XML tree to the `metronome` element. Tempo changes of the form "old value = new value" using the `metronome` element are defined in the DTD as

```
(beat-unit, beat-unit-dot*,
 beat-unit, beat-unit-dot*)
```

where the two beat units indicate the value before and after the tempo change. As a result of the optional dots, one cannot simply access the old and the new tempo unit separately by their child index of the `metronome` element, but has to iterate through the children sequentially, checking for existing dots. For most other purposes MusicXML tries to give each item with even the slightest different function its own name. Similarly, custom key signatures are defined as

```
((...|((key-step, key-alter)*)),
                          key-octave*)
```

[2] Mail message to the MusicXML mailing list on March 11, 2008

where steps and alterations alternate. This design has the additional problem that the (optional) octave definitions for each of the key elements are given after the step/alter pairs in left-to-right order. As pointed out by Tulgan[3], this would be better represented by enclosing each of the key element information (step, alter and optional octave) in a separate `key-element` child element.

5.2.4 Shortcomings of DTD and XSD Formats

In the original DTD for MusicXML, *enumerated element values* cannot be properly modeled, but have to use the #PCDATA type and explain the possible values in the comments. Thus, all enumerations are badly defined in the DTD, as only some possible values are mentioned, which are unaccessible to any syntax checker. In the XSD specification for MusicXML, the possible values are mostly properly specified using an enumeration, which however makes extensions of MusicXML to non-western music notation very hard (for example, displaying microtone accidentals used in Turkish music). Additionally, the possible values are not further explained (for example `sharp-sharp` vs. `double-sharp` for accidentals).

A similar problem are *attributes*, where the DTD allows a text value, which is then understood as an integer or a real number. These attributes are mainly fixed to integer values in the OSF PVG profile.

5.2.5 Staff-assigned Elements

Markup text in MusicXML is mostly *assigned to a whole staff*. In reality, however, a text might be inherently assigned to one particular note or voice. For example, when two instruments are combined as two voices on one staff, a markup "dolce" or "pizz." might apply only to one of the two instruments. Similar cases are instrument cue names given in a piano reduction, as can be seen for example in the Telemann MusicXML example provided by Recordare. Unfortunately, the `voice` child element of `direction` is optional and thus most MusicXML exporters ignore it, causing problems for voice-based applications.

5.3 Missing features in MusicXML 2.0

While MusicXML 2.0 already added several important features over MusicXML 1.1, like exact positioning and offsets, there are several settings for professional music typesetting, which cannot be encoded in MusicXML. For these issues, we give some suggestions for a possible inclusion in MusicXML 2.1 or 3.0.

5.3.1 Document-wide Header and Footer Lines

First, while MusicXML allows one to completely describe the page layout of the music sheet, there is no way to specify a *document-wide header or footer line*. The `credit` element allows to place arbitrary text on any page, but it refers to only one page (page 1 by default). Thus, page headers and footers need to be defined for each page separately. We propose to add values "all", "even" and "odd" to the data type of the page attribute for `credit` (currently xsd:positiveInteger):

```
<credit page="even">
  <credit-words default-x="955"
      default-y="20">Even
      footer</credit-words>
</credit>
```

5.3.2 No Information about Purpose of Credit Elements

Another problem with the `credits` elements is that they are simply text labels, but do not store its function (i.e. whether such an element gives the composer, poet, title, etc.). For a pure layout-based application this might suffice, but any application, that tries to extract metadata from the MusicXML file or that has a different handling for score titles and contributor names, needs to extract that information. We propose to add an enumerated `type` attribute to the `credit` element with possible values title, subtitle, composer, poet or lyricist, arranger, publisher, page number, header, footer, instrument, copyright, etc.

```
<credit type="composer">
  <credit-words
    default-x="1124" default-y="1387"
    justify="right">Composer</credit-words>
</credit>
```

5.3.3 System Delimiters

In orchestral scores the systems are typically separated with a *system delimiter* consisting of two adjacent thick slashes. This is currently also not possible to express in MusicXML. One possible solution would be to add it as a `system-separator` element inside the `defaults -> system-layout` block of a score.

[3] Mail message to the MusicXML mailing list on October 7, 2006.

```
<defaults>
 <system-layout>
   <system-separator>double-slash
       </system-separator>
 </system-layout>
</defaults>
```

5.3.4 Cadenzas

Finally, there is no proper way to *encode a cadenza* in MusicXML. While a measure can have an arbitrary number of beats in MusicXML, irrespective of the time signature, the information about where a cadenza starts cannot be represented. This will cause problems with applications and utilities that check MusicXML files for correctness, as they have no way to distinguish incorrectly encoded files from files with a cadenza.

6 Issues in MusicXML translation to the LilyPond

In [Good, 2002] Michael Good discusses issues that appear in the translation of MusicXML files from and to the MuseData, NIFF, MIDI and Finale file formats. In this section, we will discuss issues that appear in the conversion of the MusicXML format to the LilyPond file format.

6.1 Musical Content vs. Graphical Representation

As LilyPond [Nienhuys and et al., 2010] is a WYSIWYM (what you see is what you mean) application, its data files describe the musical contents of a score, rather than its graphical description. Thus a converter from MusicXML to LilyPond needs to extract the exact musical contents from a file. In MusicXML, several elements – most notably dynamic signs like p, f, crescendi etc. – are mainly tied to a position on the staff and in some cases its onset and end can only be deduced by the horizontal offset of the dynamic sign in the MusicXML file. In LilyPond, however, almost all notation elements are attached to a note or rest, so these elements need to be quantized and correctly assigned to a note or rest in LilyPond. This is often not easily possible, due to explicit offsets on the page, effectively assigning the element to a different position than the position in the MusicXML file.

6.2 Staff-Assigned Items

Another problem in the conversion to LilyPond is caused by the same fact that in MusicXML many notation elements like dynamic markings or text markup is assigned to a staff, while in LilyPond they must be assigned to a particular note. If a staff contains two instruments (i.e. two voices, one for each instrument), one has to determine to which note to assign an encountered dynamic marking or text markup. In most case assigning it to the nearest note will produce the desired output, but there are many cases where it is not possible to determine whether a marking belongs to both instruments or just to one of them. As an example, a dynamic sign like "p" or "ff" might either apply to both instruments at the same time, or (e.g. if one of them has a short solo) only to one of them. Similarly, each instrument cue name provided as a help for the conductor in a piano reduction apply only to one particular voice in the (multi-voice) piano part, while other text markups will apply to all voices simultaneously.

If one is only interested in generating the exact layout as provided in the MusicXML file, a misassignment will still lead to correct visual output, although the extracted music information is not entirely correct. However, if one also wants to create separate instrumental parts for the two instruments in the first case mentioned above, then it is crucial to correctly assign each staff-assigned element to either one or both instrumental voices. On the other hand, assigning an element to both voices will then lead to duplicated items in the LilyPond output.

While it is true that MusicXML defined the `direction` element to optionally contain a `voice` element for that exact purpose, in reality most GUI applications for music notation will not cater for this functionality and thus produce MusicXML files without proper voice-assignment.

6.3 Voice-Based vs. Measure-Based

A further problem is that LilyPond is voice-based, where the measure length is determined by the current time signature, while MusicXML is measure-based, where a measure can contain an arbitrary number of beats, irrespective of the time signature. In LilyPond, voices are independently split into measures during processing rather than in the input. Only later are those measures synchronized. As a consequence, if one voice has more beats in a measure than another voice, LilyPond will not be able to properly synchronize them, so some skip elements need to be inserted to line the voices up. The

voice-basedness of LilyPond also causes problems with MusicXML's optional voice element, which allows a voice to have overlapping notes, which is not allowed in LilyPond, so that a MusicXML voice will possibly need to be split into multiple voices.

6.4 Page Layout

Concerning the *page layout*, LilyPond needs metadata about the score title and the contributors and will create its own labels on the title page and the header and footer bars. As discussed above, the `credit` element does not contain that information, so that the title page from the MusicXML file cannot be reproduced. Even worse, page headers and other markup is placed at absolute positions in the MusicXML score, which is not possible in LilyPond.

6.5 Workarounds in MusicXML Files

Finally, even some sample files provided by Recordare on the MusicXML homepage mix the graphical display with the musical information. For example, in the `Chant.xml` sample file a divisio minima (a short tick through the topmost staff line) is not encoded as a barline with the proper `tick` bar-style attribute, but as a `words` direction element with text "|", which is then shifted manually so that it appears at the correct position. Such hacks cannot work with any application that tries to extract the musical contents and not the exact page layout.

7 Conclusion

The MusicXML test suite presented in this article finally provides software developers in the area of music notation with an extensive set of representative test cases to check conformance to the MusicXML specification and to perform regression and coverage tests.

Even though MusicXML has established itself as an industry standard for exchanging music notation, it is still encumbered with several minor issues. Our discussion in the second part of the paper attempts to provide implementors of MusicXML import and export features with some hints about possible pitfalls and ambiguities in the format.

Nonetheless, MusicXML is a very useful file format for the extremely hard and complex task of music notation exchange. As the OSF specification has already shown, one can expect that future versions of MusicXML will clarify, solve or at least soften most of the issues we discuss here.

References

Stuart Cunningham, Nicole Gebert, Rich Picking, and Vic Grout. 2006. Web-based music notation editing. In *Proc. IADIS Int. Conf. on WWW/Internet 2006*. Murcia, Spain, October 5-8, 2006.

Joachim Ganseman, Paul Scheunders, and Wim D'haes. 2008. Using XQuery on MusicXML databases for musicological analysis. In *Proc. ISMIR 2008*, pages 427–432. 9th Int. Conf. on Music Information Retrieval. Philadelpha, September 14-18, 2008.

Michael Good. 2002. MusicXML in practice: Issues in translation and analysis. In *Proc. First Int. Conf. MAX 2002: Musical Application Using XML*, pages 47–54. Milan, September 19-20, 2002.

Michael Good. 2006. Lessons from the adoption of MusicXML as an interchange standard. In *Proc. XML 2006*.

Reinhold Kainhofer. 2009. Unofficial MusicXML test suite. `http://kainhofer.com/musicxml/`. Representative set of MusicXML test cases.

Han-Wen Nienhuys and Jan Nieuwenhuizen et al. 2010. GNU LilyPond. `http://www.lilypond.org/`. The music typesetter of the GNU project.

Recordare LLC. 2010. MusicXML 2.0. Document Type Definition (DTD): `http://musicxml.org/dtds`, W3C XML Schema Definition (XSD): `http://musicxml.org/xsd`.

Mariusz Szwoch. 2008. Using MusicXML to evaluate accuracy of OMR systems. In *Diagrammatic Representation and Inference: Proc. Diagrams 2008*, volume 5223 of *Lecture Notes in Computer Science*, pages 419–422. Springer Verlag, Berlin. Herrsching, Germany, September 19-21, 2008.

Raffaele Vigliante. 2007. MusicXML: An XML based approach to automatic musicological analysis. In *Proc. Digital Humanities 2007*. Urbana-Champaign, IL, June 4-8, 2007. Urbana-Champaign, IL, June 4-8, 2007.

Yamaha Corporation. 2009. Open Score Format (OSF, ver. 1.0), packaging specification. `http://openscoreformat.sf.net/`.

"3DEV: A tool for the control of multiple directional sound source trajectories in a 3D space"

Oscar Pablo Di Liscia
Carrera en Composición con Medios Electroacústicos
Universidad Nacional de Quilmes
Roque Saenz Peña 352
Bernal, Argentina, B1876BXD
odiliscia@unq.edu.ar

Esteban Calcagno
Carrera en Composición con Medios Electroacústicos
Universidad Nacional de Quilmes
Roque Saenz Peña 352
Bernal, Argentina, B1876BXD
ec@lapso.org – estebancalca@gmail.com

Abstract

This Paper presents a GNU software (3DEV) developed for the creation, transformation and temporal coordination of multiple directional sound source trajectories in a three- dimensional space. 3DEV was conceived as a general tool to be used in electroacoustic music composition, and the data generated by it may be transmitted on a simple and effective way to several spatialisation programs.

Keywords

Sound Spatialisation, Electroacoustic Music, GNU Software

1 General

1.1 The spatial sound in electroacoustic composition

From its very beginning, *avant-garde* music (both instrumental and electroacoustic) favoured a significant development of timbre, texture and sound spatial quality together with the organisation of pitch and duration which are, by the other hand, structural features of traditional classical music. In the electroacoustic music, the spatial quality of sound plays a very relevant role (if only because of the fact that this music must be reproduced by loudspeakers). Because of this, the analysis of the aesthetic influence of the spatial quality of sound and its relations with other sonic organisations become an essential part of electroacoustic music theory (See, as an example of this, Wishart [1], Cap. X).

A sound trajectory may determine clearly the end of a musical motto, or a formal articulation of a musical composition, just changing its direction suddenly or stopping abruptly its movement and developing a particular sonority from a steady position (See a classification of different kind of movements in Stockhausen, 1955 [2]). It arises then in the composer the necessity of gaining control of the spatial quality of the sound through technological tools that allow accurate temporal coordination between multiple audio signals and their trajectories or positions in a virtual space.

1.2 Sound spatialisation software development

Sound spatialisation is developed in several technologic and scientific areas mutually related. Among the most important can be mentioned: Acoustics (particularly room acoustics), Psychoacoustics (specifically Spatial Listening), Digital Signal Processing and Audio Engineering.

Particularly, and together with the development of computer music, many of the results of the scientific research on this field were applied in the creation of software applications with the purpose of providing concrete tools for musical and sonic production to the composers and audio engineers. The works of Chowning [3] and Moore ([4], [5] and [6]) must be regarded as a valuable and unavoidable initial reference which was followed by other significant developments. Among of them, the ones by Kendall [7], Furse [8][9], López-Lezcano [10], Karpen [11], Pulkki [12] and Varga [13] can be mentioned. At present, a growing amount of sound spatialisation programs exist, most of them as high level environments achieved by the combination of general purpose audio programs software units (such as Csound, Super Collider or Pure Data, to mention only some of them). Two of them are *AudioScape* (Wozniewski et al [14]), and *Bin-Ambi* (Musil, [15]).

From a general point of view, the spatial treatment of sound involves three basic features:

1-Environment: the physic space into which the sound sources are located.

2-Localization - movement: the points of the physic space where the sound sources are located, or the trajectories that they follow in this space when moving.

3-Directivity: the pattern of the acoustic energy spreading of the sound sources according their

nature and orientation with respect to a listening point (see, for example, Causse [16]).
All the already mentioned programs were designed for working on the features 1 and 2 (Environment and Localization - Movement). Only some of them, however, (see Moore [4], [5] and Furse [9]) can perform sound source directivity simulation.
By the other hand, from a design point of view, three basic sections can be distinguished among all the different spatialisation programs:
1-A general setting of configuration parameters usually related to the virtual room or environment, such as: size, shape, listener position, amount of echoes to be computed, dense reverberation quality, etc.
2-A sound trajectory and/or sound position generator for the virtual acoustic sources.
3-A D.S.P. block (Audio Engine) dedicated to the processing of the digital audio signals to be spatialised accordingly with the data of 1 and 2 and the techniques which are to be used (Ambisonics, Intensity Panning, VBAP, etc.)
The work presented here address the item 2 (sound trajectories generation) and discuss the design of a general purpose Graphic Interface for the shaping of multiple three-dimensional trajectories, independently of whatever spatialisation audio engine may be chosen to be used. This separation will allow the composer to further select the technique and software environment most appropriate to process the audio signals to be spatialised and, furthermore, even to use several different Audio Engines with the same data, if desired.

2 Basic features of 3DEV

3DEV is conceived to bring assistance to electroacoustic music composers in the creation, transformation and temporal coordination of multiple directional sound source trajectories in a three-dimensional space. It is being developed by Esteban Calcagno under the supervision of Oscar Pablo Di Liscia, as a part of the Research Project "Desarrollo de software para espacialización de sonido y música" (Teatro Acústico Research Program, Universidad Nacional de Quilmes, Argentina). The binaries for Linux as well as the source code and documentation can be obtained at: http://lapso.org/EstebanCalcagno/3dev.html.
3DEV allows to generate any amount of trajectories and to work on them using several editing tools. The trajectories are delimited by points (nodes) that can be visualised individually and globally. An especially valuable feature of 3DEV is its capacity of drawing a precise temporal relation between any of the generated trajectories and the audio signals with which these are associated. Additionally, it is also possible to create and modify a *rotation vector* which may be also accurately coordinated with the temporal evolution of the audio signal. If the Audio Engine to be further used allows the work with directional sound sources, it will then be possible transmit to it this data indicating how the virtual sound source eventually changes its orientation.
The control of the time between the different spatial nodes (indicated as a percent of the total duration of the signals with which the trajectories are associated) is at present the only way of handling movement velocity.

3 Description of the 3DEV program

3DEV uses the GTK+ C Library (http://www.gtk.org/) for its graphic interface design, as well as the Pthread (https://computing.llnl.gov/tutorials/pthreads/) and Sndlib (Schottstaedt,[18]) C Libraries for the Audio I/O handling. This program is a self-sufficient executable and, at present, cannot be a module of any other external program nor can support external modules.

3DEV is divided in three different areas: the Spatial Trajectories window, the Audio Files window and the Audio Engine connection window. The two first parts took the major part of the work, whilst the Audio Engine is still in development, using at present the Csound program. However, as already mentioned, the independence between movement design and audio processing will allow the use of several other programs for sound spatialisation.

3.1.1 Spatial trajectories window.

The graphic interface for the trajectories, whose basic functions for 3D graphics were taken and adapted from (Harlow [17]), was conceived as a main window with four menus which are briefly described below:
1-The "General View" window is divided in four equal size sections. One of them (the top leftmost one) shows (with 3D rotation and zoom options) a 3D plot of the trajectory that is actually been designed. The other three sections show the trajectory projection in the XY, XZ and YZ planes in two dimensions, and allow creating and/or editing new nodes.
2-The "Full 3D View" shows a 3D plot of all the trajectories created or a selected group of them.

3-The “Point List” menu shows a list of the Cartesian coordinates of a selected trajectory as well as allows editing them numerically.
4-The “Play List” shows a list of all the trajectories, and allows the association of any of them with an Audio File. Through this menu it is possible to access the Audio File Window, which will be described in the next section.

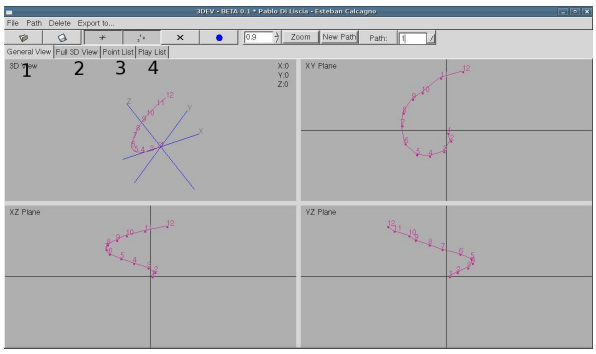


Fig. 1: 3DEV Main Window

3.1.2 Audio Files Window (Figure 2)

From top to bottom, this window is made with:
1-A plot of the waveform of the selected audio signal.
2-A horizontal line that relates the audio file duration with each one of the nodes (the 3D points between which the virtual source will travel). The horizontal position of each node can be modified to change the duration of each segment of the trajectory. This way, accurate time synchronization between movement and audio may be achieved.
3-A system of two editable envelopes indicating the orientation of the virtual sound source (azimuth and elevation angles). By this system, the orientation of the sound source can be changed dynamically also accurately synchronized with the audio signal.

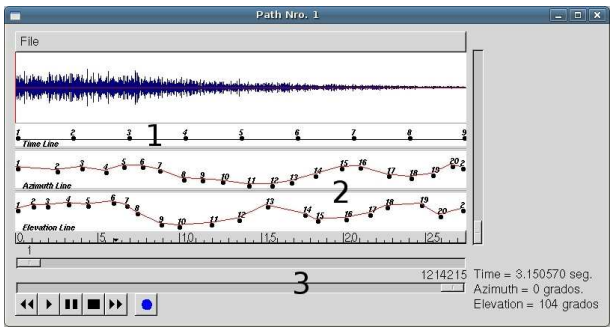


Fig. 2: Audio Files Window

4-Lastly, a section with a temporal reference of the selected audio signal (in seconds and samples) with zoom and reproduction controls.

3.1.3 Audio Engine Connection Window (Fig. 3)

At present, 3DEV uses the Csound program (Barry Vercoe et al) as Spatialisation Engine. In order to do this, 3DEV generates a text file with the format of a Csound score. This text file is a kind of script which may be further used by Csound to synthesize a spatialised signal based on a pre-designed orchestra which constitutes the basic DSP module.
In this case, the Csound´s orchestra uses basically the *spat3d* Unit Generator (ItzvanVarga [14]), that implements the Ambisonic spatialisation technique (see Malham[19]). The designed orchestra provides 3D spatialisation of both the direct signal and the early reflections of it into a virtual room whose features are provided by the user. The following figure shows the data entries needed for specifying the characteristics of such virtual room. The connection of the sound trajectories is very easily achieved by means of triplets of Tables (numerical vectors readable by Csound) containing each one the Cartesian coordinates (X, Y, Z). Each sound trajectory is represented in the Csound score as a “note” with its corresponding start time, duration, amplitude scaling and tables indicating its movement. This way, it is possible to spatialise and mix multiple virtual sound sources moving in different ways.

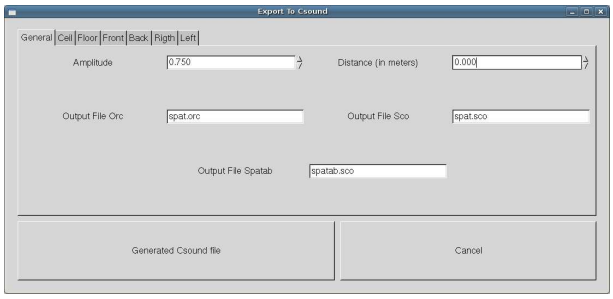


Fig. 3: Audio File Connection Window

4 Conclusions

3DEV is still under development but already constitutes a very useful tool for the electroacoustic music composers.

Further developments will address the design of more connections with other Spatial Audio Software, as the one shown in the point 3.1.3. Also, other development possibilities would be to include several drawing and transformation

routines of 3D trajectories as well as the storing of the data in diverse formats for their use with other spatialisation programs. Having the data accurately generated and stored, it would be a simple matter to convert it to different coordenate systems, as well as to adapt it to the different syntax requirements and data types used by other spatialisation software.
Finally, as the visualization of many complex spatial trajectories operating together may be difficult to achieve, a real-time animation plot of them sinchronized with their audio files associated is also a feature to be further developed.

Acknowledgements

To the Universidad Nacional de Quilmes and the National Agency of Scientific and Technological Promotion of Argentina, for granting financial support of this research.

References

[1] Trevor Wishart. 1996. *On Sonic Art.* Harwood academic publishers, UK

[2] Karlheinz Stockhausen. 1955. *Kontakte*, text published in the CD Nº3 Electronische Musik 1952-1960, Stockhausen Verlag, Alemania.

[3] John Chowning. 1971. *The simulation of Moving Sound Sources.* Journal of the Audio Engineering Society, 19.

[4] F.R. Moore. 1983. *A General Model for Spatial Processing of Sounds.* Computer Music Journal, Vol. 7, No.3.

[5] F.R. Moore. 1989. *Spatialisation of sounds over loudspeakers.* In: M. Mattews and J. Pierce: *Current Directions in Computer Music Research*, MIT Press, Cambridge, Mass.

[6] F.R. Moore. 1990. *Elements of Computer Music*. Prentice Hall., New Jersey.

[7] Kendall et al. 1989. *Spatial reverberation. discussion and demonstration.* In: M. Mattews and J. Pierce: *Current Directions in Computer Music Research*, MIT Press, Cambridge, Mass.

[8] Richard Furse. 1995). *Spatialisation Tools*. (http://www.muse.demon.co.uk/csound.html)

[9] Richard Furse. 1999). *Vspace.* (http://www.muse.demon.co.uk/vspace/vspace.html)

[10] F. Lezcano. 1992. *Quad sound playback hardware and software.* C.C.R.M.A., Stanford University.

[11] Richard Karpen. 1998. *Space and Locsig Ugs.* In The Csound Manual, http://www.csounds.com/manual/html/locsig.html, http://www.csounds.com/manual/html/space.html.

[12] Ville Pulkki. 2001. Spatial *sound generation and perception by amplitude panning techniques*, Technological University of Helsinki, Report N°62, Finland.

[13] Itzvan Varga. 2000. *Spat3d Unit Generator*, In *The Csound Manual*, http://www.csounds.com/manual/html/spat3d.html

[14] M. Wozniewski, Et al. 2007. *AudioScape: A Pure Data library for management of virtual environment and spatial audio*. Pure Data Convention, Montreal, Canada.
http://www.audioscape.org/twiki/pub/Audioscape/AudioscapePublications/audioscape_pdconv07_final.pdf

[15] Thomas Musil. 2007. *Binaural-Ambisonic 4.Ordnung 3D-Raumsimulationsmodell mit ortsvarianten Quellen und Hörerin bzw. Hörer für PD*, IEM Report Nº38/07

http://iem.at/projekte/publications/iem_report/report38_07/report38_07

[16] Rene Causse, et al. 2002. *Directivity of musical instruments in a real performance situation*, ISMA Proceedings.

[17] Eric Harlow. 1999. *Desarrollo de Aplicaciones Linux con GTK+ y GDK*, Prentice Hall, Madrid.

[18] Bill Schottstaedt. (last accessed 01-01-2010): *SNDLIB*,

http://www.ccrma.stanford.edu/software/snd/sndlib/

[19] Dave Malham. 2009 (Spanish translation by Oscar Pablo Di Liscia). *El espacio acústico en tres dimensiones y su simulación por medio de Ambisonics*, In the book: Música y espacio: Ciencia, tecnología y estética, Colección Música y Ciencia, Editorial UNQ, Buenos Aires, Argentina.

Sense/Stage — low cost, open source wireless sensor and data sharing infrastructure for live performance and interactive realtime environments

Marije A.J. BAALMAN
Harry C. SMOAK
Vincent DE BELLEVAL
Brett BERGMANN
Christopher L. SALTER
Design and Computation Arts
Concordia University
Montréal, Québec
Canada,
marije@nescivi.nl and sensestage@gmail.com

Joseph MALLOCH
Joseph THIBODEAU
Marcelo M. WANDERLEY
Input Devices and Music Interaction Lab
McGill University
Montréal, Québec
Canada,
joseph.malloch@gmail.com

Abstract

SenseStage is a research-creation project to develop a wireless sensor network infrastructure for live performance and interactive, real-time environments. The project is motivated by the economic and technical constraints of live performance contexts and the lack of existing tools for artistic work with wireless sensing platforms. The development is situated within professional artistic contexts and tested in real world scenarios.

In this paper we discuss our choice of wireless platform, the design of the hardware and firmware for the wireless nodes, and the software integration of the wireless platform with popular media programming environments by means of a data sharing network, as well as evaluation and dissemination of the technology through workshops. Finally, we elaborate on the application of the hardware and software infrastructure in professional artistic projects: two dance performances, two media projects involving environmental data and an interactive, multisensory installation.

Keywords

Wireless sensing, mesh networks, data sharing, realtime performance, interactive environments.

1 Introduction

SenseStage is a research-creation project to develop small, low cost and low power wireless sensor hardware together with software infrastructure specifically for use in live theater, dance and music performance as well as for the design of interactive, real-time environments involving distributed, heterogeneous sensing modalities. The project consists of three components:

- a series of small, battery powered wireless PCBs that can acquire and transmit input from a range of analog and digital sensors,
- an open source software environment that enables the real-time sharing of such sensor data among designers and
- plug in modules that enable the analysis of such sensor data streams in order to provide building blocks for the generation of complex dynamics for output media.

The project emerged from a desire to address a novel, emerging research field: distributed, wireless sensing networks for real-time composition using many forms of output media including sound, video, lighting, mechatronic and actuation devices and similar.

Three specific factors have motivated the SenseStage project:

1) *Economic and technical constraints of live performance:* There is an increasing interest in the use of sensing technology in live performance contexts. The economic and cultural constraints of live performance, however, make the integration and use of such technologies difficult. It is seldom possible to have long rehearsal periods with full access to a technical setup that is equivalent to the eventual performance space due to the industrial model of cultural production — show in, show out – leaving no room for exploration of new technological possibilities and the artistic impact. By providing a solution for easy application of the technology, the short rehearsal periods and "techweeks" can be spent more on the artistic exploration of the technology, rather than solving technological problems.

2) *Lack of tools for artistic use:* There are many groups currently researching the applications of wireless sensing networks, but their design decisions are normally motivated by en-

gineering innovations, thus leading to efficient yet, prohibitively expensive and complex systems. Also, despite the large number of research projects in the field, there are few wireless sensing platforms that are actually available for real world use, or that are affordable for artists. In addition these wireless sensing platforms rarely integrate with the software tools that artists use for making music, sound and media.

3) *Real world testing scenarios:* Much of the research agenda for the project was driven by many years of artistic work and technological development of tools to facilitate the creation of interactive performances and installations. These events employed distributed sensing and mapped such input data to complex parameter spaces for the control of sound and other media in real-time (e.g. *Schwelle* [Baalman et al., 2007] and *TGarden*[Ryan and Salter, 2003]). A key design element of the SenseStage project is thus to deploy SenseStage technologies into real world, professionally driven testing environments to see how such tools function "in the wild" and outside of the standard lab, demo-driven mode normally given to the presentation of new technologies.

2 Hardware

For the hardware wireless sensing node for Sense/Stage, our main requirements were cost per unit — since low costs allow experimentation with large numbers of nodes — and immediate availability. We investigated several options for wireless nodes developed by other groups, such as the μParts[1] [Beigl et al., 2005], the EcoMote[2] [Park and Chou,], the Tyndall Motes[3] and the Intel Motes[4][Nachman et al., 2005], but none of these satisfied our needs, or more importantly they were simply not available.

Our design goals were:

- Low cost
- Small form factor
- Flexible sensor configuration
- Usable for control of motors, LEDs, and other actuators.
- Operable in large groups (10+ nodes)
- Long battery life
- Ease of use
- Programmable, so that the board can take care of more logic and processing of data, if desired by the user

We decided to use the XBee in combination with the Arduino[5] platform, as the XBee already provided us with the needed ad-hoc network structure. Additionally, several other developers have documented their experience using XBees in conjunction with Arduinos[6] allowing us to skip some common development pitfalls.

We based our board design on the Arduino Mini Pro, being able to tap into many available firmware libraries, as well as the development and programming environment. Furthermore, Arduino is widely used in open source, artistic, physical computing contexts, so our board will be easy to use for this community.

The main focus then was to design a PCB that was small, and to develop standard firmware that makes it easy to setup and use the boards, as well as exploring the use of and integration with the XBee wireless chips.

The PCB layout of the SenseStage MiniBee is shown in figure 1. The first board revision came to a unit cost price of about 32 CAD, excluding the XBee chip, for a manufacturing run of 100 boards (PCB creation, assembly and parts). With a larger manufacturing run and allowing for a longer assembly time, this per unit cost will drop considerably.

2.1 Firmware

The firmware is a collection of functions to handle wireless transmission and communication, sensor reading and basic read/write operation on any available pin of the MiniBee. The firmware is built using our own library for the Arduino environment which allows users to quickly build their own application for the MiniBee.

Currently the following sensors/actuators are supported by the firmware:

- Analog sensors (connected to the analog input pins, e.g. resistive sensors, analog accelerometers, infrared distance sensors)

[1] `http://particle.teco.edu/upart/`

[2] `http://www.ecomote.net/`

[3] `http://www.tyndall.ie/mai/WirelessSensorNetworksPrototypingPlatform_25mm.htm`

[4] `http://techresearch.intel.com/articles/Exploratory/1503.htm`

[5] `http://www.arduino.cc`

[6] e.g. the ArduinoXbeeShield`http://www.arduino.cc/en/Main/ArduinoXbeeShield`, the Arduino Xbee Interface Circuit (AXIC) `http://132.208.118.245/~vitamin/tof/AXIC/` and the Blushing Boy MicroBee R3 `http://blushingboy.org/content/microbee-r3`

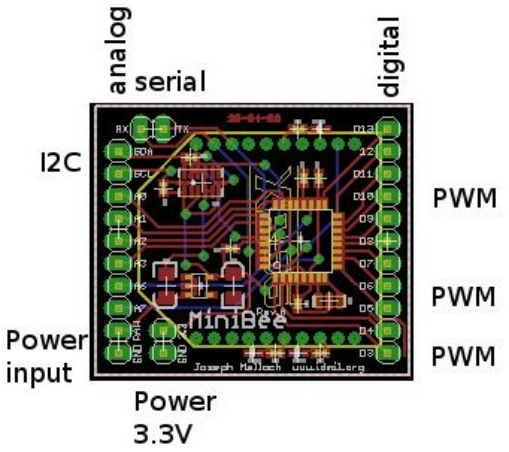


Figure 1: The SenseStage MiniBee PCB, rev. A. The XBee is mounted on the other side of the board. Changes in the second revision include smaller board-size and the footprint for a coin-cell.

- Digital sensors (on/off, e.g. buttons and switches)
- LIS302DL accelerometer[7], using I2C
- Relative humidity and temperature sensor[8]
- Ultrasound sensors[9]
- PWM output (e.g. dimmable LEDs, motors)
- Digital output (on/off)

The serial protocol is loosely based on the Serial Line Internet Protocol (SLIP)[10], and is set up as simply as possible to ensure that data packets are small. A regular package is built up as follows:

- escape character, followed by message type
- message ID
- node ID
- data bytes
- delimiter character

A full reference of the different messages supported is given in table 1.

The firmware is configured through a host computer which allows to quickly change its operation without having to physically reprogram the microcontroller. This approach is not unlike Firmata [Steiner, 2009] with the difference that our firmware stores its latest configuration in the EEPROM of the MiniBee.

Each time the MiniBee boots up, it reads the serial number of the attached XBee[11] and relays this information to the coordinator node connected to the host computer. The host then assigns a unique node ID to the MiniBee and optionally sends a new configuration to the MiniBee. If no new configuration is received, the MiniBee can access its configuration through its EEPROM. The host computer software remembers the known boards and XBee serial numbers so node IDs and configurations are maintained for a project.

Using an 8 MHz clock on the board, the maximum baud rate that can be achieved is 19200 baud. In a next revision we will include a faster (up to 20MHz) crystal, which will allow the use of higher baudrates.

Future work also includes writing a wireless bootloader to fully reprogram the SenseStage MiniBee without the need to manipulate the boards (see for example LadyAda[12]). The extra components needed to do this will be added in the next revision of the board.

3 Software

In order to make the data from the wireless sensor nodes available to several collaborators on a project simultaneously, we developed the SenseWorld DataNetwork. It is intended to facilitate the creation, rehearsal and performance of collaborative interactive media art works, by making the sharing of data (from sensors or internal processes) between collaborators easy, fast and flexible. Our aim is to support multiple media practices and allow different practitioners to use the software to which they are accustomed. The framework is intended to support *coordinated collaboration* with real-time data and multiple media types within a live interactive performance context.

The framework is different from the *KeyWorx*[13] framework [Doruff, 2005], which emphasizes net-based art and collaborative projects between different locations, and the McGill Digital Orchestra Tools[14] [Malloch et

[7]There is a footprint on the board for this sensor.

[8]Sensirion SHT1x series

[9]The "Ultrasonic Ranger", `http://www.robot-electronics.co.uk/htm/srf05tech.htm`

[10]`http://tools.ietf.org/html/rfc1055`

[11]using the AT command mode. Other people have made an Arduino library for communicating with XBees in API mode. See `http://code.google.com/p/xbee-arduino/`

[12]`http://www.ladyada.net/make/xbee/arduino.html`

[13]`http://www.keyworx.org`

[14]There is a Max/MSP bridge between the SenseWorld DataNetwork and the Digital Orchestra Tools so they

description	type	data	sender
Announce	'A'		server
Quit	'Q'		server
Serial number	's'	Serial High (SH) + Serial Low (SL)	node
ID assignment	'I'	msg ID + SH + SL + node ID + (*config ID*)	server
Configuration	'C'	configuration bytes	server
PWM	'P'	node ID + msg ID + 6 values	server
Digital out	'D'	node ID + msg ID + 11 values	server
Data	'd'	node ID + msg ID + N values	node

Table 1: Message protocol between host and MiniBee nodes. Type is preceded by the escape character (92), and messages are delimited with the delimiter character (10). The escape character is used to escape to the message type, and whenever the delimiter, the escape character, or the carriage return character (13) occurs in any of the other bytes. General convention for the message type: server message in upper case, node message lower case. (*...*) indicates an optional byte.

al., 2008], which primarily focus on the mapping and performance of monolithic digital musical instruments.

The final design criteria were to:

- Tight integration with the wireless sensing platform
- Allow reception of data from any node by any client (subscription)
- Allow transmission of data to any node by any client[15] (publication)
- Restore network and node configuration quickly
- Be usable within heterogeneous media software environments
- Enable collaboration between heterogeneous design practices
- Enable efficiency of collaboration within the limited timeframe of rehearsals

3.1 The SenseWorld DataNetwork framework

The framework's core is implemented in SuperCollider (SC), which is available as open source software and runs on several platforms (Linux, OSX, Windows and FreeBSD). Clients have been implemented in SC, PureData, Max/MSP, Processing, Java, and C++ so far. The OSC namespace is well defined, so it should be trivial to implement clients for other software environments.

What follows is a technical description of the implementation of the framework. Users of the framework need not be familiar with the inner workings of the framework (or have experience in programming SC), so long as their software environment supports OSC and is able to comply with the OSC namespace conventions in order to communicate with the network. Thus, the framework is designed to allow for ease of use within a designer's own creative practice.

A central host receives all data messages and manages the client connections (see figure 2). Each client can subscribe to one or more data *nodes* in order to use that node's data in its own internal processes. Furthermore, each client can publish data onto the network by creating a node. A new client can query the network concerning which nodes are present and is informed when new nodes appear after the client has been registered.

A *data node* can be understood as a collection of data that belongs together, e.g., data coming from the same sensor device or the output of a particular device such as the DMX control stream for theatrical light. Within each node there are *slots* which represent single data values, for example, a data node representing a 3-axis accelerometer has three slots, with each slot corresponding to one axis. If a client is only interested in one slot of a node, he can subscribe specifically to that slot.

3.2 Integration with MiniBee mesh network

We have specifically integrated the connection to the SenseStage MiniBee network, by providing options to configure, send and receive data from the sensor network, through the host of the DataNetwork.

The host communicates with the wireless network through a serial protocol. It manages all

can be used together.

[15] only one client can set data to a specific node at a time.

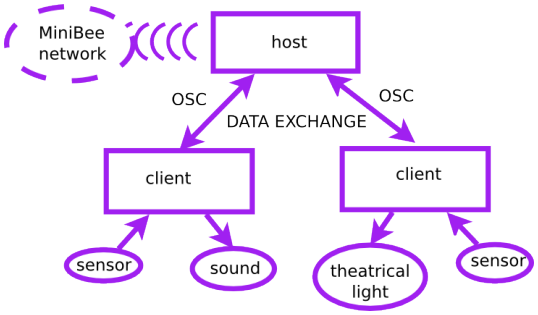


Figure 2: Diagram of the SenseWorld DataNetwork structure.

the incoming and outgoing data from and to the wireless nodes. The data from a MiniBee will appear onto the network as a DataNode, while each attached sensor is a *data slot* in this *data node*. Clients can then subscribe to this *node* to receive its data. Furthermore, a client can send a message to map the data that is on a *data node* the client has created to the digital outputs or pulse width modulation outputs of a MiniBee, in order to control, for example, lights or motors.

3.3 OSC implementation

The network is accessed through an OSC interface[16], which allows a client to join the data network, access its data and also create its own data nodes on the network.

The general setup is as follows: a client first sends a registration message to the data network server. The client will immediately begin receiving *ping* messages to which it must reply with *pong* messages confirming the client's presence. Following the initial registration, the client can submit a query message in order to receive a complete list of nodes and slots currently available from the network. The client can then subscribe to selected nodes and slots, and subsequently will receive data from the nodes and slots it is subscribed to via data messages corresponding to the subscribed data sources.

The client can supply a new node to the network by using the `/set/data` message (which is also used subsequently to set new data). A client can also label the nodes and slots it has created. Whenever a new node or slot is added (by any client) or changed (e.g., when it gets a label), the client will receive a new info message automatically. All messages to the server have a reply, which is either the requested info, a confirmation message or a warning or error.

In comparison, the Digital Orchestra tools provide a decentralized network using one general multicast port on the network to settle the ports and namespaces of clients. Since not all interactive media software environments support listening to multicast channels, we elected not to use this approach. Rather our assumption is that each client will settle its listening port itself within the operating system of the computer on which it runs. The host can then distinguish each client by its IP address (automatically included in the OSC message) and a port which is an argument of each OSC message in the namespace. The latter is necessary as several clients (Max/MSP among them) do not send OSC messages from the same port as they are listening to, or cannot configure the port they are sending from.

3.4 Auto-recovery

Practical lessons derived from rehearsal and performance experiences remind us that software applications and processes can be unexpectedly and fatally interrupted (i.e., crash). For this reason, a fast and automatic recovery of all previously instantiated connections is critical. The following methods are implemented to enable fast and automated recovery in such situations. Following (re)start of the host server and (re)establishment with the network, an announce message is broadcast on several ports. In addition, the server updates a publicly readable file with the current active listening port. Moreover, the host can restore previously connected clients from information stored in a server readable file. In turn, the client has read access over the network to the host configuration file and automatically retrieves the port information to which it has to register. Further, the client implements an auto-configuration process triggered upon receipt of a host announce message and a response from the host indicating the client has successfully registered.

3.5 SuperCollider implementation

The SuperCollider (host) implementation is done via a set of custom written classes:

SWDataNetwork base class for the network.
SWDataNode base class for a data node.
SWDataSlot base class for a data slot.
SWDataNetworkSpec implements the labelling of the nodes and slots of the net-

[16] assumed to be used via the UDP layer. The OSC protocol in itself is not dependent on the underlying protocol, but is implemented on top of TCP and/or UDP in most software systems

work.

SWDataNetworkOSC implements the OSC interface

SWDataNetworkOSCClient keeps track of a connected client

Data nodes have both unique IDs (integer numbers) and human readable labels (e.g., node "3" has the label "accelerometer"). Data slots are automatically numbered, according to the order in which they appear, as they are set in the network; they can also be given a label. The labelling is not done automatically, so that the naming becomes a conscious and integral part of establishing shared nomenclatures for the collaboration. This encourages consideration by the users of what the data represents and its potential use. The label specification (the "spec") can be stored between sessions so it can be recalled again upon startup.

Each data node and slot has methods to print debugging messages, set an action to be performed upon new incoming data, scale and/or remap the data and create a control bus on the audio server[17] with the data. Each data node also can specify an action to be taken in case there has been no input to the node for a certain amount of time (e.g., trying to reconnect to an external device).

If a client creates a node, that node is linked to the client (the client becomes the "setter" of the node), and no other client can set data to that node. Client configuration can also be stored to a file and be used for recovery on startup.

All data from the network can be written to a log-file in a text format containing lines for each time step with tab-separated data values. The log can be opened and played back with the class **SWDataNetworkLog**.

Finally, a graphical user interface has been implemented to monitor the status of the data network and the state of each node (see figure 3 for the client's version of this GUI).

3.5.1 The client

The class **SWDataNetworkClient** implements an OSC client to the SenseWorld DataNetwork so that an external SC client can also be part of the network. It implements the client side of the OSC interface. It inherits from the class **SWDataNetwork** to manage

[17] SC consists of two parts: *sclang*, which is the programming language, and *scsynth*, which is a dedicated audio server.

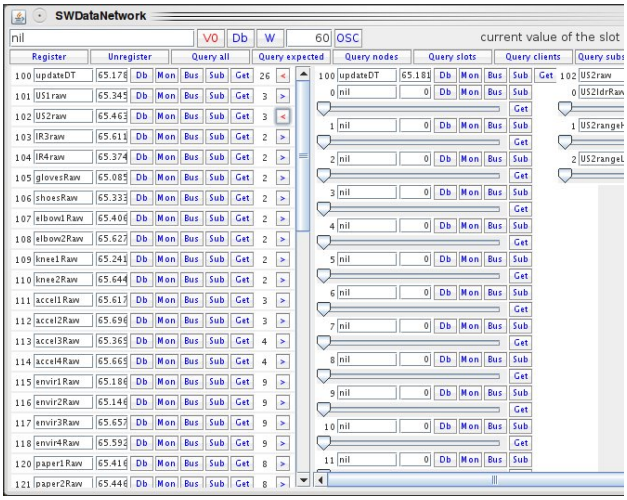


Figure 3: The DataNetwork SC Client interface.

the data it receives, and thus has the same interface in its use within SC, with some additional methods for interaction with the host.

In a setup with several collaborators using SC, one of them will act as a host for the network, while the other SC participants register as clients to that host. Since the code interface for both is almost the same, apart from the initialisation, it is very easy for SC participants to prepare and test their own inputs and outputs to the DataNetwork individually, and switch to the shared network during collective rehearsals. In figure 4 an example is given of the code interface to both the host and the client.

3.5.2 Derived data

Deriving data from existing nodes can be done for example by combining data from various nodes, calculating statistical properties of data, or smoothing data.

To facilitate this, we have developed methods to do this either making use of the language features of SC, or by making use of the server's unit generators to perform these calculations.

3.6 PureData and Max/MSP clients

Two abstractions provide access to the SenseWorld DataNetwork within a Pd or Max patch: **dn.node** and **dn.makenode**. Both implementations require a private variable for the host IP address and for the client name (shared between all instances on a client), but otherwise take care of the details of talking to the network.

dn.node subscribes to an existing node, outputting the received data as a list. In Pure Data each instance subscribes to a single node, whereas in Max it is possible to receive data from multiple nodes using a single **dn.node** ob-

Example of setting up the host in SuperCollider:

```
// create an instance of the DataNetwork
x = SWDataNetwork.new;
// adds the host OSC interface
x.addOSCInterface;
// create an instance of the class
// managing the MiniBees:
q = SWMiniHive.new( x );
// make GUIs for the DataNetwork and the hive:
x.makeGui;
q.makeGui;
```

An example of using the client in SuperCollider:

```
// create a DataNetwork client with the name ‘‘sc’’:
x = SWDataNetworkClient.new(‘‘192.168.0.104’’,‘‘sc’’);
// see what is in the network
x.queryAll;
// we know that node 2 has interesting data coming
// from a floor pressure sensor, so we subscribe:
x.subscribeNode( 2 );
// we want to create a node with a measure of the
// sample variance on node 102 and give a label
x.addExpected( 102, \floorVar );
// node 2 has the label \floor to access it
// we create a bus for the data (s is default server):
x[\floor].createBus( s );
// we create the sample variance node:
~floorVar = StdDevNode.new( 102, x, x[\floor].bus, s );
~floorVar.start;
// now we have raw data available on node 2, x[\floor]
// and sample variance data on node 102,  x[\floorVar]
```

Figure 4: The DataNetwork SC code interface.

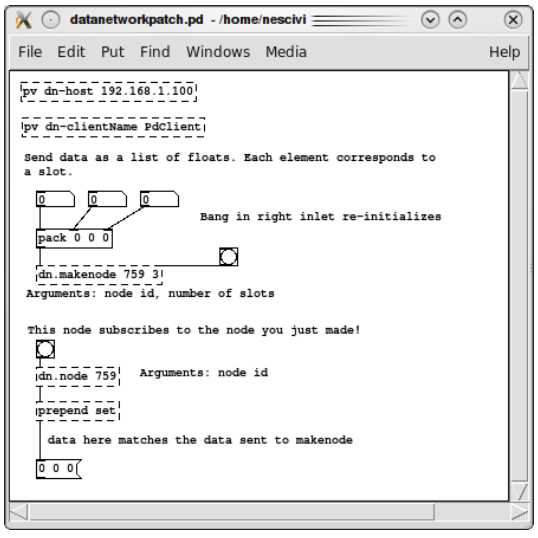


Figure 5: Screenshot of the PureData client.

ject (in which case each node's data list is preceded by its node ID).

dn.makenode does the opposite, publishing data to the network using a unique node ID. The data must be formatted as a list with a number of elements equal to the number of data slots given as a creation argument to the instance.

```
import datanetwork.*;
DNConnection dn;  // DNConnection instance
DNNode node;      // DNNode instance

void setup() {
  dn = new DNConnection(this, "192.168.0.104",
         dn.getServerPort("192.168.0.104"),
         6009, "p5Client");
  node = new DNNode(2000, 5, 0, "p5Node");
}
void stop() {
  dn.unsubscribeAll();
  dn.removeAll();
  dn.close();
}
void keyPressed() {
    if(key == 'r') dn.register();
    else if(key == 'q') dn.queryAll();
    else if(key == 'f') dn.subscribeNode(401);
    else if(key == 'd') dn.setData(node,
      new float[] { 4.0, 2.0, 1.0, 2.3, 4.4 } );
}
// receive and print data:
void dnEvent(String addr, float[] args) {
    print("Float: " + addr);
    for(int i = 0;i < args.length;i++)
        print(" "+args[i]);
    println();
}
```

Figure 6: An example of using the Processing client.

3.7 Processing and Java

The Processing client is based on the JavaOSC library[18], and is based around two classes: **DNConnection**, dealing with the communication to and from the host, and **DNNode**, dealing with the properties of a DataNode. As the implementation is basically a Java class, the client can not only be used for Processing, but also for any Java program.

An example of use of this library in a program is shown in figure 6.

3.8 C++ library

The C++ library is based upon *liblo*[19] for handling OSC communication and consists of a set of classes to deal with the various components of the DataNetwork:

DataNetwork base class for the network.
DataNode base class for a data node.
DataSlot base class for a data slot.
DataNetworkOSC implements the OSC interface.
OSCServer C++ class to implement an OSC Server (taken from *swonder*[20] and slightly expanded)

[18] http://www.illposed.com/software/javaosc.html
[19] http://liblo.sourceforge.net
[20] http://swonder.sourceforge.net

```
   DataNode * node;
   DataNetwork * dn;

// create a data network:
   dn = new DataNetwork();
// create an osc client interface for it:
// arguments (host IP, client udp port, client name)
   dn->createOSC( ``127.0.0.1'', 7000, ``libdn'' );
// register with the host:
   dn->registerMe();
// query all there is to know about the network
   dn->query();

// subscribe to a node:
   dn->subscribeNode( 5, true );

// create a node:
   dn->createNode( 4, "world", 5, 0, true );
// label one of its slots:
   dn->labelSlot( 4, 2, "hithere", true );

float dummydata[] =  {0.1, 0.3, 0.4, 0.5, 0.6};
// get a reference to the node:
   node = dn->getNode( 4 );
// set data to the node:
   node->setData( 5, dummydata);
// send the data to the network:
   node->send( true );
```

Figure 7: An example of using the C++ library.

An example of use of this library in a program is shown in figure 7.

4 SenseStage Workshop

The first SenseStage workshop[21] was held in May 2009 at Concordia University, as a test case for using many sensor nodes in one space, as well as having a number of artists, unfamiliar with the specifics of the technology and coming from diverse artistic and technical backgrounds, use the DataNetwork simultaneously. Participants were able to employ all available data in various projects, which were developed over the course of one week. While this workshop served as a test case for evaluating the use of our hardware and software, we were also interested in how participants would make artistic use of the potentials of the system.

The workshop resulted in five group projects, with groups consisting of 3 to 5 collaborators, using light, sound, animation and video as output media. The groups were able to go very quickly from concept to experimenting with various sensor modalities and using the data to drive the output media. Encouragingly, the participants seemed to be primarily concerned with "what to do with the data" rather than "how to get the data". The results of this workshop also highlighted the need for more sophisticated ways of dealing with sensor data, for combining, conditioning, and processing of multiple data streams.

Several more SenseStage workshops are planned for 2010 and 2011 as a means to familiarize people with the SenseStage infrastructure, as well as further explore issues in mapping and using large numbers of datastreams.

[21] http://sensestage.hexagram.ca/workshop/

5 Usage cases

In this section, we will discuss several projects in which the SenseStage technology has been used. All of these projects have been, or are being shown at international festivals and multiple venues, thus fullfilling the criterium that the technology has to be useable in a real world, professional artistic environment.

5.1 Dance: Schwelle and Chronotopia

The dance performance Schwelle [Baalman et al., 2007] had been developed before the SenseStage project began and has informed many of the design decisions made during the SenseStage project, so we are currently discussing how the infrastructure can be adapted and improved to use our new technology for future performances. The performance involves 3 accelerometers on the body (originally wireless based upon a Create USB interface with MicroChip RF chips), 1 accelerometer in an object (originally a WiiMote), and 3 light sensing boards (originally wired Create USB interfaces) placed at various places in the room. Furthermore, there is activation of custom lights and motors in one part of the stage. Using the SenseStage MiniBees, we will be able to use now 3 separate sensing boards on the body, saving us problems of wiring along the body; instead of using the WiiMote for the accelerometer in the object, we can now use a SenseStage MiniBee, which we can wake up from a sleep mode, once we need the sensing in the box; this will save us various problems of having to make the WiiMote set up a BlueTooth connection by pushing buttons, while it is packed inside a box. For the sensing boards inside the room, we will also be spared of the wiring. Where we were previously using a custom OSC-namespace for this piece, we can now use the DataNetwork clients instead, and gain much more robustness with regard to reconnecting; also it will be much faster to add new data to exchange between the interactive light controls and the sound control, should we feel the need to do so.

Figure 8: A still from the dance performance Chronotopia with the Attakkalari Centre for Movement.

In Chronotopia, a dance performance by the Bangalore (India) based Attakkalari Centre for Movement, and in collaboration with visual artist Chris Ziegler, we used the wireless technology for controlling a matrix of 6 by 6 cold cathode fluorescent lights (CCFL), and 3 handheld CCFL lights. Since the power required for the light matrix is quite high, it cannot be battery-powered, but the use of wireless technology freed us from running cable between the light matrix and the computer controlling it. Given the short setup time in theaters (usually just one day), and especially in technically challenging environments as India, this was a considerable advantage. For the 3 handheld objects, wireless control was critical, as the objects are carried across the stage by the performers during the show as part of the dramaturgy of the piece. Within the light control setup itself, the DataNetwork was used extensively to exchange data between different portions of the setup, such as the motion tracking data (from a camera looking down at the stage), and pitch and beat tracking data extracted in real-time from the soundtrack. We also exchanged data between the light control and the interactive video, both for synchronisation of cues with the soundtrack (using frametime of the playback, published as data on the network), and for connecting the intensity of the lights to the video image (the light control was publishing the maximum output value of all the lights in the matrix onto the network, which was used to control the brightness of the video image).

5.2 Environmental: MARIN and Arctic Perspective

In two artistic projects dealing with environmental data,
MARIN[22] and Arctic Perspective Initiative[23], the SenseStage MiniBees were used to gather environmental data, such as temperature, humidity, light and air quality, as well as 3-axis acceleration. The DataNetwork was used to access the data for real-time use, and to gather and log all the data to file for artists to use at a later time for visualisation and sonification.

From an expedition to Nunavut in Northern Canada in Summer 2009 as part of the Arctic Perspective Initiative project we learned that the range of the XBee and XBeePros is very much dependent on the environmental conditions. In the outside conditions there, the achieved range of transmission was only a few hundred meters, only about a fifth of the range specified on the datasheet of the XBeePro. At ranges larger than about 50 m. they are extremely affected by blockage, e.g. from the body. In indoor situations, the radio waves will be reflected by objects and walls and such blockage may be mitigated.

Additionally, the batteries lost charge faster in colder conditions (about 6°C) resulting in shorter battery life.

Another issue was that it was difficult to power the host computer, receiving the data from the MiniBees, using solar power, because of foggy and cloudy days. For this reason a kind of datalogger approach for the MiniBees, which stores data locally and sends it to a host when the host is online, could be useful for this kind of application.

Other challenges include finding waterproof housing to protect the electronics from extremely wet weather conditions on sea.

5.3 Installations: JND/Semblance

JND/Semblance is an interactive installation that explores the phenomenon of cross modal

[22] "M.A.R.I.N. (Media Art Research Interdisciplenary Network) is a networked residency and research initiative, integrating artistic and scientific research on ecology of the marine and cultural ecosystems." (from `http://marin.cc/`).

[23] "The Arctic Perspective Initiative (API) is a nonprofit, international group of individuals and organizations whose goal is to promote the creation of open authoring, communications and dissemination infrastructures for the circumpolar region." (from `http://www.arcticperspective.org`).

perception — the ways in which one sense impression affects our perception of another sense. The installation comprises a modular, portable environment, which is outfitted with devices that produce subtle levels of tactile, auditory, visual and olfactory feedback for the visitors, including a floor of vibrotactile actuators that participants lie on, peripheral levels of light, scent and audio sources, which generate frequencies on the thresholds of seeing, hearing and smelling.

In JND/Semblance the SenseStage MiniBees are used for gathering floor pressure sensing data. The SenseWorld DataNetwork is used to gather the raw sensor data, to extract features from it, and to establish flexible mappings to light, sound, and vibration on a platform on which the visitor is lying down.

6 Conclusions

We have presented SenseStage, an integrated hardware and software infrastructure for wireless mesh-networked sensing, actuating, data sharing and composition within interactive media contexts. The infrastructure is unique as it integrates hardware and software, and makes sensor and other data easily available for all collaborators in a heterogeneous media project, within each collaborator's preferred software environment.

We are currently revising the hardware and firmware design, including options to configure and program the boards wirelessly. We plan to have the board available for sale in the second half of 2010. Our future research will focus on techniques for composing and creating with the many streams of realtime data available from such a dense network of sensors.

7 Acknowledgements

This work was supported by grants from the Social Sciences and Humanities Research Council of Canada and the Hexagram Institute for Research/Creation in Media Arts and Sciences, Montréal, QC, Canada.

Thanks to Matt Biedermann for his feedback on the use of the SenseStage MiniBees in the Arctic Perspective Initiative project.

Thanks to Elio Bidinost, as well as all the SenseStage Workshops participants, for their input and feedback.

7.1 Download

The SenseWorld DataNetwork is available from `http://sensestage.hexagram.ca`. It is released as open source software under the GNU/General Public License.

References

Marije A.J. Baalman, Daniel Moody-Grigsby, and Christopher L. Salter. 2007. Schwelle: Sensor augmented, adaptive sound design for live theater performance. In *Proceedings of NIME 2007 New Interfaces for Musical Expression, New York, NY, USA*.

M. Beigl, C. Decker, A. Krohn, T. Riedel, and T. Zimmer. 2005. uParts: Low cost sensor networks at scale. In *Ubicomp 2005*.

Sher Doruff. 2005. Collaborative praxis: The making of the keyworx platform. In Joke Brouwer, Arjen Mulder, and Anne Nigten, editors, *aRt&D: Research and Development in the Arts*. V2/NAI Publishers, Rotterdam.

Joseph Malloch, Stephen Sinclair, and Marcelo M. Wanderley. 2008. A network-based framework for collaborative development and performance of digital musical instruments. In Richard Kronland-Martinet, Solvi Ystad, and Kristoffer Jensen, editors, *Computer Music Modeling and Retrieval. Sense of Sounds: 4th International Symposium, CMMR 2007, Copenhagen, Denmark, August 2007, Revised Papers*, number ISBN 978-3540850342 in Lecture Notes in Computer Science. Springer.

L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel. 2005. The intel mote platform: a bluetooth-based sensor network for industrial monitoring. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005) (Los Angeles, CA, April 2005)*.

Chulsung Park and Pai H. Chou. Eco: Ultra-wearable and expandable wireless sensor platform. In *Third International Workshop on Body Sensor Networks (BSN 2006)*.

J. Ryan and Christopher L. Salter. 2003. Tgarden: Wearable instruments and augmented physicality. In *Proceedings of the 2003 Conference on New Instruments for Musical Expression (NIME-03), Montreal, CA*.

Hans-Christoph Steiner. 2009. Firmata: Towards making microcontrollers act like extensions of the computer. In *Proceedings of NIME 2009 New Interfaces for Musical Expression, Pittsburgh, PA, USA*.

EDUCATION ON MUSIC AND TECHNOLOGY,

A PROGRAM FOR A PROFESSIONAL EDUCATION

Hans Timmermans, Jan IJzermans, Rens Machielse,

Gerard van Wolferen, Jeroen van Iterson.

Utrecht School of the Arts, Utrecht School of Music and Technology,

PO-BOX 2471, 1200 CL Hilversum, The Netherlands.

ABSTRACT

We describe the development and the maintenance of a program for a professional education at Utrecht School of Music and Technology. The program covers most of the field and offers various degrees up to PhD level. The program was developed over the last 23 years and is updated on a yearly basis. We deliver about 80 graduates a year to work, survive and keep up with developments. 92 % of our students develop a healthy career after graduation. Music technology as a field of studies is in constant and rapid development and because of that the characteristics of 'the professional' in the field are changing very rapidly too. For this reason we have built in mechanisms to enforce regular updates of the program and to develop the knowledge and skills of the teaching staff.

1. INTRODUCTION

Utrecht School of Music and Technology is a Dutch Professional Education in the broad field of Music and Technology. Our School is part of the Utrecht School of the Arts, a University with 5 faculties (Visual Arts, Music, Theatre & Drama, Art & Economy and our faculty of Art, Media & Technology), around 3600 students and 600 lecturers. Within the School of Music and Technology we offer several tracks in the fields of:

- Sound Design.
- Composition for the Media.
- Composition for non-linear Media
- Composition of Electronic Music.
- Composition & Music Production.
- Composition & Music Technology
- Music Production & Performance.
- Music Technology & Performance.
- Audio Design, Music System Design
- Audio Design, Recording and Production

Some of these programs have a strong emphasis on the Arts; some have a more technical focus. The programs partially overlap. Currently the School has over 350 students and a teaching staff of 35; we offer bachelor, master and research degree programs. The mission of the School is to educate students up to a level of skills and knowledge and to a level of professional attitude good enough to work and survive within the Dutch context and the international context. The emphasis of the different programs is on the personal development of the student; to help her to earn a living and to establish a professional career in which she is able to continue the professional development and keep up with the constant and rapid developments in the field of Music and Technology.

1.1. History

Our School started as a department of Utrecht Conservatory of Music in 1985. Our current program started in 1987. From then on we have been updating to synchronize with the outside world on a very regular basis to obtain a good match between the educational program and the demands of real existing professional careers. For the last 10 years we get figures like the 92% of our graduates that have a healthy career within music technology within two years after their master graduation. Pro year the School of Music and Technology admits around 90 students of whom around 85 will graduate in one of the many tracks we have in our program. On ICMC2003 we presented a paper on our program, on our School and on our educational philosophy [1].

1.2. Developing curricula and maintenance.

In the past six years some major changes have occurred, which made us adapt the curricula for Composition and the curriculum for Music Production. We are developing a curriculum for Composition for non-linear Media (Composition for Games fits in that track). Research within our School got a strong focus on applications of Music technology for grownups as well as children with special needs; research was integrated as a topic in most curricula.

Music technology as a field of studies is in constant and rapid development and because of that the characteristics of 'the professional' in the field are changing very rapidly too. We developed a very active alumni policy, we track all of our graduates, we make sure we know what types of work they do and in what type of context they work and we invite most of our alumni on conferences and as guest lecturers and advisors. We follow developments in the different contexts we educate for and we redesign parts of our curricula if changes in the actual contexts urge us to do. We have a small group of very experienced professionals acting as a taskforce to deal with misfits between our curricula and actual contexts, the total of curricula and teaching methods is adapted once every four years (every year we adapt parts of each degree-program). Every member of our teaching staff is involved in implementation and evaluation of newly designed curricula and methods.

2. VISION AND MISSION.

In our vision the most important topic to teach to a student is how to keep up with the constant developments of the field. We teach our students how to study and how to develop as a professional after graduation. To establish this attitude and to develop the skills to do so the teaching staff developed some formats for parts of the program:

- Lectures and classes, as they have been there for ages.
- Workshops in which a student learns from exercises accompany the lectures.
- Learning-projects in which the focus is on some aspects of Music technology.
- Hands-on sessions and practical assistance by older students.
- Study-groups in which students work together on theoretical and research issues.
- Study-groups in which students discuss each other's work in progress and reflect on their own work in progress.
- Industrial placement in companies or research Institutions.
- Real world projects with (paid) assignments from outside the School.
- Interdisciplinary projects with a strong emphasis on the production processes that are typical for the multidisciplinary setting of the specific project.
  (Some examples: Music Composition & Contemporary Dance, Interactive Composition & Theatre, Sound Design & Games)

The overall characteristics of our program:

- Students learn by doing,
- Students learn to be their own teacher.

We educate the student to professional independence with self-reflection as an important tool to keep up with new developments, to gain new insights and develop new concepts.

3. COMPOSITION FOR THE MEDIA

3.1. Contexts within the media industry

Looking at different contexts within the media industry such as television, advertisement, film (fiction, documentary, animation) and games, they all have some specific characteristics in common i.e. the speed and the unpredictability of developments. Both are caused by the availability and accessibility of digital means and distribution channels.
It's quite easy using free software to make your own radio- or TV-program and broadcast it through the internet. Consumers are becoming producers or – as they are called nowadays – *prosumers*. Another element of importance is the individualization of consumers. The traditional approach of the media industry through target groups is not valuable anymore because the average consumer has developed a very individual and therefore unpredictable behaviour. Additional is the given fact that media landscapes are constantly subject of discussion in politics on a national and on a European level. Parties concerned such as the advertisement industry and the television industry try to find solutions for the problems as described above in different ways. On the one hand one uses a very individual approach of the potential consumer (narrow casting). On the other hand a 'product' is published through all possible media channels and –forms so the consumer cannot escape from it (cross media publishing). In addition all kind of mechanisms and moments for measuring the effectiveness of a product are developed so the product can be re-adjusted: stand a chance of a disappointing commercial result should be minimal.

3.2. Role and function of music in the media industry

Taking a closer look at the role and function of music in the media industry, it is precisely this effectiveness that has been a problem in the past. There was no way to hear and check a final score for a film or TV-commercial in advance because usually only a piano version existed to decide upon (the composer would play the score on the piano and would explain the orchestrations verbally to the director or advertisement agency). Along with digital technology came the possibility for the composer to use a 'virtual' ensemble or orchestra to produce a demo-version of the final score in close detail and perfection and to allow changes until the very last minute. Apart from using this technology in the actual composition process, a way to obtain more control for the 'people in charge' on the concept of music for audiovisual media and its outcomes, is the use of so-called temp track or temp music (the expression 'temp track' stands for the use of existing music during postproduction to create a temporary score

for a scene, a film or any other audiovisual production) [1]. The process of developing a concept for a score through the use of temp track can be done with or without the composer. Both possibilities however end up with a final temp track that is the model for the appropriate music for this specific scene or film or TV-commercial because 'it works'. In applying this model, the outcome is often an assignment for the composer to create a so-called sound-alike, which is a copy of the original temp track (within the limitations set by the copyright). Such an assignment doesn't exactly appeal to the creativity of the composer and is one of the reasons for composers[2] to reject the use of temp track in the scoring process. Despite this rejection, the use of temp track has become a regular 'tool' in media production.

Another important element with regard to the role and function of music in the media industry is the emancipation of sound. Ever since Debussy taught us that timbre is as important as pitch, the actual 'sound' of film music became more important as digital technology allowed a far better reproduction of film sound. And when the same technology opened up the world of electronic sounds and electronic music, the possibilities for sound to be used in and as film music have become endless.

3.3. Consequences for education.

Despite these developments, professional education in composition for media usually focuses on music for film using the traditional symphony orchestra. Other musical genres and/or other contexts that combine visuals, sound and music like the advertisement industry or the game industry are most of times not taken into account. Frequently the perspective of these courses and curricula is that of the composer as an autonomous artist.

The Utrecht School of Music and Technology regards the soundtrack in audiovisuals not as a combination of separate disciplines but as a coherent object that is made up of the ingredients called dialogue, music, sound effects and sound atmospheres. The curriculum composition for media aims therefore at compositional skills and knowledge that are not restricted to certain musical genres but refer to specific principles and methods that are related to the role and function of music and sound in audiovisual media.

Another important item in the curriculum is the contextual awareness that refers to the understanding of specific contexts, its participants and the related design and production processes. Such awareness is needed to keep up with the ongoing developments as described in the first paragraphs of this article.

In addition cognitive skills are needed to understand the design and production processes (not in the least one's own design and production process) and to be able to reflect on it, to adapt it if needed and to use it as a possible means for innovation.

Designing the curriculum composition for media around these core elements, we aim to provide the student with tools that help them succeed in building a professional practice in a rapidly changing media industry.

4. MUSIC PRODUCTION

4.1. Production and distribution.

During the past two decades several major technological innovations have fundamentally changed the music industry in general and the record industry in particular. Disruptive technologies, in particular those in the areas of networking and digital audio, have caused fundamental shifts in working-methods of music production and distribution of music. Professional music production has rapidly changed from being based on more centralised, industrial models of organisation towards decentralised, networked models of organisation. The democratisation of music production tools has caused a shift from technology towards design strategy as a distinctive aspect for music production professionals.

4.2. The physical product and the virtual.

At the same time, the move from physical product (e.g. CD) towards virtual product (e.g. mp3) has caused major changes in the music industry, changes that are still ongoing. These range among others from new business models in music production en music distribution, the shifting emphasis in commercial sectors from recorded music to live performance, the ongoing discourse on intellectual property, to the rise of new products such as in-game music and other forms of non-linear music. All these developments pose challenges to education in music production.

4.3. Consequences for education.

Traditional professional education on music production tends to focus on the technological aspects of music production, especially studio engineering. It has difficulties in dealing with theoretical and artistic aspects of popular music with its complex web of subgenres, all having a high

---

[1] According to Karlin & Wright [2] filmmakers use temp track a.o.:
- to help them screen their film for the producer(s), studio, and/or network executives and preview audiences during various stages of postproduction;
- to establish a concept for the score;
- to demonstrate that concept to the composer.

[2] Film composers in general are not very keen on temp track. A lot of them have encountered the drawbacks of temp track:
- If it works, it might be difficult to imagine a better way. And if it doesn't work, then the chances are it will have spoiled your first emotional reactions to the film. And what's worse, you may still have trouble getting it out of your head [2].

rate of circulation, while at the same time maintaining a high standard of academic integrity. Consequently traditional curricula on music production tend to display implicit or explicit preferences for certain musical genres. Traditional professional education has difficulties in dealing with creative aspects of music production; relevant for the contexts it is aiming at. It also has difficulties in dealing with the current high rate of change in the music industry, and organising a continuing and significant interaction of these changes with its curricula. It has difficulties with assessing and subsequently incorporating innovative developments in products, tools and strategies in music production into their curricula.

The curriculum Sound & Music Production is designed to reflect and respond to the changes in the music industry and their consequences for the music professional. Furthermore, it is designed to balance technological aspects of music production, design strategies and its creative aspects, the ongoing development in various sectors of the music industry and emerging new contexts of music production, and innovative developments in product and distribution. We do this by approaching music production from three angles, and directing the overlap of these strands: the engineering producer, the organising producer and the composing producer. The curriculum is student aimed, and takes as a main point the potential of each individual student as a reflective practitioner in the field of music production. We aim at a curriculum which exceeds the purely technological aspects of music production, exceeds specific musical genres and contexts, and provides students with tools which help them succeed in building their professional practice in a rapidly changing music industry.

5. RESEARCH

5.1. Software development and system design

In the curriculum for Audio Design one of the tracks is focused on music system design and music software development. Most of the students graduating in this track build a professional career in research at the well-known research centers in Europe (i.e. IRCAM Paris, MTG Barcelona) or elsewhere. A growing number of graduates find their way into system design for theatre, games etc. Because of these developments we are adapting the curriculum in this track on a yearly basis. In the recent past programming in a hardcore language like C++ and programming DSP were important topics, the focus is now widening up to system design using Super Collider and/or MAXMSP as tools working on adaptive music systems. The type of research students are educated on is shifting to R&D.

5.2. Research & music production / design

In the field of music production and music design the importance of research is developing. The field is in rapid development driven by technological, commercial and social developments. Research was not a topic dealt with by individuals or small companies until recently. We see a growing need for research *into* music design processes, methods, and ways of co-operation and business models, both mono-disciplinary and multi-disciplinary. This type of research is one of the factors that can make a music designer stand out in the crowd of users of the democratized production means. Also important is the development of methods of research *through* music design. Another import aspect of research is that building up experience, however necessary, is not enough anymore in the scattered field that music design is with average company sizes that are lowest in all creative industries. A research attitude and the willingness and, especially, the capability of music designers to share results will be one of the major forces for further development of the field.
The results of the research into and through music design have a direct relevance for the professional development of students and teaching staff. Through this research we augment and develop our knowledge of music design to be able to contribute to the development of important new fields like composition and sound design for adaptive systems, the improvement of music design processes in the creative industries, the improvement of music (design) education and the support for various social themes like inclusion, interculturality, application of design, game and play concepts in new contexts like culture tradition, education, care and the like.

6. CONCLUSIONS

The period 1985 up to present we have developed a continuously adapting educational program. This program is successful in preparing students for the existing and future professional practice. Both the program and (the knowledge and skills of) the teaching staff are being adapted on a yearly basis to the developments in the field. We have learned and gathered evidence that it is necessary to build in mechanisms to guarantee such a development, to guarantee the quality of the programs and to monitor the field of studies and practice.

7. REFERENCES

[1] Timmermans et. al. “Education in Music and Technology, a Program for a Professional Education ” in *Proceedings of the International Computer Music Conference*, Singapore, 2003, pp. 119-126.

[2] F. Karlin and R. Wright, *On the track: a guide to contemporary film scoring*. London (UK): Routledge, 2004. pp. 29 - 31

Concerts

Evening Concert Saturday May 1st 2010

Location: SETUP

ZLB

Miguel Negro

In this performance a spatial array of strings is simulated using physical modeling in Supercollider and perturbed using several digital raw signals.

Flowerpulse

Marc Sciglimpaglia

Flowerpulse is a short algorithmic composition that strives to maximize musical depth within a minimalist score structure. Composed in SuperCollider, this piece is rendered as a code poem, a block of code that defines an evolving, kaleidoscopic cloud of sonic agents, each performing a variable-speed trills over selected harmonic overtones that rise, overlap and die out like falling stars. It has a simple but dense code structure which defines a generalized pattern, that of a ”Routine” of musical agents released one after another into the soundscape, each with its own voicing and lifespan. The resulting texture is a hybrid of computer decision-making and performativity - each voice has a randomly determined element in its trill, while the rate and shape of the trill is performed in real time by performance on the mouse, or by MIDI control. In this sense, Flowerpulse strikes a balance between control and freedom, the smooth and striated, order and disarray. By coupling selective or performative compositional elements at each stage with algorithmic and stochastic processes, it seeks out the center where interplay between both spheres becomes deeply entangled, and where the result is greater than the sum of its parts.

Code Livecode Live

Marije Baalman

Livecoding the manipulation of the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, which are live coded to manipulate the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, which are live coded to manipulate the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, ...

Pd live coding

Pieter Suurmond, IOhannes m zmölnig

PD live coding and/or competition

Lunch concert Sunday May 2nd 2010

Location: SETUP

Chuck Demo

Kassen Oud

Evening Concert Sunday May 2nd 2010

Location: Theater Kikker

Inverso Cosmico

Massimo Carlentini

The composition comes from the idea of one of the hypotheses regarding the creation of the universe, dividine it into three phases: the ”Era Adronica” (the cosmic Big Bang), the ”Era Fotonica” (the processes of cooling, dominated by radiation), the ”Era Barionica” (the present one, of the gravitational type). ”Inverso Cosmico” has traced its way backwards through these aforesaid hypotheses: The ”Era Barionica” a pseudomelodic line or the lifeline mixes itself with sounds of the senses or sensations (hot, cold, dark, light, evanescence,ever present); The ”Era Fotonica” rhythmns with varying timbres or radiations that mix in sound bands or fluctuating materials; The ”Era Adronica”, the reduction and expansion of fragments of materials used previously or the Big Bang, repeated through time. The sounds of this piece of music, have been created through the modulation of frequency, and through granular synthesis.

Tombak

Madjid Tahriri

Tombak is an Iranian Percussion Instrument, capable of performing very complex rhythms. My piece tries to discover new playing techniques for the Tombak, which usually aren’t playable on this Instrument. (Such as Pitch Changes, sustained Notes, Accords, to name a few) The basic idea of the Composition is a conversation between 3 different Tombak (low,middle and high register). Each of the 3 Instruments has its own character. Tombak was realized with CSound and Ardour.

GR

Frances-Marie Uitti, Yota Morimoto

GR by cellist/composer Frances-Marie Uitti and composer Yota Morimoto is a followup to their collaborative DVD 13AL. The piece is the first of a series of works made by the authors for the 192 loudspeakers in Leiden. In this condensed transient piece, F.M. Uitti explores the metalo-klang of the novel aluminum cello, while Y. Morimoto’s electronics disintegrates the sound of the instrument into numerous sonic cells whose dynamics and constellation are governed by a cellular automaton agent. Acamar is a star at ’the end of the river,’ 120 light-years away from the Earth.

Transmutation

Jonas Foerster

I put myself in the place of an alchemist and start searching for ”musical gold”. There is no need for a specific definition of this term, because it’s all about the search and not about the discovery of some ”gold synthesis”. Thus I can be sure to stay searching and never to find what I don’t know. It is a playful and experimental handling of musical substances to see how they can conglomerate, split or regroup and in which way they do so. There exist, of course, a lot of methods to achieve this, which may vary in many aspects, e.g. in speed or complexity. The properties of the educts and products themselves do play an important role, but it is first and foremost the transformations from one aggregate state or material to another, that is brought into focus: the ”transmutation of substance”, to refer to the alchemists again.

A Very Fractal Cat, Somewhat T[h]rilled (2008-2010)

Fernando Lopez-Lezcano

This is the latest version of the second piece (after ”Cat Walk”) of a series of algorithmic performance pieces for pianos, computer and ”cat” that I started working on at the end of 2007 (the proverbial cat walking on a piano keyboard). The performer connects through a keyboard controller, four pedals and two modulation wheels to four virtual pianos both directly and through algorithms. Through the piece different note and phrase generation algorithms are triggered by the performer’s actions, including markov chains that the virtual cat uses to learn from the performer, fractal melodies, plain scales and trills and other even simpler algorithms. The sound of the pianos is heard directly, and is also processed using spectral, granular and other synthesis techniques at different points in the performance, creating spaces through which the performer moves. A surround environment is created with Ambisonics spatialization, and everything in the piece (algorithms, sound generation and processing and graphical user interface) was written in the SuperCollider language. ”A Very Fractal Cat, Somewhat T[h]rilled” is a piece for performer (a classically trained pianist), four virtual pianos, an optional midi controlled player piano (Disklavier or similar), and computer-based sound processing and synthesis. The performer uses a weighted 88 note keyboard controller, four pedals and two modulation wheels to connect directly and through a program to four virtual pianos and a player piano that render both the notes the performer plays and derived materials created in real-time by the program. The computer processing involves algorithmic generation of additional note and control events and sound synthesis, processing and spatialization of the resulting audio streams. The virtual pianos used include a Yamaha C7, Steinway, Bosendorfer and a Cage prepared piano. Sound processing includes granular synthesis and frequency domain transformations that affect (in certain sections of the piece) the sound of the virtual pianos. Additive synthesis is sometimes used to augment and emphasize the pianos sounds. The piece has been (and still is) evolving since the end of 2008 when the first versions of the program and score were created. It has been performed in concert several times and a previous (older) version was submitted and performed in the LAC2009 concerts.

Landschappen
Augusto Meijer

'Landschappen' ('Landscapes') is a spacious tape piece, meant to be a sonic representation of Dutch Landscapes. There is a lot of inspiration & admiration to find in these endlessly flat, empty spaces. There's been a need to express these landscapes in music because of this. So, it is really these typical impressions of emptiness and endlessness in Dutch landscapes, which form the basic concept of the piece.
`http://augustomeijer.wordpress.com`

The Cake
Laurens van der Wee

The Cake is a sound based improvisation machine

Lunch concert Monday May 3rd 2010
Location: SETUP

Playing music lazily on the bank of a river while listening to the bells of a far away city
Louigi Verona

Using loops and a midi controller with an Irish low whistle played live.

CSound Demo
Andrés Cabrera
Demonstration of CSound

Evening Concert Monday May 3rd 2010
Location: SJU Jazzpodium

Playing music lazily on the bank of a river while listening to the bells of a far away city
Louigi Verona

Using loops and a midi controller with an Irish low whistle played live.

Frozen Images

Wanda & Nova deViator

The process of freezing is a break between movements. As such, the structure of the concert, perforated with suspensions - breaks between points, is an emergence of a performance of frozen images, full of texts, contemporary electronic rhythms, exasperated guitars, noisy oscillations, and hypnotic bass lines. Along with the performers' actions, tactile interfaces, and moving pictures, the performance raises questions about hypersexualisation and pornification, fetishisation in consumerism, mechanisms of the image and visual culture, idealisation of love, and the meaning of art and culture. Movement, text, and frozen images open up in their primary way precisely based on contemporary organised noise. The music, ambivalently contextualised through the video image and movement, stretches out to the body and its vibration, rational and affective. Frozen images get broken by the vibration of the word and the moving of the actual flesh/body in all its resistance.

Elektronengehirn: Geburt Der Kunst (Satz 1 und 2)

Malte Steiner

The piece Geburt Der Kunst (Birth Of Art) was composed 2010 for the birthday of art, which was declared by French artist Robert Filliou to be on the 17. January. (http://www.artsbirthday.net) The concert was performed at Esc Im Labor in Graz, Austria, conducted remotely via the internet from Steiners studio in Hamburg, Germany. The composition was created with Pure Data and separates the control from the synthesis part, both communicating via OSC. Steiner just sent controldata via the Internet, no streaming data. The concert at LAC 2010 happens on location without network. The second part, Satz 2, will be performed the first time. ?Elektronengehirn is an electroacoustic project by Malte Steiner, founded in 1996. For soundcreation only software based synthesis and processing is used, Steiner started with CSound, custom software and soon Pure Data. So far this project released 2 CDs, a netlabel release and several contributions to compilations and collaborations. Steiner is involved in several open source projects, like the 'Minicomputer' synthesizer for Linux.
`http://www.block4.com - http://www.elektronengehirn.de`

SuperCollider live coding jam session

Marije Baalman

Supercollider Live Coding Event. The format is under construction and will be somewhere between competitive and communal livecoding.

DS-10 Denominator

Rutger Muller

Lovis

Myrthe van de Weetering	Electric violin, Melodica, Piano
Mark IJzerman	Laptop, Monome, Piano
Joep Vermolen	Cello, Metallophone
Robin Koek	Melodica, Metallophone, various small objects

The 150-Years-of-Music-Technology Composition Competition

Than van Nispen tot Pannerden

Workshops

A bird's-eye view on Linux Audio
Lieven Moors

A comprehensive overview of Linux Audio and other Open Source Music Applications with several live music demos. Interesting for beginners as well as advanced users.

Developing parallel audio applications with FAUST
Yann Orlarey

While the number of cores of our CPUs is expected to double every new generation, writing efficient parallel applications that can benefit from all these cores, remain a complex and highly technical task. The problem is even more complex for real-time audio applications that require low latencies and thus relatively fine-grained parallelism.
In order to facilitate the development of parallel audio application, the FAUST compiler developed at GRAME provides two powerful options to automatically produce parallel code. The first one is based on the OpenMP standard, while the second one uses Posix pthreads directly. This hands on demo will give the audience the opportunity to discover these parallelization facilities, their limits and their benefits, on concrete examples of audio applications.

Pro Audio vs. Consumer Audio: Discussion & Work Group
Lennart Poettering

Supercollider for starters
Marije Baalman, Jan Trutzschler, Henry Vega, Jan-Kees van Kampen

SuperCollider has been up and running on Linux since shortly after it was released as open source software. In the past two years, it has become really easy to install on Linux, and is packaged in distributions such as PlanetCCRMA and Pure:Dyne. On Linux there is a choice between three editors: emacs, vim and gedit.

Supercollider advanced
Marije Baalman, Jan Trutzschler, Henry Vega, Jan-Kees van Kampen

For more advanced users of SuperCollider we offer another workshop covering specific advanced topics. Topics could include:

- Input and output to devices or other programs
- Patterns (a different paradigm of composing based on musical sequences)
- Extending SuperCollider (writing your own classes)
- Livecoding with SuperCollider
- Using the JITLib

Chuck
Kassen Oud

A session on the ChucK programming language. How a more intimate familiarity with the workings behind synthesis can increase our options and how a DIY approach to instruments may force us to think more deeply about what we ourselves want, musically (as opposed to what's offered commercially).

Using ambisonics as a production format in ardour
Jörn Nettingsmeier

The paper describes a pop production in mixed-order Ambisonics, with an ambient sound field recording augmented with spot microphones panned in third order. After a brief introduction to the hard- and software toolchain, a number of miking and blending techniques will be discussed, geared towards the capturing (or faking of) subtle natural ambience and good imaging. I will then describe the expected struggle to make the resulting mix loud enough for commercial use while retaining a natural and pleasant sound stage and as much dynamics as possible.

Making Waves
Micz Flor, Adam Thomas

A workshop introducing Campcaster, an open source radio management application for use by both small and large radio stations (yes, real radio stations, not internet radio) to schedule radio shows. Learn the first steps of operating a radio station, including scheduling, live studio broadcast, play-out and even remote automation and networking via the web, all using free software. The workshop will take participants through hardware set-ups, software installations and studio configurations, resulting in the creation of a fully-functioning Linux Audio Conference station.

Creative Commons
Björn Wijers

Be informed about the options and implications for licensing your work using Creative Commons.

Live Coding with QuteCsound
Andrés Cabrera

This workshop will explore the Live Coding capabilities of QuteCsound for generating score events for a running instance of Csound, through the use of the Live Event Panel and the QuteSheet python API. It will show how the QuteCsound frontend can serve as a python IDE for the processing of note events to be generated, transformed, sent live, or looped for a running Csound process.